11

State Management
with Redux

I have not failed. I've just found 10,000 ways that won't work.

- Thomas A. Edison

Contents

How To Use This Document

Bits of text in red are links and should be clicked at every opportunity. Bits of text in `monospaced green` represent code. Most the other text is just text: you should probably read it.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a [View on GitHub] link underneath them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I've switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn't need to take any additional notes. However, I know that this doesn't work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into "smart-quotes". These can be almost impossible to spot in a text-editor, but will completely break your code.

Chapter 1

Application Architecture

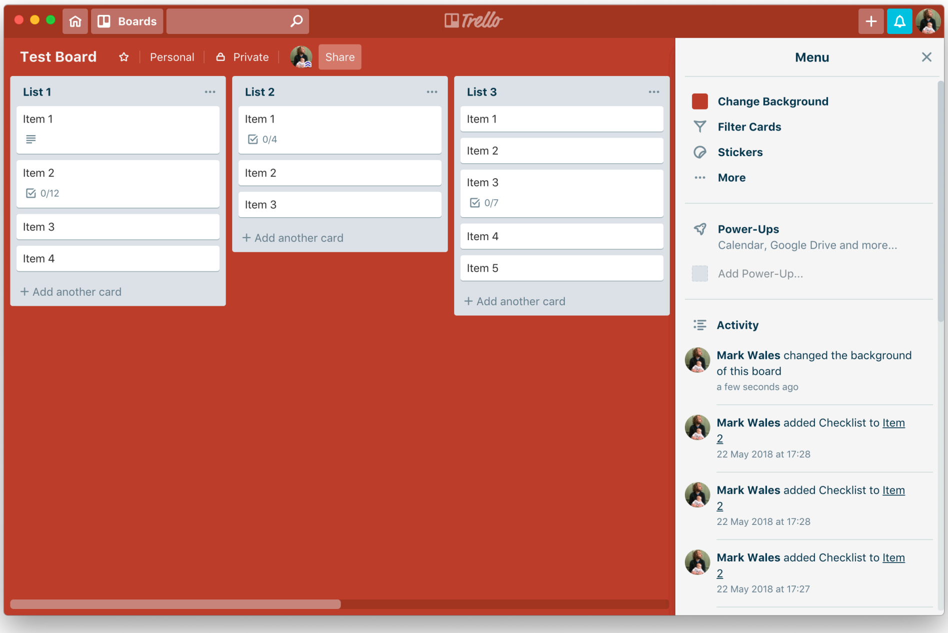
A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

- Gall's Law

Good architecture is necessary to give programs enough structure to be able to grow large without collapsing into a puddle of confusion

- Douglas Crockford

All the components that we've built so far have been very simple and, with the exception of the lifting state examples, have not needed to interact with any other components in our app. But consider a web-app like Trello:



Mark Wales changed the background of this board
a few seconds ago

Mark Wales added Checklist to Item 2
22 May 2018 at 17:28

Mark Wales added Checklist to Item 2
22 May 2018 at 17:28

Mark Wales added Checklist to Item 2
22 May 2018 at 17:27

A Trello board

It consists of many different components: a board component, a menu component, list components, list item components, and many more. Most of these components need to affect and know stuff about many other components. For example, the "Activity" section needs to update whenever the user does pretty much anything. And the board background needs to change if the "Change Background" option is used in the Menu.

Because React has one-way data flow, we know that the only way for components to interact with each other is to store the state higher-up the component hierarchy ("lifting" state). This might at first seem like a limitation: if we had two-way data flow then this would not be necessary. But, remember, once you introduce two-way data flow components are no longer reusable as they become tied to the components that they are used in. If you have two-way data flow throughout your entire app it becomes impossible to know where certain bits of logic belong and it quickly becomes a mess of interconnected components that all need to know about all the other components.

Hopefully by this point in the course you're starting to see a common theme: write small easy to understand "black boxes" and then combine them to create more complex behaviour:

- Functions: create functions that do a single thing well, once you've written them you just need to know *what* they do, not *how* they do it
- PHP Interfaces: create classes with a predictable way of interacting with them (method signatures), once you've written them all you need to know is *what* data to pass to them, not *how* they work inside
- React Components: create simple UI components and then combine them together to create more complex UIs, all you need to know is *what* props to give to the component not *how* they are used internally

There's a reason this keeps coming up: brains are pretty incredible things[1], but they're still fairly limited. You can't be expected to hold how every single aspect of your program works in your head. So we find ways of writing code where once we've got something working we don't need to worry about *how* it works, just *what* it needs to work.

The key concept of a React + Redux app is that we can build complex apps out of simple components. It's not uncommon, when faced with building a complex app, to end up with *complicated* code: files where it's hard to tell what anything is doing. But, ideally, *no single part of our app should be difficult to understand on its own*. However, the joining up of all these small parts can produce very complex behaviour.

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies*
>
> \- C.A.R. Hoare

[1]Citation needed

1.1 Additional Resources

- Wikipedia: Separation of Concerns
- MVC - a common pattern for separation of concerns
- MVC Architecture for JavaScript Applications - an old article, but it explains the concept well
- 10 Things You Will Eventually Learn About JS Projects
- Functional Programming Fundamentals

Chapter 2

Redux

Redux is a library that handles application state management. It's one of the most common ways to handle application state with React (along with MobX). You can also use Redux with other view libraries or even with plain old DOM code.

At first Redux can seem quite daunting and complicated, but it manages to capture most of the complexities of app state management in only a few key concepts.

Redux consists of three parts:

1. **The Store**: The store keeps track of the current **state**. We can **subscribe** to it to listen for changes and we can **dispatch** actions to it. We can't change the state directly, only via the store.
2. **Actions**: Actions are simple messages (POJOs[1]) that we **dispatch** to the store.
3. **Reducers**: Reducers take an action and **transform** the current state in some way. This updates the store which then lets any subscribers know that it's changed.

Basically Redux takes the idea of lifting state to its logical conclusion: all our different components need to know about each other, so we lift the state to the top most level.

[1]Plain Old JavaScript Objects

Avoiding Redux

Because it takes a while to learn, there is a temptation to avoid something like Redux and just do all of your state management within React components. This is all well and good for basic apps with a few components which don't have a lot of shared state, but it becomes incredibly complicated once you start working on more complex apps. By using Redux from the start you can avoid these issues.

You also find that every month or so there's a new app state management library that is like Redux but tries to make some aspect of it simpler. The examples given for these libraries do indeed seem to simplify some aspect of Redux… but they only ever show examples using a few components.

Once you start writing bigger apps you'll really appreciate the flexibility that Redux gives you.

∞

Let's build our first proper app: one where the data and views are separate. We're going to use a really basic `<App>` component:

```
import React from "react";

const App = () => (
  <React.Fragment>
    <h1>iCounter</h1>

    <p className="well" />

    <div>
      <button className="btn btn-primary">+</button>
      <button className="btn btn-primary">-</button>
      <button className="btn btn-danger">Reset</button>
    </div>
  </React.Fragment>
);

export default App;
```

[View code on GitHub]

We want to get it so that when we click on the buttons they change the value of the counter. But our state is going to live inside Redux, not inside the component itself.

2.1 Initial State

First we need to decide on the shape of our state and create an **initial state**. This is much like the initial state for a React component, except that it represents the state for the *entire app*.

In this case we just need to store a single number:

```
// the initial state
const initial = {
  count: 1,
};
```

We create an object literal with a `count` property and set it equal to `1`.

As our app gains more functionality we'll need to add additional properties to our state.

2.2 Reducers

Next we need to decide how we want to be able to change the state. We put these inside a **reducer**. The reducer is called by the store whenever an action is dispatched. It gets given the *current* state as well as the action that was dispatched.

This is the *only* way that the state can be changed. This means that just by looking at the initial state and the reducer it's possible to work out all possible ways in which the state can be changed.

Actions, which we'll get to shortly, are just an object literal with a `type` property. By passing different `type`s, we can transform the state in different ways. For now, let's just accept an `increment` type, which will add `1` to the current `count`:

```
// the reducer gets given the current version of the state
// and the action that called it, which has a "type" property
const reducer = (state, action) => {
  // depending on what the action's type property is
  // we can do different things
  switch (action.type) {
    // using object spread to create a new object
    // with the same properties
    case "increment": return { ...state, value: state.value + 1 };
    default: return state;
  }
};
```

[View code on GitHub]

The reducer has to return a *new* state object: you can't just update a property and then return it. This is where the spread operator comes in handy. If you ever think you've updated the state but nothing re-renders, it's probably because you've updated an existing data structure, rather than returning a new one.

The reducer is just a regular JavaScript function, so we can test that it works without needing to use Redux:

```
// call the reducer, passing in a valid version of the state
// and an object literal with a "type" property
console.log(
  // if the previous value is 1, we should get 2 if we increment
  reducer({ count: 1 }, { type: "increment" }), // { count: 2 }
  // if the previous count is 10, we should get 11 if we increment
  reducer({ count: 10 }, { type: "increment" }), // { count: 11 }
  // if we pass an action it doesn't recognise, we should get
  // the given state back
  reducer({ count: 10 }, { type: "fishsticks" }) // { count: 10 }
);
```

[View code on GitHub]

2.3 The Store

Once we've got an initial state and a reducer, we can create the **store**. This is a wrapper around our state: it allows us to dispatch actions, which will change the state, and subscribe to any changes, so we can get the latest state. Importantly, we can't access the state directly as it's wrapped inside the store.

First install Redux:

```
npm install redux
```

Then import `createStore`:

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

We can then pass the initial state and reducer to `createStore`, which gives us back a new Redux store:

```
const store = createStore(reducer, initial);
```

[View code on GitHub]

2.4 Subscribing

Next, we will **subscribe** to the store, which will let us know when the state changes. We pass it a function that will run whenever the state is changed. This gives us a single place to deal with anything that needs to be updated on the page.

We can use the `getState()` method of `store` to *get* the current state. This doesn't allow us to change the state - we can only do that with actions.

```
// subscribe to any changes
store.subscribe(() => {
  // get the current state using the getState method
  // we can get the state, but not update it
  let state = store.getState();

  // for now, just log the new count
  console.log(state.count);
});
```

[View code on GitHub]

2.5 Actions

Finally, we can **dispatch** some actions: each time we dispatch an action, the store runs the reducer function for us, which looks at the action `type` property and transforms the state appropriately. Once the reducer has returned a value, the function we passed to `store.subscribe()` will be run and we can update our page appropriately.

```
// dispatching an action
store.dispatch({ type: "increment" });
```

[View code on GitHub]

∞

Putting it all together in the `index.js` file:

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

const initial = {
  count: 1,
};

const reducer = (state, action) => {
  switch (action.type) {
    case "increment": return { ...state, count: state.count + 1 };
    default: return state;
  }
};

const store = createStore(reducer, initial);

store.subscribe(() => {
  let state = store.getState();
  console.log(state.count);
});

store.dispatch({ type: "increment" });
```

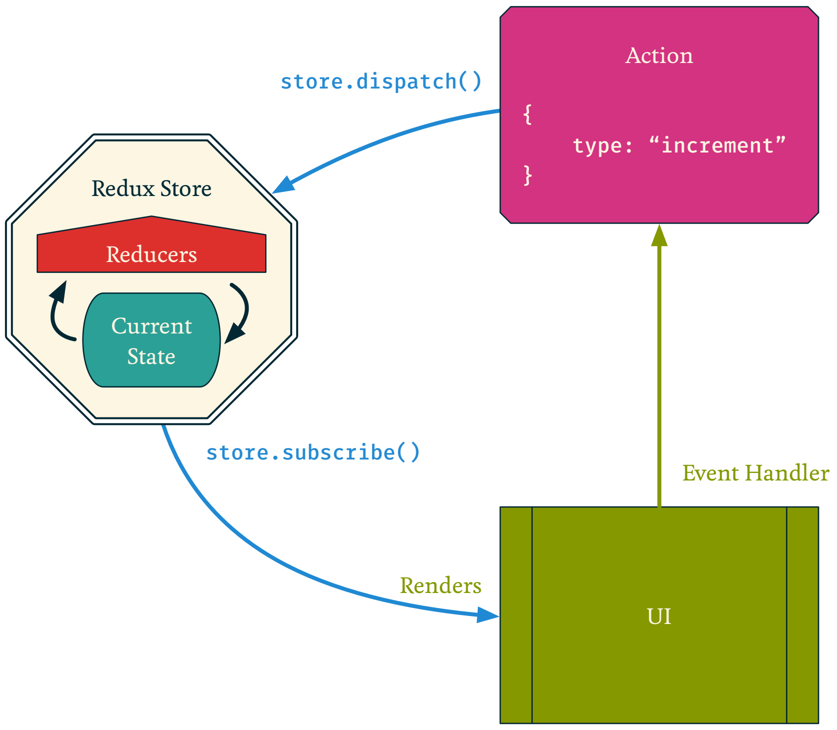
[View code on GitHub]

If you open this in the browser you should see `2` logged in the console: the value of the counter after the increment action has run. Try dispatching the `increment` action multiple times and you'll see that a new value is logged each time.

To recap:

- We setup an **initial state**[2]
- We create a **reducer**, which takes different actions and **transforms** our data based on the action's `type` property
- We create our **store** passing it our reducer and the initial state
- We **subscribe** to the store, so that we can respond whenever the state changes (usually by re-rendering the view)
- We **dispatch** actions to the store to make changes to our state (usually in event handlers)

If you've understood up to here then you understand Redux. The harder part is getting it to work with React!



The Redux Architecture

[2] Remember, the initial state is just the values the state will have when we first load the app. Once we start dispatching actions the state will change, but the store keeps track of it for us.

2.6 Working with React

Now let's get Redux working with React. We'll need to tell React to re-render the `App` component whenever the state changes. We can do this by calling `ReactDOM`'s `render()` method inside our `subscribe` function:

```
// we update subscribe to call the ReactDOM.render
// method whenever the state changes
const render = () => {
  let state = store.getState();

  // pass in state.value as a value prop
  ReactDOM.render(
    <App count={ state.count } />,
    document.getElementById("root")
  );
};

store.subscribe(render); // render when state changes
render(); // render when page loads using initial state
```

[View code on GitHub]

We can pass the current `count` through using a prop on `<App>`. So we need to update the `<App>` component to accept the `count` prop:

```
const App = ({ count }) => (
  { /* ...etc. */ }

  <p className="well">{ count }</p>

  { /* ...etc. */ }
);
```

[View code on GitHub]

We'll also need to handle the increment button click. Whenever the user clicks the "+" button we want to dispatch an `increment` action. We can pass this in as an anonymous function for now:

```
ReactDOM.render(
  <App
    count={ state.count }
    handleIncrement={ () => store.dispatch({ type: "increment" }) }
  />,
  document.getElementById("root")
);
```

[View code on GitHub]

And then add this as an event handler inside our `<App>`:

```
// add it as a prop
const App = ({ count, handleIncrement }) => (
  { /* ...etc. */ }

  <button
    className="btn btn-primary"
    onClick={ handleIncrement }
  >+</button>

  { /* ...etc. */ }
);
```

[View code on GitHub]

Generally we'll only use `dispatch` for event handlers: if the user hasn't done anything then there's not normally a good reason for the state to change. If you need to make multiple changes to the state in response to a specific action then you should put all of that code in the reducer.

Redux Dev Tools

There's a really useful browser plugin called Redux Dev Tools.

Once you've added it to your browser you need to add an extra argument to your `createStore` call, which links everything up:

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

[View code on GitHub]

The Redux Dev Tools let you look at what's going on with the Redux store: which actions have been dispatched, how the state has changed each time, and the current state. It even lets you change the history of your app by skipping actions or rewinding time. This is only possible because the app's state is fully controlled by Redux.

∞

Next, let's get the decrement and reset buttons working.

To add these behaviours we just need to add two more reducers:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "increment": return { ...state, count: state.count + 1 };
    case "decrement": return { ...state, count: state.count - 1 };
    case "reset": return initial;
    default: return state;
  }
};
```

[View code on GitHub]

The `decrement` action simply subtracts `1` from `count`. The `reset` action sets the state back to the value of `initial`: resetting everything back to the initial values.[3]

Now add props to pass in a dispatch function for each action:

```
<App
  // ...etc.
  handleDecrement={ () => store.dispatch({ type: "decrement" }) }
  handleReset={ () => store.dispatch({ type: "reset" }) }
/>,
```

[View code on GitHub]

And add the `onClick` event handlers for each button in `<App>`:

```
const App = ({
  // ...etc.
  handleDecrement,
  handleReset,
}) => (
  <React.Fragment>
    { /* ...etc. */ }

    <button
      onClick={ handleDecrement}
      className="btn btn-primary"
    >-</button>

    <button
      onClick={ handleReset }
      className="btn btn-danger"
    >Reset</button>

    { /* ...etc. */ }
  </React.Fragment>
);
```

[View code on GitHub]

[3]This is really neat. It lets you reset the *entire* state of the app in one line. If you've spread your state logic around lots of separate components, this is incredibly complicated to do. In the pre-Redux days I once spent an entire day trying to fully reset the state of my entire app, spread around hundreds of files. I eventually gave up and just told the browser to refresh the page.

Because actions are just object literals we can pass along any additional information that we like to the reducer. This is often referred to as the action's **payload**. For example we could combine the `increment` and `decrement` actions by instead passing along a number to change by:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "change": return {
      ...state,
      // change the count by the amount property of the action
      count: state.count + action.amount,
    };
    case "reset": return initial;
    default: return state;
  }
};
```

[View code on GitHub]

Now we can pass in a `amount` property to change the value by any amount we like:

```
handleIncrement={() => store.dispatch({ type: "change", amount: 1 })}
handleDecrement={() => store.dispatch({ type: "change", amount: -1 })}
```

[View code on GitHub]

It's also fairly common practice to pull complex functions out of the reducer to keep our code a bit clearer:

```
// the change code pulled out into its own function
const change = (state, action) => {
  return { ...state, count: state.count + action.amount };
};

const reducer = (state, action) => {
  switch (action.type) {
    // call the change function and pass it the state and the action
    case "change": return change(state, action);
    case "reset": return initial;
    default: return state;
  }
};
```

[View code on GitHub]

We can now use object destructuring to tidy up the code:

```
// we only need the amount property at this point
const change = (state, { amount }) => {
  return { ...state, count: state.count + amount };
};
```

[View code on GitHub]

2.7 Persistence

Finally, it would be nice if when we refresh the page we don't lose the app state. We can do this fairly easily by storing the state in localStorage.

First, install the package:

```
npm install redux-localstorage
```

Then we'll need to update the `import` statements in `index.js`:

```
import { createStore, compose } from "redux";
import persistState from "redux-localstorage";
```

Finally, we need to change how we create the store, to include the localstorage middleware. This is a little bit messy because we want to keep using the Redux developer tools:

```
const composeEnhancers =
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const store = createStore(
    reducer,
    initial,
    composeEnhancers(persistState())
);
```

Now, if we refresh the page you can see that the app state sticks around.

2.8 What Goes in State?

When we were writing React components we said it's best to keep your state as slimline as possible: if you can work something out from some other bit of state then you don't need to store it in state as well. This is fine for components, as everything is all in one place, but it's not the case for application state.

With application state we want to keep *all* of our **business logic** in one place. Any calculations that need doing with the data of the app should be done inside the reducers. If you have to work something out inside a component then your UI knows too much and is probably tied into your specific use case.

One way that I often think about whether something belongs in application state or not is to imagine that the app I'm building needs both a regular React app in English and also a React Native version in Spanish. If both versions can share the same application state without needing to repeat any logic inside the equivalent components then you've got the balance right.

2.9 Dear God Why?

You might be asking yourself, "Why did we just go to all of that effort to get a bloomin' counter working?" And it would, of course, be a bit extreme to use Redux for this particular case. But once your apps start to get more complex, with lots of interconnected components, you'll see that it has many advantages:

- All of your business logic lives in one place: all of the code that is specific to your app lives in the reducers, so you always know where to look for it
- Your React components are just for UI: the components don't need to know anything about what they'll be used for, which means they can be used in different apps without making any changes to them
- You could get rid of React: if you really wanted to you could switch from React to Vue.js (or whatever view framework is currently in vogue) without needing to change any of your application logic

- Your state lives in one place: if you want to persist the current application state it's simply a matter of getting it out of the store. If it was spread between lots of separate components this would be very hard to do. It also allows you to reset the application state with one line of code.

For example, consider changing the background in Trello. If there's a property in the app state called `backgroundImage` then all we have to do to change the background is update this one property: every component that needs to can get the new value. If this information was stored in the `<Board>` component, then it's not obvious how a completely separate component in the Menu could affect this.

Once you get used to the Redux way of doing things (which *will* take a while), you'll begin to appreciate the predictability that it offers.

2.10 Additional Resources

- Redux: Basics
- Egghead: Getting Started with Redux
- A Practical Guide to Redux
- An Unforgettable Way to Learn Redux
- Understanding Redux
- What is Redux? A Designer's Guide

Chapter 3

React Redux

We've got Redux working with React, but there are a few issues:

- Currently if our app has multiple components we'd have to pass down values from state *and* the event handlers using props, which could get messy in a more complex app. It would also mean some components need to accept props purely to pass them through to other components, which makes them less reusable.

- The entire React app re-renders whenever the state updates: this is very inefficient, as it's likely that the state change only affects a few components. It would be better if we only re-render the components that use the specific bit of state that has changed.

We *could* pass `store` down to all our components using props so that they can read the state and dispatch actions, but adding a `store` prop to every single component will get cumbersome even in a small app. It would also make all of our components dependent on Redux, which makes them much less reusable.

That's where the `react-redux` library comes in: its sole purpose is to hook up our Redux store with our React components.

```
npm install react-redux
```

The `react-redux` library wraps our entire app in a `<Provider>` component, which does some stuff with the `children` prop behind the scenes so that we can **connect** to `store` from anywhere in our app.

We need to `import` the `<Provider>` component from `react-redux` and wrap it around our entire app. Inside `index.js`:

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

```
// wrap Router with Provider
ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById("root"),
);
```

[View code on GitHub]

The `<Provider>` attaches the Redux store to the `connect()` function. In the next two chapters we'll look at how to use this to subscribe to the store, get data from state, and dispatch actions from any component in our app.

Now we don't need to manually subscribe to the store, as the `<Provider>` does this for us. So we can remove the `render` function from `index.js`.[1]

While we're in `index.js` it might also be worth splitting out some of the Redux code into separate files. We'll put the reducers into `src/data/reducers.js` and the initial state into `src/data/initial.js`:[2]

```
import initial from "./data/initial";
import reducers from "./data/reducers";
```

[1] If you were starting from scratch now, you would just use the `<Provider>` straight away and not write the render code at all.
[2] As with the `<Provider>`, we'd probably do this from the beginning when starting an app from scratch.

The Context API

The Context API was added to React in version 16.4. It was quickly followed by a large number of blog posts suggesting that now we had the Context API we didn't need Redux anymore.

This conflates Redux (the state management library) with React Redux (the library for joining up React with Redux). The Context API *can* be used in place of React Redux to pass information between different layers of the component hierarchy without directly passing it through every intermediate layer. However, it does nothing to help with state management.

Since version 6.0 of React Redux it actually uses the Context API under the hood. The Context API is an **implementation detail**: if you use React Redux version 6.0 or above then you're using it, if you use an older version then you're not - but your usage of React Redux would be exactly the same in both cases.

Personally I prefer using React Redux to using the Context API as it allows you to write your code using just functions. Once you start using Context you have to start worrying about `this`. And nobody wants that.

> *Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.*
>
> - The React Docs

3.1 Additional Resources

- React Redux
- The History and Implementation of React Redux
- React: Context

Chapter 4

React Redux: Subscribing

First things first, let's break up our iCounter app into a few separate components. This is not strictly necessary, but it will make the code examples more like an actual app made up of lots of components.

We're going to put all our JSX files into a directory called `components`, this will keep the `src` directory clean. First move `App.jsx` into `src/components` and make sure you update the `import` in `index.jsx`.

We'll create a `<Value>` component in `src/components/Value/index.jsx`:

```
import React from "react";

const Value = ({
  value,
}) => (
  <p className="well">{ value }</p>
);

export default Value;
```

[View code on GitHub]

And a `<Buttons>` component in `src/components/Buttons/index.jsx`:

```
import React from "react";

const Buttons = ({
  handleIncrement,
  handleDecrement,
  handleReset,
}) => (
  <div>
    <button
      onClick={ handleIncrement }
      className="btn btn-primary"
    >+</button>

    <button
      onClick={ handleDecrement }
      className="btn btn-primary"
    >-</button>

    <button
      onClick={ handleReset }
      className="btn btn-danger"
    >Reset</button>
  </div>
);

export default Buttons;
```

[View code on GitHub]

Importing `index.jsx`

If we have a file called `index.js` or `index.jsx` in a directory then we can import it using `import` just by using the directory name, just as you can with an `index.html` file.

We can then update the `<App>` component to use it:

```
import React from "react";

import Value from "./Value";
import Buttons from "./Buttons";

const App = ({
  value,
  handleIncrement,
  handleDecrement,
  handleReset,
}) => (
  <React.Fragment>
    <h1>iCounter</h1>

    <Value value={ value } />

    <Buttons
      handleIncrement={ handleIncrement }
      handleDecrement={ handleDecrement }
      handleReset={ handleReset }
    />
  </React.Fragment>
);

export default App;
```

[View code on GitHub]

4.1 Connect

Now we can use React Redux's `connect()` function to join up our components with the store. This allows us to get the current state and pass it into our component. It will also `subscribe` to the store for us to re-render the component whenever the state changes.

We'll need to create a **container component** which uses the `connect` function.

First, rename your `<Value>` component from `src/components/Value/index.jsx` to `src/components/Value/Value.jsx` (you'll see why in a second). This will break your app temporarily, but we'll fix it shortly.

Now create a new file `src/components/Values/index.js` (`js` *not* `jsx` - this file won't contain any JSX). This is going to hold the container component.

The idea of a container component is to wrap around a regular React component and pass in some props. This is useful because the container component can use the connect function to get values from the store. It returns a regular React component that we can use elsewhere.

```
// import connect from React Redux
// this will talk to the Provider component, which has
// wrapped the entire app, so that it can access the store
import { connect } from "react-redux";

// import the React component that we want to wrap
// in the same directory, so path is short
import Value from "./Value";

// mapStateToProps: maps the current state (from the store)
// to the props that get passed into the wrapped component
// needs to return an object literal which gets merged in
// to any other props being passed in
const mapStateToProps = state => {
  return {
    value: state.value,
  };
};

// use the connect function to connect mapStateToProps
// to the React component we want to wrap
// returns a new React component
export default connect(mapStateToProps)(Value);
```

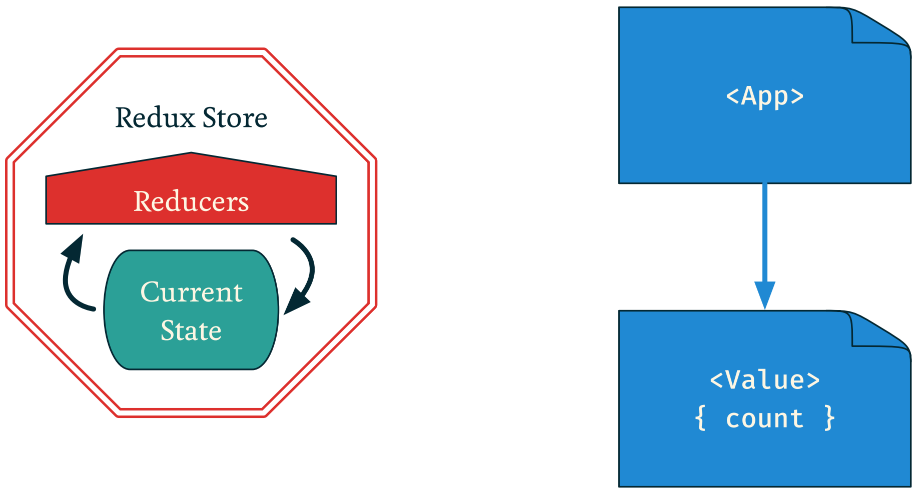
[View code on GitHub]

First we import the connect function. This talks to the <Provider> wrapper that we put around the entire app and which has access to the Redux store. We also import in the component that we're interested in giving access to the store, in this case the <Value> component.

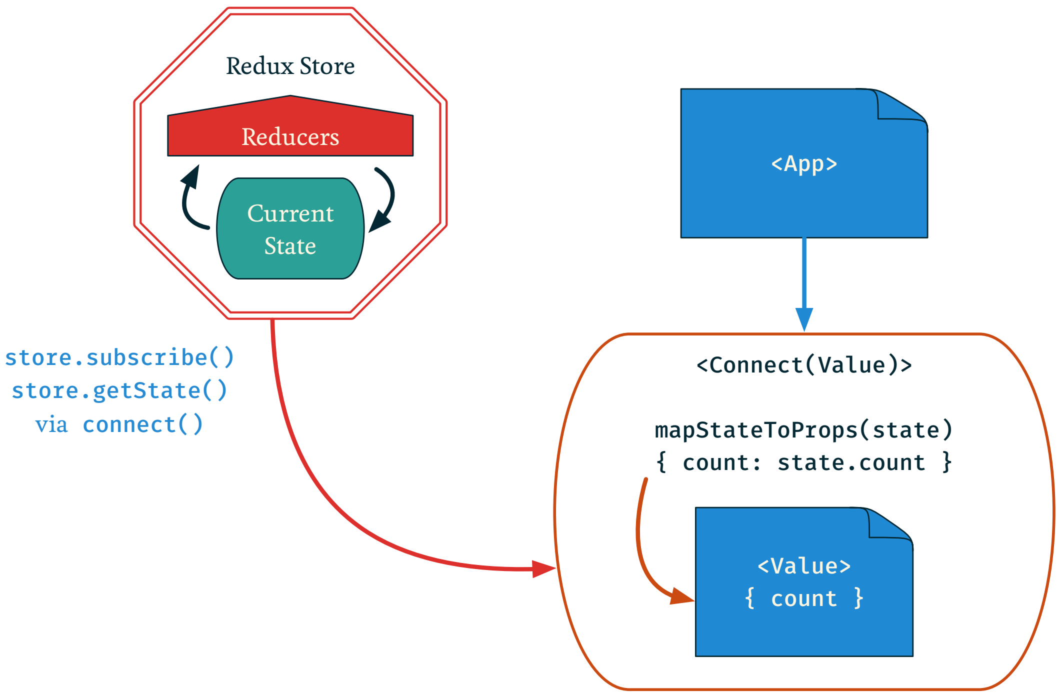
Then we create a function called mapStateToProps.[1] We're going to pass this function to connect, which will then call it for us whenever the Redux store updates the state (using store.subscribe()). When this happens it will pass in the latest version of the state and merge whatever is returned into the props for the wrapped component.

[1]You can call it whatever you like, it's just a variable that we pass to connect, but it's pretty standard to stick to this name.

You'll probably need to restart the webpack server at this point, as changing from `index.jsx` to `index.js` probably broke it.



Before: no way to access the Redux store



After: wrap with a component that has access to the Redux store

Working with Objects

We could actually have written `mapStateToProps` like this:

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

This is just using `=>` to return a single value. However, we need to use the brackets around the object literal, otherwise the curly braces are interpreted as opening a function block.

We've also used destructuring to get the `count` property from `state` and taken advantage of the fact that if an object key and variable name match, we can just pass the variable and JS will add the matching key for us (known as shorthand properties).

Now we can update `index.js` to not pass the `count` prop in:

```
ReactDOM.render(
  <Provider store={ store }>
    <App
      handleIncrement={ () => store.dispatch(/* ...etc. */) }
      handleDecrement={ () => store.dispatch(/* ...etc. */) }
      handleReset={ () => store.dispatch(/* ...etc. */) }
    />
  </Provider>,
  document.getElementById("root")
);
```

And we can also update `<App>` to not deal with the `count` prop (as it doesn't get passed in now anyway):

```
const App = ({
  handleIncrement,
  handleDecrement,
  handleReset,
}) => (
  <React.Fragment>
    { /* ...etc. */ }

    <Value />

    { /* ...etc. */ }
  </React.Fragment>
);

export default App;
```

[View code on GitHub]

4.2 Own Props

Sometimes it can still be useful to pass props into wrapped component. We can do this just as before:

```
<Value offset={ 5 } />
```

Any props passed to a wrapped component get passed through to the underlying component:

```
const Value = ({
  count,
  offset,
}) => (
  <p className="well">{ count + offset }</p>
);
```

You can also access props inside the container component, as they are passed in as the second argument to `mapStateToProps`:

```
const mapStateToProps = (state, { offset }) => {
  return {
    count: state.count + offset,
  };
};
```

This can save passing values all the way into the wrapped component.

4.3 Additional Resources

- React Redux: `mapStateToProps`
- Redux's Mysterious Connect Function
- 7 Tricks with Resting and Spreading JavaScript Objects

Chapter 5

React Redux: Dispatching

Next we'll want to update iCounter so that we don't have to pass the event handler functions via the `<App>` component. We'll do this in much the same way as with the `<Value>` component, except this time we're not interested in the state, we're interested in *dispatching* actions.

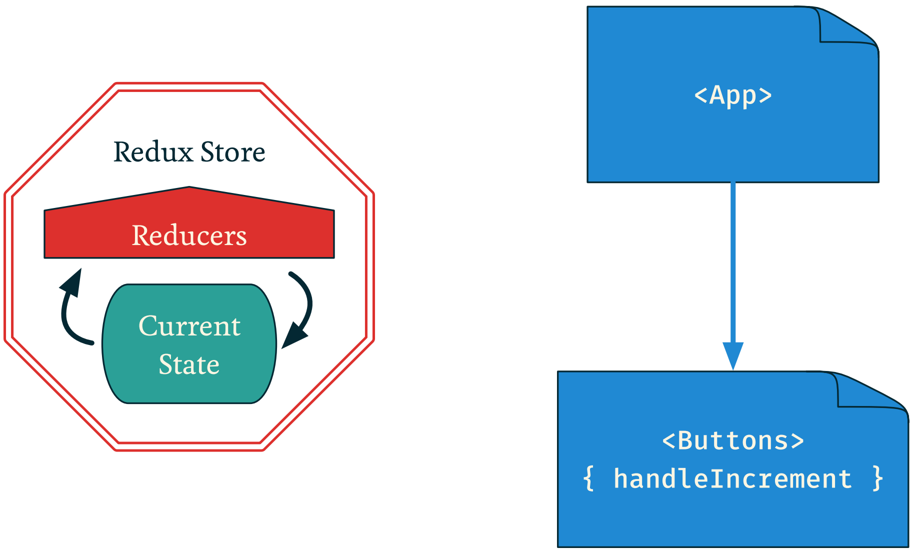
First rename the `<Buttons>` component from `src/components/Buttons/index.jsx` to `src/components/Buttons/Buttons.jsx` - which will temporarily break our app. Then we'll create a container component in `src/Buttons/index.js`. This time we're going to use the second argument of `connect` - this is normally called `mapDispatchToProps`:

```
// import connect from React Redux
// this will talk to the Provider component, which has
// wrapped the entire app, so that it can access the store
import { connect } from "react-redux";

// import the React component that we want to wrap
// in the same directory, so path is short
import Buttons from "./Buttons";

// mapDispatchToProps: passes in the store.dispatch method
// we can then create anonymous functions to call it with the
// correct actions - these get merged in to any other props being
↪  passed in
const mapDispatchToProps = dispatch => {
  return {
    handleIncrement: () => dispatch({ type: "change", amount: 1 }),
    handleDecrement: () => dispatch({ type: "change", amount: -1 }),
    handleReset: () => dispatch({ type: "reset" }),
  };
};

// use the connect function to connect mapDispatchToProps
// to the React component we want to wrap
// note that we pass null for the first argument (mapStateToProps)
// returns a new React component
export default connect(null, mapDispatchToProps)(Buttons);
```

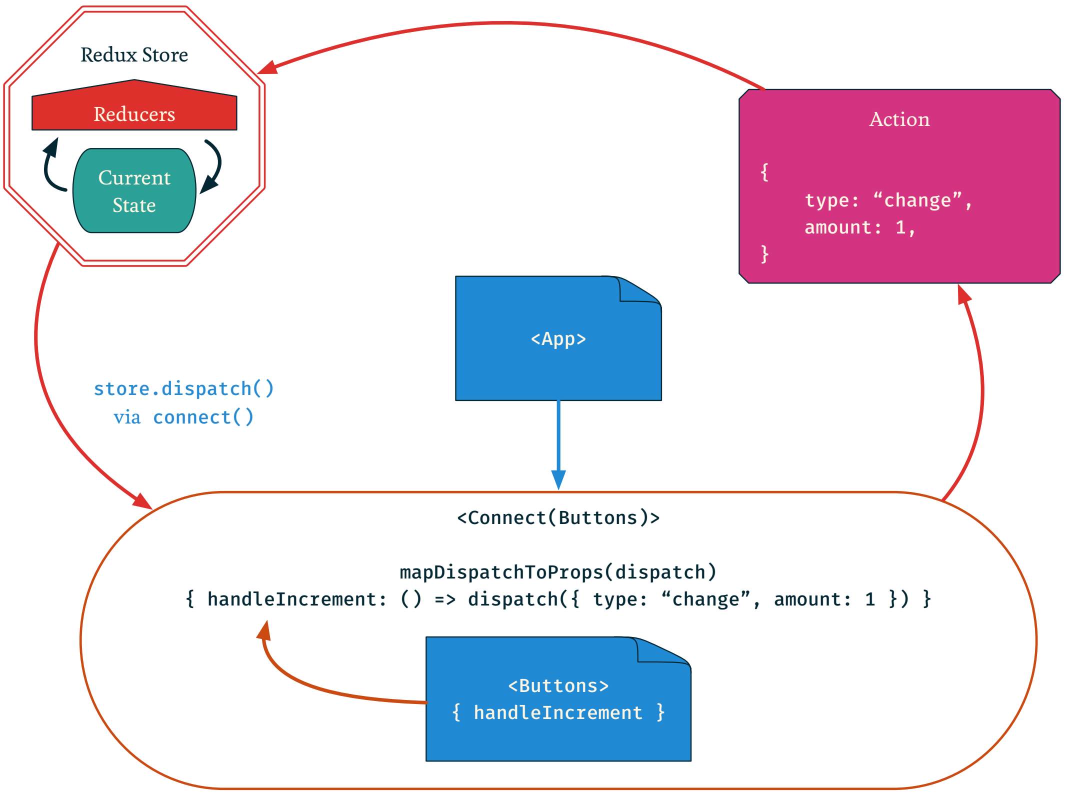
[View code on GitHub]

Much of this is the same as using `mapStateToProps`: we import in the `connect` function, import in the component that we want to pass props to, and use `connect` at the bottom to export a new component.

The two key differences are that we pass `mapDispatchToProps` in as the *second* argument to `connect` (and pass `null` in as the first) and the function that we pass it gets passed the Redux store's `dispatch` method, rather than the latest copy of the state. This allows us to dispatch actions to the store, thus triggering the reducers (which will then update the state, and cause any components using `mapStateToProps` to re-render).



Before: no way to dispatch to the Redux store



After: wrap with a component that has access to the Redux store

5.1 Action Creators

In larger apps it's not uncommon for different event handlers to dispatch the same (or similar) actions. For this reason it's common to create **Action Creators**: functions that create an action for us.

First, create a file called `src/data/actions`. Then add action creators for the two action types:[1]

```
// a change action creator
// accepts an amount variable so we can change it easily
export const change = (amount) => {
  return {
    type: "change",
```

[1]These could be made shorter using object property shorthand and skipping the `return` by wrapping in brackets.

```
    amount: amount,
  };
};

// a reset action creator
// doesn't need any additional data, so no parameters
export const reset = () => {
  return {
    type: "reset",
  };
};
```

[View code on GitHub]

We export each function separately, as we might require them in different files.

Now we can use these in the container component:

```
import { change, reset } from "../../data/actions";

const mapDispatchToProps = dispatch => {
  return {
    handleDecrement: () => dispatch(change(-1)),
    handleIncrement: () => dispatch(change(1)),
    handleReset: () => dispatch(reset()),
  };
};
```

[View code on GitHub]

All we've done is created some functions to create the action object literals for us, rather than writing them out each time.

5.2 Forms

Next we need to consider forms. Remember that in React if we have an `<input>` then we create a **controlled component**, where we use the component state to keep track of the value.

You might naturally assume that this state should be part of the application state - after all, that's where everything else has gone. But in this case it's probably better to keep the state locally: if you have lots of forms with lots of inputs it can get complicated keeping track of them all. Of course we will need to get the value out of the form *at some point*, but not until it is submitted.

The Local State Rule

A good rule of thumb for whether state belongs in the application state, or whether you should just store it locally, is to think:

> "If I refreshed the page, would I expect this to still be there?"

In the case of a form input, you probably *wouldn't* expect it to still be there on a refresh, but you *would* expect the app to have remembered the data you've added and the settings that you've changed.

Let's add a "Settings" page to our app so that we can change how much the counter goes up by:

```
import React, { Component } from "react";

class Settings extends Component {
  constructor(props) {
    super(props);

    this.state = {
      step: props.step,
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(e) {
    this.setState({ step: e.currentTarget.value });
  }

  handleSubmit(e) {
    e.preventDefault();
  }

  render() {
    let { step } = this.state;
    return (
      <form onSubmit={ this.handleSubmit }>
        <div className="form-group">
          <label>Step</label>
          <input
            className="form-control"
            onChange={ this.handleChange }
            value={ step }
          />
        </div>
        <button className="btn btn-primary">Save</button>
      </form>
    );
  }
}

export default Settings;
```

[View code on GitHub]

Currently it's just a basic controlled component with an `<input>` element and a "Save" button. On submit we just prevent default. We'll use ReactRouter to show this page when the user visits `/settings`.

We set the initial `step` state to `props.step`, so we need to pass in this value from somewhere. We'll want to store the step value in the application state, so lets add it to the state and then get it passed into the component.

First, add a `step` property to the initial state and set it to `1`:

```
const initial = {
    count: 1,
    step: 1,
};
```

Now we need to pass this value into the `<Settings>` component. We'll do this by wrapping it in a container component and use `mapStateToProps`:

```
import { connect } from "react-redux";
import Settings from "./Settings";

const mapStateToProps = state => {
  return {
    step: state.step,
  };
};

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

[View code on GitHub]

Try updating the initial state and seeing if the step value changes.

Next we need to make it so that when the form is submitted the data is dispatched to the store and updates the `step` value.

We'll need to dispatch a new type of action, so let's create an action creator for this purpose:

```
export const step = (amount) => {
  return {
    type: "step",

    // we can use the action creator to tidy up the data
    // make sure it's definitely a number before it
    // gets to the reducer
    amount: +amount,
  };
};
```

[View code on GitHub]

It's almost identical to the `change` action, except for the `type`.

Now we need to dispatch this action when the form is submitted, passing along the data from the form. To do this we'll use the trick we learnt in week 9 for passing data out of a component: pass in a function that accepts an argument and then call that function from within the component.

We can pass in a function that has access to `dispatch` by using `mapDispatchToProps`:

```
// import in the action
import { step } from "../../data/actions";

// as before
const mapStateToProps = state => { /* ...etc. */ };

// pass in a function that accepts a value
// pass the value along to the action creator
const mapDispatchToProps = dispatch => {
  return {
    handleSave: value => dispatch(step(value)),
  };
};

// make sure you pass in mapDispatchToProps as the second argument
export default connect(mapStateToProps, mapDispatchToProps)(Settings);
```

[View code on GitHub]

Now we just need to call this function from inside the `<Settings>` component when the form is submitted:

```
handleSubmit(e) {
  e.preventDefault();

  // call the passed in function
  // pass it the current input value
  this.props.handleSave(this.state.step);
}
```

[View code on GitHub]

If we look in the Redux developer tools we should now see an action being dispatched! But the state isn't changing. The last thing we need to do it update the reducers to do something with this action:

```
const step = (state, { amount }) => ({ ...state, step: amount });

const reducer = (state, action) => {
  switch (action.type) {
    // ...etc.
    case "step": return step(state, action);
    default: return state;
  }
};
```

[View code on GitHub]

Now if we look in the Redux developer tools you should see that the state is being updated.

Finally, to actually update the counter by the right amount we need to update the `change` reducer:

```
const change = (state, action) => {
  return {
    ...state,
    count: state.count + (action.amount * state.step),
  };
};
```

[View code on GitHub]

5.3 Programmatic Navigation

It would be nice if our app could navigate to the homepage once we save the settings. To do this we can use ReactRouter's **history** functionality.

First, create a new file `src/history.js`:

```
import { createBrowserHistory } from "history";
export default createBrowserHistory();
```

[View code on GitHub]

Then add a `history` prop to your `<Router>`:

```
// import your history file
import history from "../history";

// use Router instead of BrowserRouter
import { Router } from "react-router-dom";
```

```
// pass history in as a prop to the Router
<Router history={ history }>
```

Now we can import this file and use it elsewhere to change the URL:

```
import history from "../../history";
```

```
const mapDispatchToProps = dispatch => {
  return {
    handleSave: value => {
      // dispatch the action
      dispatch(step(value));

      // go to the homepage
      history.push("/");
    },
  };
};
```

[View code on GitHub]

5.4 Process

As with React components there's an order you can do things in which will mean that nothing breaks as you go along:

1. Create the JSX template
2. If your component needs to read values from state:

    (a) If you haven't got one already, create a container component using `connect`
    (b) Use `mapStateToProps` to pass in any of the state that's needed
    (c) Update the initial state to see if it changes the UI

3. If your component needs to update state:
    (a) If you haven't got one already, create a container component using `connect`
    (b) Create the necessary action creators
    (c) Use `mapDispatchToProps` to pass in any event handlers, using your action creators
    (d) Update your reducers to handle the new action type

Check that nothing has broken at each stage by viewing it in the browser.

5.5 Additional Resources

- React Redux: `mapDispatchToProps`
- `mapDispatchToProps`
- ReactRouter History

Chapter 6

API Actions

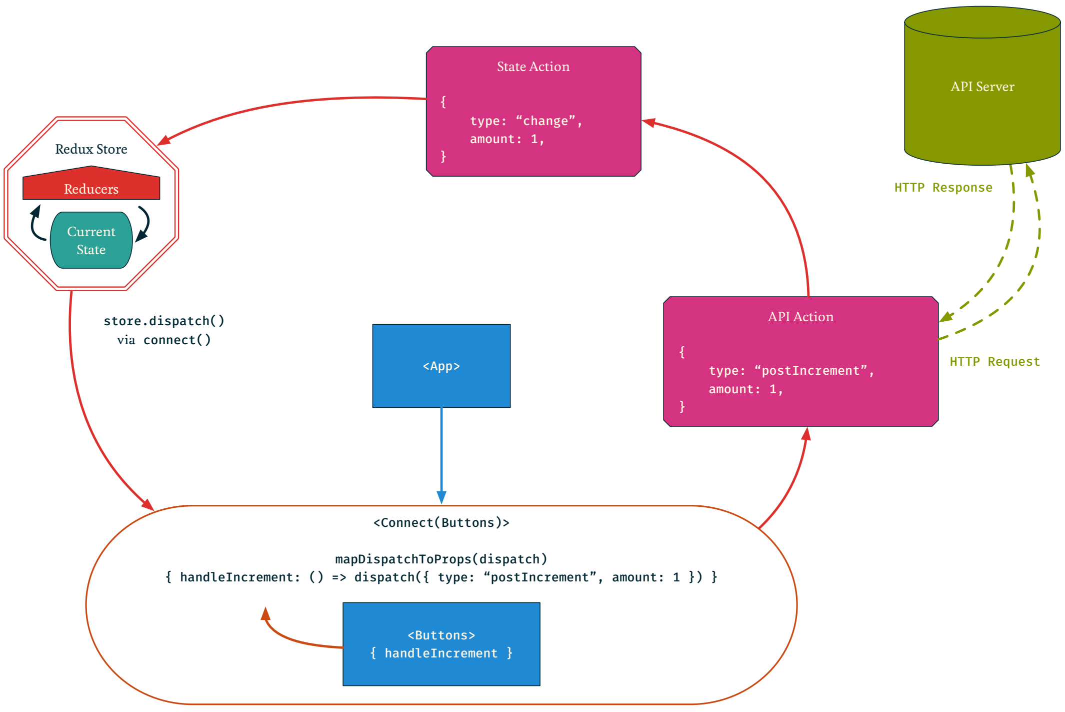
Persisting our state using `localStorage` is all well and good for a basic app, but it's no good for something like Trello, where the data needs to be viewable by multiple people on different machines. We'll want to use an API to store our data.

But where are we going to put the API requests?

- Components shouldn't handle data: it limits their re-usability
- Containers are just for hooking up React to Redux: they're still technically part of the "view" layer, so we shouldn't put data stuff there either
- Reducers should be pure and return immediately: they can't return unpredictable asynchronous results

This only leaves us with one place where we could put an asynchronous request: our action creators.

The idea is that we'll have two different types of action creators: **state action creators** and **API action creators**. If we just want to update the state without persisting anything via the API then we'll just dispatch a state action as before. However, if we need to make an API call, we'll dispatch an API action creator, which will use Axios to send an API request. Once we get back an API response, we use the data we get back and dispatch a state action to update what the user is seeing.



Simples

6.1 Thunks

However, currently we have no clean way of using `dispatch` inside our actions. That’s where `thunks` come in.

The `redux-thunk` **middleware** adds the ability to dispatch from our actions, so we can put asynchronous code inside an action and then dispatch a new action once we get back a response.

Although we dispatch thunk actions, they don’t have a `type` property, and so aren’t handled directly by our reducers.

6.1.1 Setup

Setting up Thunks while keeping Redux Dev Tools working gets a little complicated, but hopefully you can follow along.

First, install `redux-thunk`:

```
npm install redux-thunk
```

Next, update your `redux` import in `index.js`:

```
import { createStore, applyMiddleware, compose } from "redux";
```

Then, import `thunk`:

```
import thunk from "redux-thunk";
```

Finally, update the call to `createStore`:

```
const composeEnhancers =
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(
    reducer,
    initial,
    composeEnhancers(applyMiddleware(thunk))
);
```

The last bit just mixes together the Redux Dev Tools and the `thunk` package so that they can all be passed in as one argument to `createStore`.

You can remove the `redux-localstorage` import from `index.js`, as you'll be using an API to keep track of things.

6.2 Axios

We'll also need to get Axios setup for making the API request.

First install `axios`:

```
npm install axios
```

Axios gives us the ability to setup a default configuration including a base URL and URL parameters. We can use this to avoid needing to repeat this information with every request.

Put something like the following (with the correct base URL and key) in `src/axios.js`:

```
// import the axios library
import axios from "axios";

// return a new version of axios with useful settings applied
export default axios.create({
    baseURL: "http://username.restful.training/api/counters",
    params: {
        key: "b1a4b1a4b1a4-api-key-b1a4b1a4ab1a4",
    },
    headers: {
        Accept: "application/json",
    },
});
```

We can then import *this* file (instead of just `axios`) to keep our request code nice and clean.

6.3 Loading

It's fairly common to want to load some information from an API when you first display a component. Let's update the counter to get the current values from the API.

We'll need to keep track of whether the app has loaded the API request, so let's add an extra bit of state:

```
const initial = {
    // set it to false initially
    loaded: false,
    // ...etc.
};
```

We're going to create a `<Loading>` component which we can show while the user is waiting for a response:

```
import React, { Component } from "react";

class Loading extends Component {
  render() {
    const { children, loaded } = this.props;

    return loaded ? children : (
      <div className="progress">
        <div
          className="progress-bar progress-bar-striped active"
          style={{ width: "100%" }}
        />
      </div>
    );
  }
}

export default Loading;
```

[View code on GitHub]

We've created a class based component (you'll see why later) that accepts a `children` and a `loaded` prop. We can now wrap the two parts of our app in this component and if `loaded` is `true` they will display, otherwise it will show a loading animation.

We'll need to wrap the main components:

```
<Loading>
    <p><Link to="/settings">Settings</Link></p>
    <Value />
    <Buttons/>
</Loading>
```

And also the settings component (in case the user loads `/settings` initially):

```
<Loading>
    <Settings />
</Loading>
```

Let's wrap the `<Loading>` component with a container component so that it can access the `loaded` property from state:

```
import { connect } from "react-redux";
import Loading from "./Loading";

const mapStateToProps = ({ loaded }) => ({
    loaded,
});

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

Now, it looks like our app is always loading. We'll need to do an API request and, when it's successful, let the `<Loading>` component know about it. We can use the `componentDidMount` method to trigger an action when the component first shows:

```
componentDidMount() {
    if (!this.props.loaded) {
        this.props.handleLoad();
    }
}
```

First, check to see if `loaded` is `true`, because if it is there's no point triggering an action. If not then call the `handleLoad` prop.

We'll need to add the `handleLoad` prop next:

```
// ...etc.

// we'll need to dispatch and action, so we need mapDispatchToProps
// just put a console.log() in for now
const mapDispatchToProps = (dispatch) => ({
    handleLoad: () => console.log("loaded"),
});

// make sure to add mapDispatchToProps into connect
export default connect(mapStateToProps, mapDispatchToProps)(Value);
```

We're going to dispatch our first **API action creator**. It's going to look something like this:

```
// import in the pre-configured axios object
import axios from "../../axios";

// create a function that returns another function
// the second function takes an argument that
// represents dispatch
export const getCounter = () => dispatch => {
    // now use axios to make an API request
    axios.get("/").then(({ data }) => {
        console.log(data);
    });
};
```

We're using thunks for the first time. A thunk lets us return a function from an action creator. The function we return should accept `dispatch` as it's first argument. This function is then called when we dispatch the API action. This lets us run whatever code we like in the action creator and then dispatch another action at a later point.

We're going to make a request to the API, wait for it to return the data, and then once it does dispatch a regular **state action**, which will trigger the reducers in the Redux store.

Let's update the `mapDispatchToProps` to call this API action creator:

```
import { getCounter } from "../../data/actions/api";

// ...etc.

const mapDispatchToProps = (dispatch) => ({
    handleLoad: () => dispatch(getCounter()),
});
```

Now if we look in the "Network" tab of Developer Tools, we should see a `GET` request when the `<Loading>` component loads. Now we just need to update the API action so that it updates the app state.

First we'll need to create a state action:

```
export const loaded = ({ count, step }) => {
    return {
        type: "loaded",
        count,
        step,
    };
};
```

And then call this from the API action creator:

```
axios.get("/").then(({ data }) => {
    // pass the data property of the API data
    dispatch(loaded(data.data));
});
```

Finally, we need to add a reducer to handle the new `loaded` action:

```
// destructure the action
const loaded = (state, { count, step }) => ({
    ...state,
    // set loaded to true
    loaded: true,
    count,
    step,
});

const reducer = (state, action) => {
    switch (action.type) {
        case "loaded": return loaded(state, action);
        // ...etc.
    }
};
```

6.4 Sending Data

It's also common to need to send data to the API. Let's update the Settings form so it submits the new `step` value to the API. First, let's create the API action:

```
// import in existing state action
import { step } from "./state";

// accept a value argument
export const putStep = value => dispatch => {
    // make a PUT request
    axios.put("/", {
        // pass along the data to the API
        // can pass in a regular object literal
        // axios will convert into JSON
        step: value
    }).then(({ data }) => {
        // get the step result off the data
        // pass it along to the existing step action
        dispatch(step(data.data.step));
    });
};
```

We can call the existing `step` action creator once the API request comes back.

Now we just need to update the `<Settings>` container component to use the API action instead:

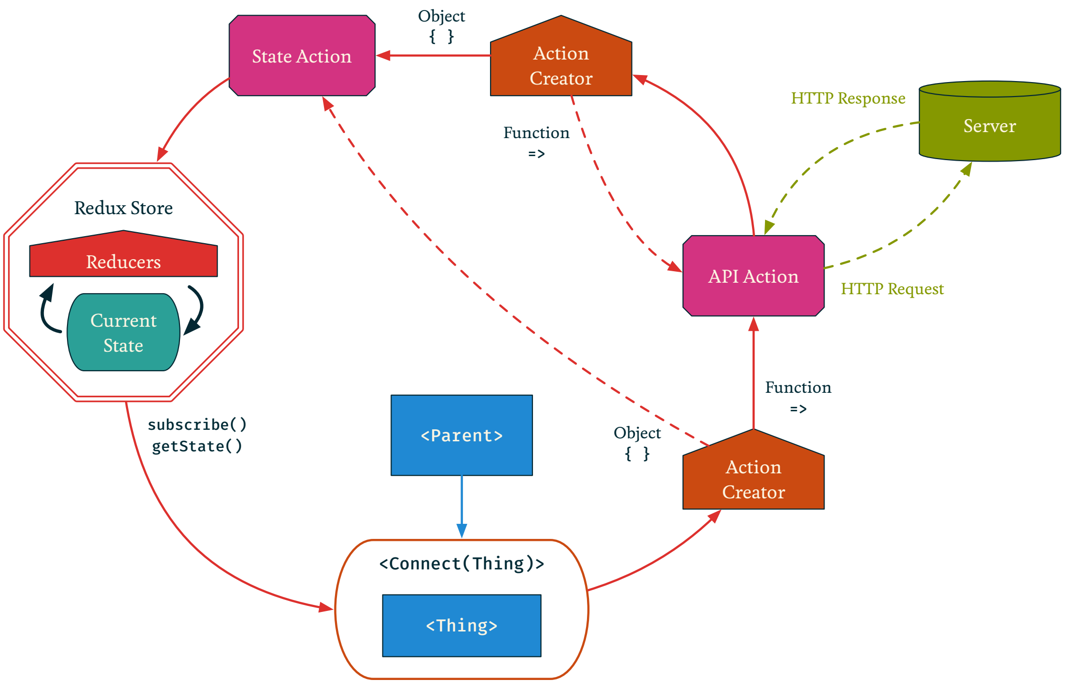
```
import { putStep } from "../../data/actions/api";

// ...etc.

const mapDispatchToProps = dispatch => {
    return {
        handleSave: value => {
            dispatch(putStep(value));
            history.push("/");
        },
    };
};
```

∞

We haven't got the full functionality of iCounter working with the API, but we have demonstrated all of the key concepts with making API requests.

6.5 Summary

It might look complicated, but this architecture scales well from really simple apps to complex ones like Trello.



The whole shebang

- If we need to do something with the API we'll need to create an **API action creator** which will make an API request
- Once we get back an API response, we pass that along to a **state action creator** which will update the state so that the user interface updates with the change
- As before we use `mapDispatchToProps` to let our components `dispatch` actions
- We use the `componentDidMount` method if we need to `dispatch` when a component first loads

Throughout this process you should use Redux Developer Tools and the console and Network tabs to keep track of what's going on.

6.5.1 Tips

It's a good idea to name your API action creators things like **get**Article, **post**Article, **delete**Article - using the HTTP methods. For your state actions use other names, like `addArticle`, `updateArticle`, `removeArticle`.

6.6 Additional Resources

- Redux Thunk
- Where to fetch data
- React Lifecycle Events

Chapter 7

Building an App From Scratch

Without requirements or design, programming is the art of adding bugs to an empty text file

- Louis Srygley

Before starting to build an app you should go through various design stages:

- Functional Spec (often from User Stories)
- Wireframes
- API Routes
- Database Structure
- Redux Actions

7.1 Functional Spec

First we need to decide what functionality our app will have. On bigger projects this will usually be driven by **User Stories**.

When coming up with a functional spec it's important to focus on a **Minimum Viable Product**: what is the most minimal functionality the app could have while still being useful? By releasing early and often you can get useful feedback from users.

7.2 Wireframes

Next we'll need to create wireframes for our app. Make sure that everything from the functional spec is covered on the wireframes. Remember, wireframes are not concerned with styling, just about showing the functionality visually.

7.3 API Routes

It's useful to plan your API routes in advance. For each route you'll need to list:

- The resource URL
- The HTTP method (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`)
- The controller and method it should point to

7.4 Database Structure

It's also helpful to plan your database structure in advance. List the tables that you'll need, the columns for each table (name and type), and how the tables relate to one another (make sure you set up the foreign keys correctly).

7.5 Redux Actions

Finally we'll need to think about the Redux actions that our front-end app will need. For each action we'll need to think about:

- Is it an API or State action?
- What to call it (its `type` property)
- What data does the action need (don't forget about `id`)

Remember, for every API action you'll probably need a State action that takes the API response and updates the state.

Glossary

- **Action Creator**: A function that generates a Redux action - these are not a necessary part of Redux, they just make actions more reusable
- **Action**: An object literal which tells our reducers how to transform the state
- **API action creator**: An action creator that makes an API request and then dispatches a regular **state action** when the API response comes back
- **Business Logic**: The application logic specific to how data is transformed/calculated
- **Container Component**: A file that uses **connect** to wrap a standard component, allowing us to pass in props that can access **store**
- **Dispatch**: We dispatch an action to the store in order to change the state
- **Payload**: The additional data sent with an action
- **Provider**: React redux wraps our app with a `<Provider>` component, which allows us to access the store using **connect**
- **Reducer**: Transforms the state based on the action that was dispatched
- **Store**: Keeps track of our state for us
- **Subscribe**: We subscribe to the store in order to track changes to the state
- **Thunk**: A redux **middleware** that allows us to dispatch from our actions

Colophon

Created using TEX

Fonts

- Feijoa by Klim Type Foundry
- Avenir Next by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- Fira Mono by Carrois Apostrophe

Written by Mark Wales

∞

August 30, 2019