10

Building More UIs
with React

"[Debugging] is damnably troublesome work, and plagues me."

- Ada Lovelace, 1840

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

One-Way Data Flow

1.1 Lifting State

What if we've got two components that need to know about each other?

Say, for example, we have two buttons but we want it so that only one of them can be selected at once:

Clicked on first button

Clicked on second button

We can easily create a self-aware button that knows when *it's* been clicked:

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

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

    // keep track of whether the button is selected
    this.state = { selected: false };

    // bind the click even handler
    this.handleClick = this.handleClick.bind(this);
```

```
  }

  // when clicked
  handleClick() {
    // invert the value of selected
    this.setState({ selected: !this.state.selected });
  }

  render() {
    // destructure props
    let { name } = this.props;

    // destructure state
    let { selected } = this.state;

    // render the button
    return (
      <button
        className={ `btn btn-${selected ? "danger" : "primary"}` }
        onClick={ this.handleClick }
      >
        { name }
      </button>
    );
  }
}

export default Button;
```

[View code on GitHub]

But there's no way for the first button to know anything about the state of the second button and vice-versa. Remember: we can only pass information down from one component to its sub-components, we *can't* pass data back up.

This is where the concept of **lifting state** becomes important. If we store the state of both buttons in a shared parent component, then the parent component can keep track of which button has been selected and then use props to pass this information back down to each button.

First let's create the component. We'll use a number to keep track of which `<Button>` we want highlighted:

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

// import the currently self-aware Button component
import Button from "./Button";

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

    // keep track of which button is selected
    // we'll just use a number for now
    this.state = { selected: 1 };
  }

  render() {
    return (
      <>
        <Button name="First" />
        <Button name="Second" />
      </>
    );
  }
}

export default Buttons;
```

[View code on GitHub]

Next we'll update the `<Buttons>` component to pass down the value of `selected` to `<Button>` using a prop:

```
import React, { Component } from "react";
import Button from "./Button";

class Buttons extends Component {
  constructor(props) {
    super(props);
    this.state = { selected: 1 };
  }

  render() {
    return (
```

```
      <>
        { /* if selected is 1, this will be true, otherwise false */ }
        <Button name="First" selected={ this.state.selected === 1 } />

        { /* if selected is 2, this will be true, otherwise false */ }
        <Button name="Second" selected={ this.state.selected === 2 } />
      </>
    );
  }
}

export default Buttons;
```

[View code on GitHub]

We'll need to update our `<Button>` so it works out the class name based on the `selected` *prop* (as opposed to using its own state):

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

class Button extends Component {
  // ...etc.

  render() {
    let { name, selected } = this.props;

    // using the selected prop rather than state
    return (
      <button
        className={ `btn btn-${selected ? "danger" : "primary"}` }
        onClick={ this.handleClick }
      >
        { name }
      </button>
    );
  }
}

export default Button;
```

[View code on GitHub]

Now, when we load the app, the first button should be selected, but clicking won't

make any difference. We could check it works for the second button by temporarily changing the initial state in `<Buttons>`.

The `<Button>` JSX isn't using state anymore, so the `onClick` handler won't do anything. And we can't set an event handler on `<Button>` as it isn't an HTML element.

But remember that functions in JavaScript can be passed around just like any other value, so we *can* pass in an event handler from the parent component:

```
import React, { Component } from "react";
import Button from "./Button";

class Buttons extends Component {
  // ...etc.

  render() {
    return (
      <>
        { /* when this button is clicked, set selected to 1 */ }
        <Button
          name="First"
          selected={ this.state.selected === 1 }
          handleClick={ () => this.setState({ selected: 1 }) }
        />

        { /* when this button is clicked, set selected to 2 */ }
        <Button
          name="Second"
          selected={ this.state.selected === 2 }
          handleClick={ () => this.setState({ selected: 2 }) }
        />
      </>
    );
  }
}

export default Buttons;
```

[View code on GitHub]

We pass through an anonymous function that sets `<Buttons>`'s state with the correct value.

Each instance of the `<Button>` component gets given a *different* event handler: the first button gets one that sets the `selected` value to 1 and the second button one that sets it to 2.

Finally we need to accept the `handleClick` prop inside our button and set it as the `onClick` event handler. And, as we're not using state in the component anymore, we can also refactor it into a stateless component while we're at it:

```
import React from "react";

const Button = ({ name, selected, handleClick }) => (
  <button
    className={ `btn btn-${selected ? "danger" : "primary"}` }
    onClick={ handleClick }
  >
    { name }
  </button>
);

export default Button;
```

[View code on GitHub]

By passing a function into the component we've avoided the need for two-way data flow. The `<Button>` component still doesn't need to know anything about the component that it is used in, as long as the parent component passes in a `handleClick` function everything will work; but the function that gets passed in could do *anything*.

1.2 Passing Data Up

We can use a similar process to get data out of a component without needing to break one-way data flow.

Say that you have a `<Counter>` component:

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

class Counter extends Component {
  constructor(props) {
```

```
    super(props);

    this.state = {
      count: 0,
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    let { count } = this.state;

    return (
      <>
        <p onClick={ this.handleClick }>{ count }</p>
      </>
    );
  }
}

export default Counter;
```

[View code on GitHub]

How could we add a button which, when clicked, passes the current value of the counter up to the parent component?

We can't pass data up with props, but we can pass arguments to functions:

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

class Counter extends Component {
  // ...etc.

  // make sure you bind in constructor
  handleButtonClick() {
    // destructure props to get the handleSubmit function
    // this must be passed in from the parent
    let { handleSubmit } = this.props;
```

```
    // now, call the passed in handleSubmit function
    // and pass it the current value of count
    handleSubmit(this.state.count);
  }

  render() {
    let { count } = this.state;

    return (
      <>
        <p onClick={ this.handleClick }>{ count }</p>
        { /* call the local handleButtonClick method */ }
        <button onClick={ this.handleButtonClick }>Submit</button>
      </>
    );
  }
}

export default Counter;
```

[View code on GitHub]

In the parent component we can accept this argument as a parameter to the function we pass in:

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

// import the Counter component
import Counter from "./Counter";

class Parent extends Component {
  // because we've passed it in as the handleSubmit prop
  // this is the function that gets called inside Counter
  // so if we accept the first argument
  // we have access to the value of the counter
  handleCounterSubmit(value) {
    console.log(value);
  }

  render() {
    // pass in a the handleCounterSubmit method as
    // the handleSubmit prop
```

```
    return <Counter handleSubmit={ this.handleCounterSubmit } />;
  }
}

export default Parent;
```

[View code on GitHub]

Again, we've still got one-way data flow: the `<Counter>` component doesn't need to know *anything* about the `<Parent>` component. As long as the parent component passes in a `handleSubmit` prop, which is a function that accepts a single argument, everything will work.

Also note that the parent component can't *ask* for the data from `<Counter>`, it can only get it when some event happens inside `<Counter>`. Really we're not getting data *out* of the sub-component, we're passing *in* a function from `<Parent>`.

1.3 Additional Resources

- React: Lifting State Up
- Building Reusable Components Using React
- Composition vs Inheritance
- Inversion of Control

Chapter 2

Lifecycle Methods

The "lifecycle methods" are the special named methods of class based components that get called by React. We've already been using `constructor` and `render`, but there are some other useful ones to cover.

2.1 `render`

The `render` method is called by React when it needs to re-render the component. It should return valid JSX. The only code that should go in `render` should be related to generating the JSX. It should not make changes to the DOM or have any other side-effects, as these can create all sorts of bugs.

2.2 `constructor`

The `constructor` method is not a React-specific method, it's part of JavaScript, but it does have a very specific use for React components.

It should only be used for:

- Setting up the initial state
- Binding methods

If you don't need to do either of these then you don't need to write a `constructor` method at all and it will fallback to the parent's (`Component`) `constructor`.

2.3 componentDidMount

This method is called for us by React when the component first renders, so any code we put in it will run when a component is rendered on screen.[1] This is subtly different from the `constructor`.

`componentDidMount` is a safe place to perform DOM manipulation and other side-effects.

We could use this, for example, to setup a timer so that our component hides itself after 5 seconds:

```
componentDidMount() {
    setTimeout(() => {
        // assumes logic in JSX to hide component
        this.setState({ hide: true });
    }, 5000);
}
```

Sometimes it's also necessary to add event handlers to, say, the `window` object. We can set this up in `componentDidMount` too:

```
componentDidMount() {
    // a drag-and-drop component might need to know when the mouse
    // is no longer held down, no matter what component it's over
    window.addEventListener("mouseup", this.handleStopDragging);
}
```

If you do setup event handlers in `componentDidMount`, make sure you remove them using `componentWillUnmount`.

2.4 componentWillUnmount

React does all sorts of clever stuff for us, including adding and removing all necessary event listeners to DOM elements that it creates. However, if we add event listeners ourselves (as in the `componentDidMount` example), we need to make sure we remove them when the component is no longer needed:

[1] If a component is removed and then re-added, the `componentDidMount` method will run again.

```
componentWillUnmount() {
    // we need to remove any event listeners
    // that React hasn't setup itself
    window.removeEventListener("mouseup", this.handleStopDragging);
}
```

If we forget to do this, over the life of our app multiple copies of the same event listener will build up, using memory and CPU unnecessarily (and potentially leading to all sorts of weird bugs).

2.5 `componentDidUpdate`

Sometimes we want to run a bit of code every time the component updates.

Again, `componentDidUpdate` is a safe place to perform DOM manipulation and other side-effects.

For example, say we wanted to update the `<title>` tag of the web-page every time some value in state changes:

```
componentDidUpdate() {
    let { name } = this.state;
    document.title = `Hello ${name}`;
}
```

∞

If you're using any of the lifecycle methods to perform DOM manipulation, **make sure there isn't a React way to achieve the same thing first**. As a general rule of thumb, if the DOM manipulation is on DOM elements that are part of the component (or its children), you should be able to use React-specific methods.

2.6 Additional Resources

- React: The Component Lifecycle
- React: `render`
- React: `constructor`

- React: `componentDidMount`
- React: `componentDidUpdate`
- React: `componentWillUnmount`

Chapter 3

AJAX

If a React app needs to work with data that's stored on a server somewhere, it will need to make API calls.

So far the only way we know to get data in the browser is to navigate to a new page and the only way to submit data is to create a form with a `POST` method. Both of these involve refreshing the page, which is no good for an app that runs purely in JavaScript.

"AJAX" (**Asynchronous JavaScript and XML**) is an outdated term for doing HTTP requests using JavaScript.

The name comes from the pre-JSON era, when XML was the most common way to transfer data:[1]

```
<employees>
    <employee>
        <name>Jenny</name>
        <email>jenny1987@gmail.com</email>
    </employee>
    <employee>
        <name>Rahul</name>
        <email>rahul12@gmail.com</email>
    </employee>
    <employee>
        <name>Mia</name>
        <email>mia87@gmail.com</email>
```

[1] XML is still used in some legacy systems

```
    </employee>
</employees>
```

As you can see, XML is verbose and repetitive. Nowadays JSON is used pretty much everywhere. So it should really be called "AJAJ", but that doesn't sound as cool.

In any case, people still say "AJAX" when they're talking about JavaScript making HTTP requests.

3.1 Axios

There are various ways to make an AJAX request from the browser.

The original method, `XMLHttpRequest`, is horrifying to use, as it pre-dates the time when people realised that JavaScript didn't have to be unpleasant to work with.

The more modern `fetch` API is much nicer to work with and is now supported in most browsers. However, it's still a bit ungainly to work with.

We'll be using Axios for our HTTP requests. It's a handy library that makes doing HTTP requests with JavaScript really easy.[2]

You can install it with:

```
npm install axios
```

Axios provides us with methods for the various HTTP methods:

```
import axios from "axios";

// make a GET request
axios.get("/articles");

// make a POST request, with the given data
axios.post("/articles", {
    title: "Hello",
```

[2] And, as an added bonus, it works in Node too.

```
    article: "Blah blah blah",
    tags: ["fish", "cat", "spoon"],
});

// make a PUT request, with the given data
axios.put("/articles/5", {
    title: "Hello Again",
    article: "Blah blah blah!",
    tags: ["fish", "cat"],
});

// make a DELETE request
axios.delete("/articles/4");
```

3.1.1 Config

One of Axios' best features is it's easy to configure the parts of your HTTP request that are the same every time.

For example, if we're using a RESTful API, our base URL, the `Accept` header, and the `Authorization` header are going to be the same for every request, so we can setup a version of Axios that has those already setup:

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

// create a version of axios with useful defaults
export default axios.create({
    baseURL: "https://restful.training/api/",
    headers: {
        // make sure we get JSON back
        Accept: "application/json"

        // use your own key
        Authorization: "Bearer
        ↪  4fdea06a65ba1491091c0db709faf0cce944067a"
    },
});
```

Then we can import *that* file and use it as follows:

```
// import *local* version of axios
import axios from "./axios";

// automatically handle base URL and key
axios.get("/articles");
```

3.2 Additional Resources

- Axios
- How To Make HTTP Requests Like a Pro with Axios

Chapter 4

Asynchronous Programming

When we make a request to the API we don't get a response straight away. Even on a fast connection it can take 100ms or more to get a response. That might not sound like a long time, but for a computer that's ages.

So, JavaScript doesn't just stop doing everything until you get a response back: it keeps on doing whatever else needs to be done until it gets a response.

When we don't get back a response immediately, we need to deal with the response **asynchronously**: in other words, the code that runs when the response comes back could run at *any time* in the future (or never). We've dealt with a similar concept when we wrote event handlers.

4.1 Promises

So, when we make a request with Axios, we can't be given back the response to store in a variable, as the response doesn't exist yet. What we get back instead is a **Promise**.

A Promise is an object with a `.then()` method. We can pass the `.then()` method a function, which will run once the response comes back. With Axios, the function will be passed the response as the first parameter.

```
// make the request
let promise = axios.get("/articles");

// setup the handler for when the response is successful
promise.then(response => {
```

```
    console.log(response);
});
```

Rather than using an intermediary variable, generally we use chaining with Promises:

```
// use destructuring to get the data property
axios.get("/articles").then(({ data }) => {
    console.log(data);
});
```

Be aware that **there's no guarantee the promises will resolve in the order you wrote them**. For example, `GET` requests tend to be quicker than other types of request as they generally involve less server activity.

We can use https://httpbin.org/delay/<delay> to play around with this:

```
// will take 5 seconds to return a response
axios.get("https://httpbin.org/delay/5").then(() => {
    // will show after (at least) 5 seconds
    console.log("Five");
});

// will take 1 second to return a response
axios.get("https://httpbin.org/delay/1").then(() => {
    // will show after (at least) 1 second
    console.log("One");
});
```

4.1.1 Errors

Sometimes things will go wrong: you might have sent invalid data, the API server might be down, or any number of other issues.

The `.then()` method of a Promise accepts a second argument which will be called if something goes wrong:

```
axios.get("/articles").then(response => {
    console.log("Everything has worked");
}, error => {
```

```
    console.log(error); // logs an error message
    console.log(error.response); // the response object
    console.log("Something has gone wrong");
});
```

Alternatively you can use the `.catch()` method:

```
axios.get("/articles").then(response => {
    console.log("Everything has worked");
}).catch(error => {
    console.log("Something has gone wrong", error.response);
});
```

You could use the above methods to handle form validation errors and the like.

4.2 API Requests from Components

We can put API calls in our component methods, we just need to make sure that we use `this.setState()` *inside* the `.then()` function:

```
handleSubmit() {
    // get the values of some controlled components
    let { title, article } = this.state;

    // post data to an API
    axios.post("/api/article", {
        title: title,
        article: article,
    }).then(() => {
        // once the server responds successfully, clear the inputs
        this.setState({ title: "", article: "" });
    });
}
```

If we need to fetch data from the API to show in our component, then we'll need to use `componentDidMount`:

```
import React, { Component } from "react";
import axios from "./axios";
```

```
class Articles extends Component {
  constructor(props) {
    super(props);

    this.state = {
      loaded: false,
      articles: [],
    };
  }

  // runs when the component first renders
  componentDidMount() {
    // make the GET request
    axios.get("/articles").then(({ data }) => {
      // once the data has come back update the component state
      this.setState({
        loaded: true,
        articles: data.data,
      });
    });
  }

  render() {
    let { articles, loaded } = this.state;

    return !loaded ? <p>Loading...</p> : (
      <>
        <h2>Articles</h2>
        <ul className="list-group">
          { articles.map(article => (
            <li key={ article.id } className="list-group-item">
                { article.title }
            </li>
          )) }
        </ul>
      </>
    );
  }
}

export default Articles;
```

[View code on GitHub]

If it's not doing what you expect, you can use the "Network" tab in Developer Tools to see if your API requests are working or not.

Not Ideal

If we make an API request inside our component, then it means that the component can *only* be used for displaying that specific data.

However, the whole purpose of React components is that we can reuse the same bits of UI with different data. So this is far from ideal.

When we look at Redux next week we'll see a much cleaner way to do API calls, which does not tie a component to a specific API call.

4.3 Additional Resources

- MDN: Promises
- Promises: A Technical Look
- MDN: `async` functions
- Writing Asynchronous Tasks In Modern JavaScript

Chapter 5

Hooks

Hooks are a newer way of writing stateful components in a more functional style, without needing to use a `class`. This avoids issues with `this` and, done properly, can lead to more composable code.

Let's take a look at a basic `class` based component again:

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

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

    // setup initial state
    this.state = {
      counter: 0,
    };

    this.handleClick = this.handleClick.bind(this);
  }

  // click handler method
  handleClick() {
    // destructure state to get value of counter
    let { counter } = this.state;
    // use setState to update counter
    this.setState({ counter: counter + 1 });
  }

  render() {
    // destructure state to get value of counter
```

```
    let { counter } = this.state;
    // display counter and setup click handler
    return <p onClick={ this.handleClick }>{ counter }</p>;
  }
}

export default Counter;
```

[View code on GitHub]

And here it is using hooks:

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

const Counter = () => {
  // sets up the initial value of counter
  // and gives back the current value and a function to update counter
  const [counter, setCounter] = useState(0);

  // the event handler
  // uses the provided function to set the value of counter
  const updateCounter = () => setCounter(counter + 1);

  // display counter and setup click handler
  return <p onClick={ updateCounter }>{ counter }</p>;
}

export default Counter;
```

[View code on GitHub]

The hooks version is less than half the length, but it does exactly the same thing.

Before we delve into exactly how this works, we'll need to look at destructuring arrays.

5.1 Destructuring Assignment: Arrays

So far we've only used destructuring to get properties from an object:

```
let person = { firstName: "Ben", lastName: "Folds", favouriteNo: 5 };
```

```
// get the firstName property and store it in a firstName variable
let { firstName } = person;
console.log(firstName); // "Ben"
```

We can also destructure an array. However, we don't have property names to work with and we're not allowed (and wouldn't want) variables that are just numbers. So destructuring an array is based purely on the *order* of the items in the array:

```
let values = [1, 2, 3, 4];

// get first and second value
let [first, second] = values;
console.log(second); // 2
```

In the above example we could have called the variables `first` and `second` anything we liked: it's purely the order that's important. You'll also notice that we've ignored the third and fourth values in the array.

5.2 `useState`

Now we understand destructuring arrays, we can start to understand `useState`.

`useState` sets up state for a *single* value. It does three things for us:

- Setting up the initial value
- Giving us the current value
- Giving us a function to update the value

In the counter example, we want to keep track of a value that we're calling `counter`. It should have an initial value of `0`, so we call `useState(0)` passing in the initial value.

The `useState` function returns an array containing two values: the current value and a function to update that value. We can then destructure this array to get the value and update function back, using whatever names are most appropriate (usually using the `x` and `setX` convention):[1]

[1] If it returned an object it would be messier to name them whatever we like

```
// useState returns an array containing two values
// the current value
// a function to update the value
const [counter, setCounter] = useState(0);
```

As mentioned above, `useState` can only be used for setting/updating a `single` value. You will need more than one call to `useState` if you need more than one value in state:

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

const Scorer = () => {
  // sets up player 1 score value
  const [player1, setPlayer1] = useState(0);
  // sets up player 2 score value
  const [player2, setPlayer2] = useState(0);

  // event handlers for both
  const player1Scores = () => setPlayer1(player1 + 1);
  const player2Scores = () => setPlayer2(player2 + 1);

  // display counter and setup click handler
  return (
    <>
      <p onClick={ player1Scores }>Player 1: { player1 }</p>
      <p onClick={ player2Scores }>Player 2: { player2 }</p>
    </>
  );
}

export default Scorer;
```

[View code on GitHub]

As you can imagine, this gets quite messy if you have lots of values in state. In that case it's best to use `useReducer`.

Tuples

You might notice that `useState` returns an array containing mixed data-types: any type for the first value and a function for the second. We've previously seen that mixing your types in an array is a **bad idea**, as generally you want to do the same thing to every item in an array. The exception to this is if you're using an array as a "tuple".

Many programming languages, particularly functional ones, have a data type called a tuple. A tuple is a data-structure that contains two values that are somehow related. It might look something like this:

```
(40.242531, -109.013390) -- a GPS coordinate
("Marte", 99) -- a name and age
```

It's common, although not necessary, for the two parts of a tuple to be *different* data types: it just ties two values together so they don't get separated.

JavaScript doesn't have native support for tuples, so the closest approximation is to use an array with two values.

5.3 `useReducer`

5.3.1 Reducers

Reducers are like the `reduce` method of arrays, except that instead of accumulating a value by going through each item of an array, they accumulate the current value of state over time. Simples!

A reducer is a **pure** function which gets passed the current value of the state plus some additional data, it then has to return a new version of the state based on those two bits of information. Reducers should be pure functions and **must always return a valid copy of the state**.

The simplest possible reducer is one that takes the current state and returns it:

```
// the reducer
const reducer = state => {
```

```
  return state;
};

// our state
const initial = {
  player1: 0,
  player2: 0,
};

// run the reducer, passing in state
let newState = reducer(initial);
console.log(newState); // same as initial
```

[View code on GitHub]

This is not particularly useful, as it means the state can never change!

For our reducer to be useful we need to also pass along some additional information, in React[2] this is normally an "action".

An action is just an object literal with various properties. If we give it a `type` property then we can change the state in different ways depending on the value of this specific property:

```
const reducer = (state, action) => {
  switch (action.type) {
    // if the type property is "PLAYER_1_SCORES"
    // add one to the player1 property of state
    case "PLAYER_1_SCORES": return {
      ...state,
      player1: state.player1 + 1,
    };
    default: return state;
  }
};

// run the reducer, passing in state and action
let action = { type: "PLAYER_1_SCORES" };
let newState = reducer(initial, action);
console.log(newState); // { player1: 1, player2: 0 }
```

[2] And Redux

[View code on GitHub]

In the above reducer if the action has a `type` property of `PLAYER_1_SCORES`[3] we add `1` to the `player1` property of state. In all other cases (`default`) it will just return state unchanged - remember, **we always need to return a valid copy of the state**.

We can now easily add a `PLAYER_2_SCORES` action:

```
case: "PLAYER_2_SCORES": return {
    ...state,
    player2: state.player2 + 1,
};
```

It's sometimes tidier to pull the `case`s out into their own functions:

```
// reducer functions
const player1Scores = state => {
  return {
    ...state,
    player1: state.player1 + 1,
  };
};

// ...other functions

const reducer = (state, action) => {
  switch (action.type) {
    // call the function, passing state
    case "PLAYER_1_SCORES": return player1Scores(state);

    // ...other cases
  }
};
```

[View code on GitHub]

If you do this make sure you pass `state` and, if necessary, `action` as arguments.

[3] It's a convention to use uppercase separated by underscores

5.3.2 Action Payloads

Remember, a reducer is a pure function, so it should only work with information that's been passed in (or other pure functions).

But because an action is just an object literal, we can have as many properties as we like on it.

Say we wanted to keep track of lots of different player scores. It would be nice to be able to just have a single action and say *which* player has scored:[4]

```
// destructure the player property from the action
let scores = (state, { player }) => {
  // make a copy of the state
  let copy = { ...state };

  // set the appropriate property
  // alternatively, we could use "computed property names"
  state[player] = state[player] + 1;

  // return the updated state
  return state;
};

const reducer = (state, action) => {
  switch (action.type) {
    // pass along the state *and* action to the score function
    case "PLAYER_SCORES": return scores(state, action);
    // ...other switch cases
  }
};

// add an additional property, "player", to the action object
let action = { type: "PLAYER_SCORES", player: "player1" };
let newState = reducer(initial, action);
console.log(newState); // { player1: 1, player2: 0 }
```

[View code on GitHub]

This additional data is often referred to as the **payload**.

[4]This can be made even shorter using computed property names

5.3.3 Dispatching Actions

Now that we understand what a reducer is, we need to know how to use it.

We never call the reducer directly, instead we use the `dispatch` function that `useReducer` gives us:

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

// the intial state
const initial = {
  player1: 0,
  player2: 0,
};

// reducer functions
const player1Scores = state => {
  return {
    ...state,
    player1: state.player1 + 1,
  };
};

const player2Scores = state => {
  return {
    ...state,
    player2: state.player2 + 1,
  };
};

const reducer = (state, action) => {
  switch (action.type) {
    case "PLAYER_1_SCORES": return player1Scores(state);
    case "PLAYER_2_SCORES": return player2Scores(state);
    default: return state;
  }
};

// the Scorer component
const Scorer = () => {
  // useReducer returns the *current* state
  // and a dispatch function
  const [ state, dispatch ] = useReducer(reducer, initial);
  const { player1, player2 } = state;
```

```
  return (
    <>
      <p
        onClick={ () => dispatch({ type: "PLAYER_1_SCORES" }) }
      >
        Player 1: { player1 }
      </p>

      <p
        onClick={ () => dispatch({ type: "PLAYER_2_SCORES" }) }
      >
        Player 2: { player2 }
      </p>
    </>
  );
};

export default Scorer;
```

[View code on GitHub]

We pass `useReducer` the reducer that we've created as well as an initial value for that state.[5] `useReducer` then gives us back two values: `state` and `dispatch`.

The first value, which we've called `state`, is an object that represents the *current* state (which will be `initial` the first time the component renders). We can destructure this to get the values that we're interested in.

The second value, which we've called `dispatch`, is a function that we call, passing in an action as the first argument. Dispatching an action will cause React to call the reducer function, passing in the *current* state along with the action that was dispatched. Once the reducer returns a version of the state, the component re-renders with the new state.

It's usually always tidier to use `useReducer` when your state contains multiple values. Used properly, it can also make your code much more composable.

[5] In the same way that we have to give an initial value for the accumulator in an array reduce

Double Destructuring

In the above example we could have used multi-level destructuring to save a line of code:

```
// rather than destructuring state on a separate line
// destructure state within the useReducer destructuring
const [ { player1, player2 }, dispatch ] = useReducer(reducer, initial);
```

You can see that we've destructured the state directly within the destructuring of the `useReducer` tuple.

5.4 `useEffect`

We need to be careful about where we put some of our code. We don't want to get in the way of React's "render phase".

With our `class` based components we used the lifecycle methods (e.g. `componentDidMount`) to avoid side-effects (changing the DOM, adding event listeners, &c.) inside `render`, but with hooks based components we don't have methods and our function is effectively the `render` method.

Anything that causes side-effects needs to be wrapped with `useEffect`: this delays running the code until *after* the component has rendered. It can be used to recreate Lifecycle methods, although that is not its only purpose.

A simple example might be if you wanted to updated the `<title>` tag of the page every time something changes on your page:

```
import React, { useState, useEffect } from "react";

const Counter = () => {
  // sets up the initial value of counter
  // and gives back the current value and a function to update counter
  const [counter, setCounter] = useState(0);

  // the event handler
  // uses the provided function to set the value of counter
  const updateCounter = () => setCounter(counter + 1);
```

```
  useEffect(() => {
    document.title = `Clicked ${counter} times`;
  });

  // display counter and setup click handler
  return <p onClick={ updateCounter }>{ counter }</p>;
};

export default Counter;
```

[View code on GitHub]

This function will be called *every* time that component updates, similar to the behaviour of `componentDidUpdate`.

Sometimes, if the component has more than one value in state, the update will not change the value that we're interested in. `useEffect` lets us optionally pass in an array of values that the effect is dependent on. Now the function will only run if the component updates *and* that value has changed:

```
import React, { useState, useEffect } from "react";

const Counter = () => {
  // sets up the initial value of counter
  // and gives back the current value and a function to update counter
  const [counter, setCounter] = useState(0);

  // the event handler
  // uses the provided function to set the value of counter
  const updateCounter = () => setCounter(counter + 1);

  useEffect(() => {
    document.title = `Clicked ${counter} times`;
  }, [counter]); // only run if counter changes

  // display counter and setup click handler
  return <p onClick={ updateCounter }>{ counter }</p>;
}

export default Counter;
```

[View code on GitHub]

React only does a shallow check of the values in the array, so be careful passing in data structures, as it might not behave as you expect.

If we want to recreate the behaviour of `componentDidMount` we can pass an *empty array* as the second argument to `useEffect`. This will run the first time the component renders and then, because it only runs if the values in the given array change, but you've *not passed in any values to check*, it won't run again:

```
import React, { useState, useEffect } from "react";
import axios from "axios";

const StarWarsFolks = () => {
  // setup state
  const [ loaded, setLoaded ] = useState(false);
  const [ people, setPeople ] = useState([]);

  // wrap API request with useEffect
  useEffect(() => {
    axios.get("https://swapi.dev/api/people").then(({ data }) => {
      setLoaded(true);
      setPeople(data.results);
    });
  }, []);

  // render
  return !loaded ? <p>Loading...</p> : (
    <>
      <h2>Some Star Wars Peeps</h2>
      <ul className="list-group">
        { people.map(person => (
          <li
            key={ person.url }
            className="list-group-item"
          >
            { person.name }
          </li>
        )) }
      </ul>
    </>
  );
};
```

```
export default StarWarsFolks;
```

[View code on GitHub]

We can also recreate the `addEventListener` example from the lifecycle methods section:

```
useEffect(() => {
  // assumes we have a function called dragEnd
  // that does something
  window.addEventListener("mouseup", dragEnd)

  // the returned "clean up" function runs on unmounting a component
  return () => {
    window.removeEventListener("mouseup", dragEnd);
  };
}, []); // only run once
```

[View code on GitHub]

Notice that the `componentWillUnmount` code goes in a function which you return from the function passed to `useEffect`.[6] This is a nice way of doing it, as it groups the adding and removing of the event listener in one place.

5.5 Custom Hooks

We need to use `useEffect` within our component function because it uses the state of the component, so that needs to be in scope. However, this means that we can't reuse the logic inside `useEffect`, which restricts function reuse. Sad times.

Say we needed various components that load all the articles from an API: an articles list for the homepage and an archive list for the sidebar. We can write our own custom hook to avoid needing to rewrite the article fetching code:

```
import React, { useState, useEffect } from "react";
import axios from "./axios";

const useGetArticles = (initial) => {
```

[6] This is a "higher-order function": a function that accepts/returns another function

```
  const [articles, setArticles] = useState(initial);

  useEffect(() => {
    axios.get("/blog/articles").then(({ data }) => {
      setArticles(data.data);
    });
  }, []); // make sure to pass in an empty array!

  return [articles, setArticles];
};

export default useGetArticles;
```

[View code on GitHub]

We create a function that *just* deals with the state aspect of articles - no JSX involved. It's good practice to prefix the function name with `use`, to make it clear it's a custom hook. The function takes an initial value as its argument and returns a tuple containing the state and state setting function. We can now use this in any component we like:

```
import React, { useState, useEffect } from "react";
import useGetArticles from "./useGetArticles";

const Articles = () => {
  // use the custom hook - does all the work for us
  const [articles] = useGetArticles([]);

  // display articles
  return (
    <ul>
      { articles.map(article => (
        <li key={ article.id }>{ article.title }</li>
      )) }
    </ul>
  );
}

export default Articles;
```

[View code on GitHub]

All we've really done is moved the article fetching code into a separate function, which we then call instead of `useState`. We don't even need to destructure the setter function as the custom hook is doing all the work for us.

5.6 Rules of Hooks

Hooks can only work if the order that they are called in never changes. Because of this, you cannot call hooks inside loops, conditions, or nested functions. They should **always be at the top level of your React function**.

Under the Hook

Hooks seem almost magical. Where's the state actually getting stored? If the component function is called every time it renders, how is it being kept track of?

How hooks actually work under the hood is a little bit messy. If you didn't know better, you might think that functions like `setState` are pure: but they keep track of values over time, and this is a give away that something very impure is going on under the hood.

You don't need to understand this, but here is a super-simplified version of what's going on (it's waaay more complicated in real life):

```
// an array storing the states for all components
let state = [];

// keep track of the current component being rendered
// updated when each component is rendered
let currentComponent = 0;
let currentUseState = 0;

// use state uses the state and currentComponent
let useState = (initial) => {
    // store the current component
    // need for setValue further down
    let componentIndex = currentComponent;

    // if this component has rendered yet
    // add an empty array to keep track of its states
    if (state[componentIndex] === undefined) {
```

```
        state[componentIndex] = [];
    }

    // keep track of the current useState for this component
    // some components have more than one call to useState
    let useStateIndex = currentUseState;

    // increment for the next run
    currentUseState += 1;

    // set the initial value if it doesn't exist already
    if (state[componentIndex][useStateIndex] === undefined) {
        state[componentIndex][useStateIndex] = initial;
    }

    // create a setValue function
    // updates the current component's state
    // with the given value
    let setValue = (value) => {
        state[componentIndex][useStateIndex] = value;
    };

    // return the current value
    // and the setValue function
    return [state[componentIndex][useStateIndex], setValue];
};

// some super simple "components"
let components = [
    () => {
        let [name, setName] = useState("Bob");
        return <p>{ name }</p>;
    },
    () => {
        let [count, setCount] = useState(1);
        return <p>{ count }</p>;
    }
];

// get the JSX for each component
// then do something with it
components.forEach((component, i) => {
    currentComponent = i; // keep track of current component
    currentUseState = 0; // reset currentUseState
```

```
    let rendered = component();

    // ...DOM stuff
});
```

[View code on GitHub]

As you can see, state is being kept track of outside of the component. And there's a bunch of other global(ish) variables too. Flagrantly impure.

You can hopefully also see why the order that hooks are called in is so important: React *only* has the order to work with.

5.7 Additional Resources

- React: Rule of Hooks
- React's useEffect and useRef Explained for Mortals
- Writing your own React custom hooks
- A Complete Guide to `useEffect`
- Do React Hooks Replace Redux?
- Composing Behavior in React or Why React Hooks are Awesome
- Tuples in JavaScript
- Higher Order Functions

Glossary

- **AJAX (Asynchronous JavaScript and XML)**: a catch-all term for making HTTP requests using JavaScript
- **Asynchronous**: code that doesn't immediately return a value
- **Hooks**: a way to keep track of component state without using classes
- **Life-cycle Methods**: methods on a class based component that are called at specific points during a component's use
- **Lifting State**: taking state out of a child component and putting it in the parent component
- **Payload**: the data that is sent as part of an action
- **Promise**: a functional way to deal with asynchronous code
- **Reducer**: a function that gets given the current state and an "action" and updates the state appropriately
- **Shape**: the structure of your state

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

∞

May 29, 2020