9

Building UIs
with React

Success consists of going from failure to failure without loss of enthusiasm.

- Winston Churchill

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

React

1.1 What is React?

> *React is a declarative, efficient, and flexible JavaScript library for building user interfaces*
>
> - The React Docs

React is a library, created by Facebook[1], that makes doing DOM stuff nice. It's currently the most popular JS library for doing UIs and it looks like it may remain so.[2]

You'll probably remember that doing stuff with the DOM was long winded, overly-complicated, and you had to be very careful about performance issues: making sure you didn't use the DOM more frequently than you needed to.

React deals with all the DOM stuff for us: we just write basic components, that look almost like HTML, and it wires everything up.

React is built on the philosophy that large complex apps should be built by combining small simple components.

1.2 What React Isn't

React is *not* a framework: it only deals with the rendering of HTML and DOM events. That doesn't stop a lot of people from trying to build their entire app us-

[1] But it's open source, so they can't put the usual tracking/democracy-sabotaging code in there
[2] Fingers crossed

ing just React components, but that is not its intended purpose.

For this reason we'll also need to use a few other libraries:

- **ReactRouter**: handles page/URL changes
- **Redux**: looks after the **state** of our app
- **Axios**: lets us deal with HTTP requests

This is a common combination of libraries - an *ad hoc* framework of sorts. As an added bonus, Redux and Axios are both useful even when not making React apps.

When to use React

React is designed for building the UIs of complex single-page web-apps: if you just need a few bits of interactivity on an otherwise non-interactive page, then you probably shouldn't use React.

When you're thinking about using React to build something it's important to consider whether you need to use React at all:

- Using React means the user has to have JavaScript switched on for the site to do *anything*
- Using React will add a significant amount of extra data that needs to be downloaded
- Using React will run slowly on older computers

These rules apply equally to other libraries/frameworks such as Angular, Vue, and Ember.

As a general rule of thumb: if you have various JS bits of the page that don't need to be aware of each other (e.g. a carousel, a menu system, some form validation) then you're probably best using DOM/jQuery. If the different parts of the page *do* need to know about all the others, then you're probably building an app and React would be a good choice.

1.3 Creating a Project

NPM allows us to easily create the **scaffolding** for a React app (this uses Facebook's `create-react-app` package under the hood):

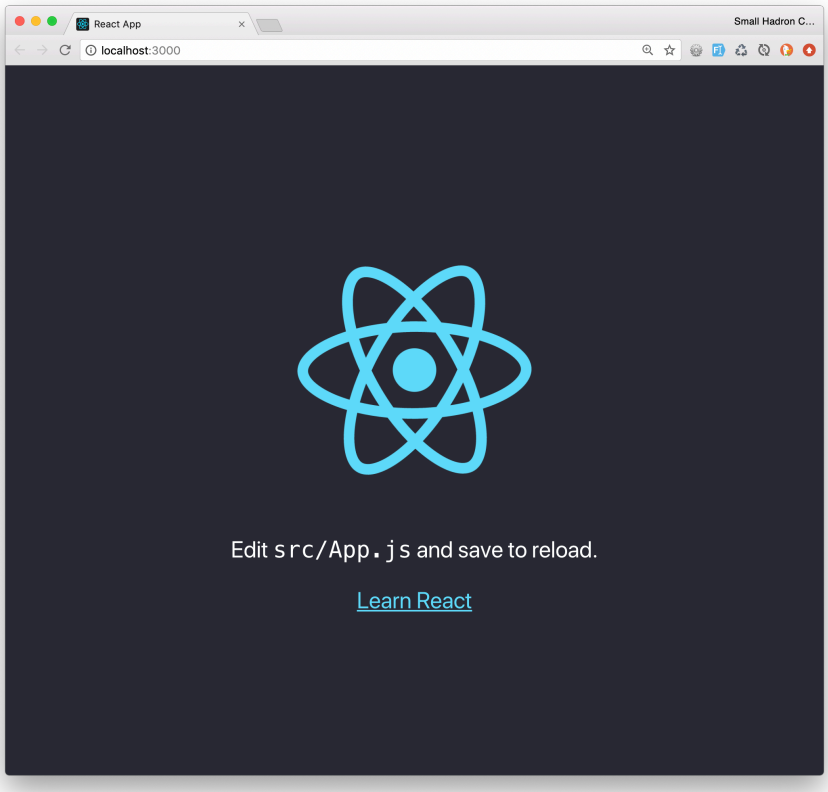
```
npm init react-app project-name
```

This will create a directory called `project-name` in the current working directory. You'll probably want to call your project something more descriptive.

Once the directory is setup we can go into the project directory and run:

```
npm start
```

This will run a web-server on your machine, with the `public` directory as its root, and then load a web browser:



Your shiny new app

The page will automatically refresh when you make changes to the code. It will also tell you about any errors that you make.[3]

Under the hood various packages are being used:

- **Babel**: converts JSX and modern JS into browser compatible code
- **Webpack**: combines all the files together
- **Webpack Server**: serves the web page
- **ESLint**: checks your code for errors

One day you'll probably need to learn how to use all of these tools yourself. For now, it's best to let `create-react-app` do its thang.

Let's have a look inside the newly created folder:

```
.
├── package.json         # which packages to install
├── node_modules         # where npm installs packages
├── public               # the server root
│   └── index.html           # the HTML template
└── src                  # where our code lives
    ├── App.js               # the root React component
    └── index.js             # the JS entry point
```

We'll be doing most of our work in the `src` directory.

1.4 Additional Resources

- React: Create React App
- When Does a Project Need React?
- Website vs Web Application
- The Complete Introduction to React
- React Status: Weekly React Newsletter
- Overreacted: Blog about React by one of the core developers
- Modern JavaScript Explained for Dinosaurs

[3] This is called **linting** your code

Chapter 2

JSX

In React we create **components**. These represent the different UI elements in your app. A component can be made up of other components and we can reuse components in more than one place. Thus, the UI of the entire app is made up entirely of components and sub-components.

The top-level component in the React app scaffolding that's been created for us is called `App` and it lives in `src/App.js`. This component is loaded in from the `src/index.js`, which is where everything gets started. So all of our components will need to be used inside the `App` component or a sub-component thereof.

> **`js` or `jsx`**
>
> It's a good idea to rename `src/App.js` to `src/App.jsx`: this will ensure that your text editor uses the right syntax highlighting. This isn't necessary for all text editors, but it's still useful to be able to look at the extension of a file and see if it's a React component or a regular JS file.
>
> If you've already started the server you'll get a nasty looking error: you'll need to stop the server (`Ctrl+C`) and rerun `npm start`.

2.1 My First Component

If you have a look inside the `src/App.jsx` file, it should look something like this:

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

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to
          ↪  reload.
        </p>
      </div>
    );
  }
}

export default App;
```

[View code on GitHub]

There's a lot of cruft here, so let's update the App component to just say "Hello, world!":

```
// import the React library
import React from "react";

// create our component: just a function that returns JSX
const App = () => (
  <h1>Hello, world!</h1>
);

// export our component
export default App;
```

[View code on GitHub]

Now save the file and check the app in your browser - it should have updated itself.

Notice that we wrap our fat-arrow function with `(` and `)` instead of `{` and `}`. That's because a chunk of JSX code counts as a single value, so we don't want to have to use `return` (which is required if we used curly braces), but we still want to split it onto multiple lines.

`import` & `export`

The `import` and `export` keywords were added as part of ES6. They allow us to easily load bits of JavaScript from other files.

The file that we want to use must `export` something. If the file only contains one thing (such as a React component) then it's common to use `export default`, which means that we don't have to specify which part of the file we want to import.

To `import` a file that has used `export default` we use the `import` keyword, followed by the variable name we want to refer to it by in the current file, then the word `from`, then a relative path to the file (or a package name if it was installed with NPM).

```
// import the package react and call it React in this file
import React from "react";

// import the file MyComponent.jsx
// and call it MyComponent
import MyComponent from "./MyComponent";
```

You can also export different parts of the file using **named exports**:

```
export let add = (a, b) => a + b;
```

These are imported using object destructuring-style syntax:

```
import { add } from "./maths";
```

2.2 Templating Languages

JSX is a **templating language** which mixes HTML-style tags with JavaScript. Basically we can write standard HTML elements in our JavaScript and it will automatically get turned into DOM elements for us.

JSX has a few differences from regular HTML:

- All attributes must be written in camel-case, e.g. `accept-charset` would become `acceptCharset`
- The `class` attribute (for styling with CSS) is called `className`
- All tags should self-close if they are empty (e.g. `<div />`)
- It allows us to easily add event handling
- We can use moustaches (`{` and `}`) to insert JS *expressions*

A contrived example:

```
<form className="form" acceptCharset="utf-8">
    <div className="weird-empty-div" />
    <p>There are { 24 * 365 } hours in a year</p>
</form>
```

Under the Hood

Under the hood Babel transforms the HTML bits of JSX into standard JS:

```
React.createElement( // this is why we have to import React
  "h1",              // element type
  {},                // attributes
  "Hello, world!"    // content
);
```

So, we're not actually writing the code that the browser runs - the browser wouldn't know what to do with a JSX file. We write our code, Babel converts that into standard JavaScript, and then it runs in the browser.

2.3 CSS

Let's update our app to use Bootstrap[1] so that it's a bit nicer to look at.

Add the stylesheet to the `<head>` of `public/index.html`:

```
<link rel="stylesheet"
↪  href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
```

Then add Bootstrap's `container` class to the `<div id="root">`:

```
<div id="root" class="container"></div>
```

This is just a regular HTML file - not JSX - so we don't need to use `className`.

CSS-in-JS

This week we'll just be using a stylesheet and class names to style our components. This is probably how styling would work in a company where front-end development is split between CSS developers and JavaScript developers.

However, it's becoming popular to do "CSS-in-JS", where the styling of individual React components is done as part of the JSX.

This is not built into React, but there are various libraries that let you do it. The scaffolding created by `create-react-app` is setup to allow CSS-in-JS.

[1] We'll be using v3.4

2.4 Sub-Components

If we had to put all of our HTML into a single file, it would get very big. JSX lets us break up different parts of the UI into separate components.

Let's create a `<Header>` component to keep our app title in. Create a new file called `Header.jsx` and add the following:

```
import React from "react";

// we use className to add classes for Bootstrap styling
const Header = () => (
  <header className="page-header">
    <h1>My Amazing App <small>It's actually amazing</small></h1>
  </header>
);

export default Header;
```

[View code on GitHub]

Now update `App.jsx` to include our Header component:

```
import React from "react";

// import the Header component
// we give React components capitalised names
// this makes it clear that they are components
import Header from "./Header";

// use the Header component as if it was an HTML element
const App = () => (
  <Header />
);

export default App;
```

[View code on GitHub]

We import the `<Header>` component that we just created, then JSX lets us use HTML-style syntax to include it as part of the `<App>` component.

The tag name we use is determined by what you call the imported component:

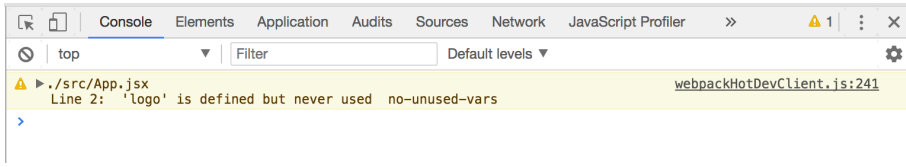
```
// import the component from the Header file
// but call it Fishsticks
import Fishsticks from "./Header";

// now we have to use the Fishsticks component
const App = () => (
  <Fishsticks />
);
```

[View code on GitHub]

Extra-Helpful Console

As always with JavaScript in the browser, you should have the console open in Developer Tools (Mac: `Command+Option+J` / Win: `Control+Shift+J`) when developing a React app.

You will often get given advice about best-practices and other minor tweaks that will make your code easier to maintain.



Thanks console! Thansole

You should **always** read these bits of advice and fix them immediately, otherwise bugs will start to sneak into your code.

Next, let's add a `<Content>` component to hold the main part of our app.

```
import React from "react";

// a component can only have one top-level element
// so we wrap everything in a <main> tag
const Content = () => (
  <main>
```

```
    <p className="lead">This app is the best</p>
    <img className="img-thumbnail" alt="A cat"
    ↪  src="https://goo.gl/tRdW93" />
    <p className="alert alert-success">
      If you'd like to fund us for £500k+, please get in touch
    </p>
  </main>
);

export default Content;
```

[View code on GitHub]

Let's update it so we can easily change the amount of money:

```
import React from "react";

// just some standard JS
const valuation = 5e6;

// use the JS number toLocaleString() method to format
const formatted = valuation.toLocaleString(
  "en-GB", { style: "currency", currency: "GBP" }
);

const Content = () => (
  <main>
    <p className="lead">This app is the best</p>

    <img className="img-thumbnail" alt="A cat"
    ↪  src="https://goo.gl/tRdW93" />

    <p className="alert alert-success">
      { /* we can use the variable from above */ }
      { /* by wrapping it in curly braces */ }
      If you'd like to fund us for { formatted }+, please get in touch
    </p>
  </main>
);

export default Content;
```

[View code on GitHub]

We're just using standard JS code outside the component: it's only when we're inside JSX tags that we have to do anything special.

Don't forget to add the `<Content>` component to `<App>`:

```
import React from "react";

import Header from "./Header";
import Content from "./Content";

// wrap in a <div> so there's only one top-level element
const App = () => (
  <div>
    <Header />
    <Content />
  </div>
);

export default App;
```

[View code on GitHub]

2.5 `React.Fragment`

In the last example we had to wrap our two sub-components in a `<div>` because a React component can *only return a single element*. However, the `<div>` doesn't add any semantic value to our HTML code and we're not using it for styling, so it would be better if we could return a *fake* element that doesn't actually get rendered - sort of like a document fragment.

That's where `React.Fragment` comes in: it lets us return a single element from a component without adding an additional element to the DOM.

```
// add an import of Fragment from react
import React, { Fragment } from "react";

import Header from "./Header";
import Content from "./Content";

// wrap with Fragment
```

```
// won't appear in the final ouput
const App = () => (
  <Fragment>
    <Header />
    <Content />
  </Fragment>
);

export default App;
```

[View code on GitHub]

You should use `React.Fragment` whenever you don't need the wrapping element to appear in the DOM. If you don't, the HTML Standards Goblin will steal all the vowel keys from your keyboard while you sleep.

Fragments are so useful that there's a shorthand for it:

```
<>
  <Header />
  <Content />
</>;
```

We open with `<>` and close with `</>`. This isn't fully supported with all tooling (e.g. text editors, syntax highlighters, &c.), but it's worth giving it a go and seeing if it works with your setup.

2.6 Sub-Sub-Components

Our sub-components can have sub-components of their own, which in turn can have their own sub-components. Of course, nothing about a particular component makes it inherently a *sub*-component, it's entirely about whether you use it in *another* component or not.[2]

[2] In the same way that an HTML element isn't necessarily a child or parent of any other HTML element - it's entirely about how you use them

Let's pull out the funding code into its own component:

```
import React from "react";

const valuation = 5e6;

// use the JS number toLocaleString() method to format
const formatted = valuation.toLocaleString(
  "en-GB",
  { style: "currency", currency: "GBP" }
);

const Funding = () => (
  <p className="alert alert-success">
    If you'd like to fund us for { formatted }+, please get in touch
  </p>
);

export default Funding;
```

[View code on GitHub]

And now update `<Content>` to use the new component:

```
import React from "react";

// import Funding
import Funding from "./Funding";

const Content = () => (
  <main>
    <p className="lead">This app is the best</p>
    <img className="img-thumbnail" alt="Cat"
    ↪  src="https://goo.gl/tRdW93" />
    <Funding />
  </main>
);

export default Content;
```

[View code on GitHub]

We don't need to change anything in `<App>`, as it's just using the `<Content>` component.

Now, if we wanted to, we could using the `<Funding>` component elsewhere on our site, without it being tied to the `<Content>` component.

2.7 Working with Arrays

JSX also makes it easy to work with arrays using the `.map()` array iterator:

```
import React from "react";

// just an array of numbers
let numbers = [1, 2, 3, 4];

const Pagination = () => (
  <ul className="pagination">
    { /* use map to output an <li> for each */ }
    { /* item in the array */ }
    { numbers.map((value, index) => (
      <li key={ index }>
        <a href={ "/page/" + value }>{ value }</a>
      </li>
    ))}
  </ul>
);

export default Pagination;
```

[View code on GitHub]

The `.map()` returns an array of `<li>` elements, which React then adds as children of the `<ul>` element.

JSX or JS?

One thing that can take a while to get used to with JSX is the current context that you're in: JSX or JS?

As a general rule, you're in JavaScript land until you open your first JSX tag. Once

you go into JSX land you need to use a moustache ({) to go into JS-in-JSX land. Once you're in JS-in-JSX land you don't need to use moustaches anymore *unless* you start a new JSX tag.

JS-in-JSX land is much like JavaScript land, except every instance of JS-in-JSX must be an expression (i.e. it must evaluate to a single value). So you can't use `if` statements, variable declarations, and the like.

```
import React from "react";

// JS land
// plain old javascript stuff out here

const Thing = () => (
  <div> { /* in JSX mode now, so need moustaches to do JS */ }

    { /* basic JS expression */ }
    <p>{ 12 * 12 }</p>

    { /* more complex expression */ }
    { /* all in JS, so no extra moustaches needed */ }
    <p>{ [1, 2, 3, 4].map(n => n * n).join(", ") }</p>

    { /* really complex expression */ }
    { /* second map goes into JSX mode when we open <span> */ }
    { /* so need moustaches to get back into JS-in-JSX mode */ }
    <p>
      { [1, 2, 3, 4].map(n => n * n).map(n => <span>{ n }</span>) }
    </p>

    { /* to do a comment in JSX, open a moustache to */ }
    { /* get into JS-in-JSX mode, then use comments */ }
  </div>
);

export default Thing;
```

We can use `.map()` on *any* array, including ones with more complex values:

```
import React from "react";

let animals = [
  { name: "Rafiki", animal: "Baboon" },
  { name: "Paddington", animal: "Bear" },
  { name: "Judy", animal: "Rabbit" },
  { name: "Bing Bong", animal: "?" },
];

const Animals = () => (
  <table className="table">
    <thead>
      <tr><th>Name</th><th>Animal</th></tr>
    </thead>
    <tbody>
      { animals.map((a, i) => (
        <tr key={ i }><td>{ a.name }</td><td>{ a.animal }</td></tr>
      ))}
    </tbody>
  </table>
);

export default Animals;
```

In both examples above we had to add a special `key` attribute to the `<tr>`s. This needs to be unique for each element, so we use the array index. React can use this to efficiently re-render large lists of items.

What `key`?

If you're just using an array of items then using the array index as the key is probably fine: it's guaranteed to be unique (unlike the items in the array). But it won't give you any performance benefits in more complex code.

Of course, often in more complex apps you'll get your arrays of data from an API, which will mean each item has an associated unique `id` of some sort. If you use this as the `key` attribute you will be helping React when it comes to re-rendering the list.

2.8 Additional Resources

- React: Introducing JSX
- React: JSX in Depth
- React: Fragments
- MDN: `import`
- MDN: `export`
- Bridging the Gap Between CSS and JavaScript: CSS-in-JS

Chapter 3

Props

Currently our components aren't very reusable. Say we wanted to create a component that shows an image in a panel:

```
import React from "react";

const Figure = () => (
  <figure className="panel panel-default">
    <div className="panel-body">
      <img
        className="img-thumbnail"
        alt="A cat, strutting its stuff!"
        src="https://goo.gl/tRdW93"
      />
    </div>
    <figcaption className="panel-footer">
      A cat, strutting its stuff!
    </figcaption>
  </figure>
);

export default Figure;
```

[View code on GitHub]

As is, the image and caption are hard-coded in the component. If we wanted to display *another* `<figure>`, we'd need a whole new component. But other than the caption text and the image URL, the rest of the component is just boilerplate which would be the same for any `<figure>`.

JSX lets us pass values into a component tag using HTML-attribute-esque syntax:

```
<Figure
  caption="A cat, strutting its stuff!"
  src="https://goo.gl/tRdW93"
/>
```

These are called **props** (short for "properties").

We'll need to update the `<Figure>` component to use these props:

```
import React from "react";

// props are passed in as an object as the
// first argument to our component
const Figure = props => (
  <figure className="panel panel-default">
    <div className="panel-body">
      { /* use the props we passed in: props.caption and props.src */}
      { /* attributes using JS values don't have quotation marks */ }
      <img
        className="img-thumbnail"
        alt={ props.caption }
        src={ props.src }
      />
    </div>
    <figcaption className="panel-footer">
      { props.caption }
    </figcaption>
  </figure>
);

export default Figure;
```

[View code on GitHub]

Anything that you add as a prop when you use the component gets passed into the component as a property of the props object, which is passed in as the first argument to the component function.

Destructuring Props

We can use object-destructuring syntax to make this even neater:

```
import React from "react";

// destructure the props object when it's passed in
const Figure = ({ caption, src }) => (
  <figure className="panel panel-default">
    <div className="panel-body">
      <img className="img-thumbnail" alt={ caption } src={ src } />
    </div>
    <figcaption className="panel-footer">{ caption }</figcaption>
  </figure>
);

export default Figure;
```

[View code on GitHub]

This makes it much easier to work with multiple props inside the component.

Now, if we wanted multiple `<Figure>`s:

```
<Figure
  caption="A cat, strutting its stuff!"
  src="https://goo.gl/tRdW93"
/>

<Figure
  caption="Another cat, also strutting its stuff!"
  src="https://goo.gl/FYXPaS"
/>
```

Props can only be passed *down* from parent components to their sub-components. This is called **one-way data flow**. This might seem like a limitation, but it's actually a very good idea when it comes to building large apps as it means that components never need to know anything about the components in which they are used, so they can be used in *any* component.

JS Types and Props

If we want to pass down string using props then we can use the HTML-like syntax:

```
<Thing blah="A String" />
```

If we want to pass other types of values (numbers, booleans, array, etc.) we can use moustaches to pass them.

A number:

```
<Thing blah={ 12 } />
```

An array:

```
<Thing blah={ [1, 2, 3, 4] } />
```

An object literal (notice the *double* curly-braces: the first to get into JS-in-JSX land and the second to open an object literal):

```
<Thing blah={ { name: "Ada", job: "Programmer" } } />
```

We can also use moustaches to pass down values stored in variables:

```
let x = ["A", "B", "C", "D"];

// ...elsewhere in JSX-land

<Thing blah={ x } />
```

Let's update our `<Header>` component so we can pass in the title and subtitle:

```
import React from "react";

const Header = ({ title, subtitle }) => (
  <header className="page-header">
    <h1>{ title } <small>{ subtitle }</small></h1>
  </header>
);

export default Header;
```

[View code on GitHub]

And then pass in the props from `<App>`:

```
<Header title="My Amazing App" subtitle="Is actually amazing" />
```

3.1 The Ternary Operator

We can only use JS expressions in JSX moustaches, which means we can't use `if` statements for deciding what to show. However, you'll remember that the ternary operator *does* evaluate to a value, so we can use it to do basic logic in our JSX.

For example, say we don't want the `<small>` element to render at all if no subtitle prop is passed into the component:

```
import React from "react";

const Header = ({ title, subtitle }) => (
  <header className="page-header">
    <h1>
      { title }
      { /* return null if you don't want anything rendered */ }
      { !subtitle ? null :
        <small style={ { marginLeft: 10 } }>{ subtitle }</small>
      }
    </h1>
  </header>
```

```
);

export default Header;
```

[View code on GitHub]

If we output `null` in the JSX then React just ignores it and doesn't render anything. We've used `!subtitle` as it lets us put `null` as the *first* value, which makes it easier to read what the ternary does.

Also notice that we've had to add a left margin to the `<small>` element. React strips any whitespace between elements, which means if we don't add this the `<small>` element would smash into the main title text. We pass in an object literal with style properties. React automatically adds the "px" for us.

3.2 The `children` Prop

When we're using HTML, we often put content *between* opening and closing tags:

```
<h1>Kittens</h1>
```

JSX lets us do something similar with our components:

```
<Header>My Awesome App</Header>
```

Whatever we put between the opening and closing tags gets passed in as the special `children` prop. We can then use this in our components:

```
import React from "react";

// accept children, no more title prop
const Header = ({ children, subtitle }) => (
  <header className="page-header">
    <h1>
      { /* use children instead of title */}
      { children }
      { !subtitle ? null :
        <small style={ { marginLeft: 10 } }>{ subtitle }</small>
```

```
      }
    </h1>
  </header>
);

export default Header;
```

[View code on GitHub]

We can only pass in a single value with the `children` prop, so if we want to pass the subtitle in too we'll still need one prop:

```
<Header subtitle="Is really awesome!">My Awesome App</Header>
```

We can pass *any* valid JSX in as children, including other components we've created:

```
<Header subtitle="Is really awesome!">
    My Amazing App
    <Funding />
</Header>
```

This is an incredibly powerful concept and is used by a lot of the libraries we'll be looking at in the next few weeks.

3.3 Default Props

For some of our components we might want to set **default** prop values. You can do this by adding properties to the `defaultProps` property of any component:

```
import React from "react";

const Figure = ({ caption, src }) => (
  <figure className="panel panel-default">
    { /* ...etc. */ }
  </figure>
);

// add default values for the caption and src props
// if the prop is not given these will be used
Figure.defaultProps = {
  caption: "A caption",
  src: "http://via.placeholder.com/350x350",
};

export default Figure;
```

[View code on GitHub]

This is particularly useful for components that do not render correctly if certain props are not provided.

Default props are only used *if the prop isn't present*. They will not be used if a null/empty/falsey value is given (otherwise we couldn't usefully pass in such values).

3.4 Additional Resources

- React: Components and Props

Chapter 4

Events & State

4.1 Events

React lets us easily hook into events using the `on...` JSX attributes:

```
import React from "react";

const App = () => (
  <header
    className="jumbotron"
    onClick={ () => console.log("clicked") }
  >
    <h1>Hello</h1>
  </header>
);

export default App;
```

[View code on GitHub]

There is an `on...` attribute for pretty much every type of event that the DOM supports. You can find a full list on the React Docs. Under-the-hood the JSX gets translated in to plain-old DOM event-handlers. React is also smart enough to automatically remove event handlers for components that are no longer visible (which is a complete nightmare to do manually).

You can only add event handlers to HTML elements, not to React components: components are just wrappers for HTML elements, so they don't exist in the final DOM structure.

Old Skool Events

In the before-times, when JS was still in its infancy, you would write event-handlers in-line on the elements themselves:

```
<p onClick="console.log('Noooooooooo')">
  The 90s weren't all great
</p>
```

Needless to say, this was horrific.

When JSX first came out, some developers saw the event handler syntax, curled up into a ball and began weeping uncontrollably. However, although they might at first look quite similar, they're actually *very* different.

4.2 Class Components

We don't really want to have our event handler functions mixed into our JSX: it would become hard to follow for anything but the simplest functions.[1]

We *could* store the event handler outside the component function, but this will cause us issues when it comes to state. However, we can use a *class*:

```
// we have to import Component from react
import React, { Component } from "react";

// use standard JS class syntax
// we "extend" React's Component class
class App extends Component {
  // put the event handler in a method
  // this keeps it separate from our JSX
  handleClick() {
    console.log("Clicked");
  }

  // our class needs a render method - this should return the JSX
  render() {
```

[1] It also makes your React apps *much* less performant

```
    // because we're in a class we can refer
    // to the method using "this"
    return (
      <header className="jumbotron" onClick={ this.handleClick }>
        <h1>Hello</h1>
      </header>
    );
  }
}

export default App;
```

[View code on GitHub]

From Function to Class

To get from a function based component to a class based component:

- Create a `class` with the same name as the function you're replacing, and give it a `render` method
- The `render` method should return the exact JSX that the function version returned
- `props` don't get passed into `render`, instead you use `this.props`, which is automatically setup for you when you extend `Component`
- You can use object destructuring on `this.props` to avoid rewriting any of your JSX

For example:

```
import React from "react";

const Header = ({ children, subtitle }) => (
  <header className="page-header">
    <h1>{ children } <small>{ subtitle }</small></h1>
  </header>
);

export default Header;
```

[View code on GitHub]

Becomes:

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

// turn into a class of same name
// and extend Component
class Header extends Component {
  // add a render function
  render() {
    // destructure this.props so we don't need to update the JSX
    let { children, subtitle } = this.props;

    // the return of render is exactly the same as what you
    // had in the original function version
    return (
      <header className="page-header">
        <h1>{ children } <small>{ subtitle }</small></h1>
      </header>
    );
  }
}

export default Header;
```

[View code on GitHub]

It's best to write your components as functions and only turn them into `class` based components if you need to.

4.3 State

As well as setting up `this.props` for us, extending the `Component` class also sets up the special `this.state` property.

We've looked at the idea of **state** before: we use **long-lived variables** to store values that can potentially change over time. These are normally updated when an event fires.

Using state with `class` components consists of three parts:

- Setting up our **initial state**
- Displaying values based on `this.state` in our JSX
- Updating `this.state` when events are fired

We cannot use state with our function based components, as they do not have access to `this`. That's why functional components are usually referred to as **stateless components** and `class` based components are sometimes called **stateful components**.

To set up the initial state we need to use `class`'s `constructor` method:

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

class App extends Component {
  // add a constructor method
  // it gets passes props as its first argument
  constructor(props) {
    // make sure you always add this, it makes Component work
    super(props);

    // setup our state
    // just a plain old JavaScript object
    this.state = {
      counter: 0,
    };
  }

  // ...etc.
}

export default App;
```

[View code on GitHub]

`this.state` is just a plain old JavaScript object ("POJO"). It can take as many properties as you like, using any values you can usually have as part of an object literal.

Next let's update our JSX to display something based on the state:

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

class App extends Component {
  // ...etc.

  render() {
    return (
      <header className="jumbotron" onClick={ this.handleClick }>
        { /* use the state - will use the inital state the first time
        ↪  */ }
        <h1>{ this.state.counter }</h1>
      </header>
    );
  }
}

export default App;
```

[View code on GitHub]

Finally, let's update the `handleClick()` method to update the state:

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

class App extends Component {
  // ...etc.

  handleClick() {
    // get current value of counter
    let current = this.state.counter;

    // set new value
    // pass in a POJO with values we want to update
    this.setState({ counter: current + 1 });
  }
```

```
  // ...etc.
}

export default App;
```

[View code on GitHub]

We can't just use `this.state.counter += 1` to update `this.state` - if we did, React wouldn't know that anything has changed.

Instead we use the `this.setState()` method: this lets React know that something has changed, so it should re-run the `render` method of the component.

Using `this.setState()`

We have to be careful to *only* use `this.setState()` to change values in the state. This is particularly an issue if you're storing an array or an object in the state.

For example if you use array methods like `.pop()` or `.push()` you change the *original* array (you don't get back a new one). So if we use these methods to manipulate arrays stored in `this.state` we might accidentally change their values. So, as a general rule you should avoid methods that change (or "**mutate**") arrays or objects.

Luckily we can use the spread operator (`...`) to always return a fresh version of objects/arrays. And the array iterator methods *always* return a new version of the array, so are safe too.

You should also be careful not to use the shorthand assignment operators like `+=` and `-=`, as these will also update the value in the state.

When you're developing your React app you'll get a warning in the console if you mutate the state unintentionally. The code that checks this is automatically removed from your React app when you build your app for deployment, as it is a bit of a performance killer.

4.3.1 F**k `this`

> *In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders*
>
> - Douglas Crockford

What we've done so far *should* work... but it won't because `this` is broken in JavaScript.

Because of this we need to **bind** any of methods that use `this` inside the `constructor` method:

```
constructor(props) {
    // ...etc.

    // force this to always be *this* this in handleClick
    this.handleClick = this.handleClick.bind(this);
}
```

When React calls `this.handleClick` the value of `this` gets lost. Using `bind` makes sure that the method always knows what `this` should be.[2]

React Developer Tools

If you try to use `console.log(this.state)` in an event handler where you've used `this.setState()` you'll probably find that it doesn't log the values you'd expect. That's because `this.setState()` doesn't actually update the state until *after* the event handling function has finished.

React Developer Tools lets you inspect the current state of any component in your app. This means that you rarely need to use `console.log()` at all.

4.3.2 The Shape of the State

State is to keep track of things that change in your component. You'll need to ask yourself two questions to work out what your state should look like:

[2] Once we start using Redux we'll mostly stick to stateless components, so this stops being something we need to worry about.

- What do we want to be able to change in the component?
- What types would be best to track these changes?

The state object can contain as many properties as you like, but you should aim to store the least amount of information necessary to work out how to render your component.

Working out the state's **shape** can only come with practice.

Tips

- If it can be in one of two states, use a boolean.
- Use arrays to keep track of all previous values.
- If you always update two values at the same time with `this.setState()`, think about whether you can work out one of the values from the other. If you can, you might not need to store both in the state.

Creating a Component

If you do things in the following order and test at each stage you should avoid getting tied in knots:

1. Add in the boilerplate: the `import`s, `exports` and basic `class` declaration
2. Write a `render` method that outputs JSX that looks about right
3. Think about the shape of the state and add the `constructor` method to set up your initial state
4. Update the `render` method to use the initial state
5. Try temporarily changing the initial state values to check your JSX changes as you expect when the state changes
6. Add the event handlers

4.4 Additional Resources

- React: State and Lifecycle
- React: Handling Events

- Props vs. State
- Stateful vs. Stateless Components
- How does React decide to re-render a component?
- This is why we need to bind event handlers in Class Components in React
- 5 things you didn't know about React DevTools

Chapter 5

Controlled Components

5.1 Forms

One thing that we have to be particularly careful with is form inputs.

A component's state is meant to represent *any* values that change in the DOM. That means that, as the user types into a form input, the state should be updated.

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

// a fairly useless input component
// the user can change the value of the input
// but no way of getting/setting the value
class Input extends Component {
  render() {
    return (
      <div className="form-group">
        <label>{ this.props.label }</label>
        <input className="form-control" name={ this.props.name } />
      </div>
    );
  }
}

export default Input;
```

[View code on GitHub]

So, we should store the current value of the input item in the `state`:

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

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

    // add the value of the input to the state
    this.state = { input: "Test Value" };
  }

  render() {
    return (
      <div className="form-group">
        <label>{ this.props.label }</label>

        { /* use the value of the state */ }
        <input
          value={ this.state.input }
          name={ this.props.name }
          className="form-control"
        />
        <p className="help-block">
          Length: { this.state.input.length } characters
        </p>
      </div>
    );
  }
}

export default Input;
```

[View code on GitHub]

The input's value is now stored in the state but *we can't change the value of the input!*

So, we'll need to update the state when the value changes. We can use the **event object** to do this. React passes the same event object that we got given when we added events to the DOM.

We can use the `currentTarget` property of the event object to get back whatever DOM element our event was called on, in this case the `<input>`. We can then get

the value of the `<input>` using the DOM `value` property:

```
// e is the standard DOM event object
handleChange(e) {
    // e.currentTarget: DOM element we attached the event handler to
    // use the value property to read its current value
    this.setState({ input: e.currentTarget.value });
}
```

We'll need to add an event handler to the `<input>`:

```
<input
  onChange={ this.handleChange }
  value={ this.state.input }
  name={ this.props.name }
  className="form-control"
/>
```

And don't forget to `bind` the `handleChange` method:

```
constructor(props) {
    // ...
    this.handleChange = this.handleChange.bind(this);
}
```

Now when a user types into the input the state will update too. This is called a **controlled component**. All form elements in a React app should be controlled components.

Some libraries/frameworks like Vue.js support "two-way data binding" which removes the need to handle form input manually. It seems like a good idea initially, but actually once you start building more complex apps it stops being particularly useful and can lead to all sorts of hard-to-spot bugs. You can remove a lot of the repetitive boilerplate just by using sub-components carefully.

5.2 Notes About React Form Elements

There are a few things to be aware of when using form elements with React:

- If you set the `value` attribute you *have* to set an `onChange` event handler (otherwise it would be impossible for the user to change the input's value)
- The `onChange` event is closer to the `input` DOM event: it gets fired on every keystroke for an input.
- If you want to set a default value you should use the `defaultValue` attribute.
- The `value` property works for all form elements, including ones like `<select>` which don't have a `value` property when using native DOM.

5.3 Additional Resources

- React: Forms

Chapter 6

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.

6.1 Lifting State

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 Fragment
import React, { Fragment, 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() {
    // need to use a Fragment to output multiple elements
    return (
      <Fragment>
        <Button name="First" />
        <Button name="Second" />
      </Fragment>
    );
  }
}

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

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

  render() {
    return (
      <Fragment>
        { /* 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 } />
      </Fragment>
    );
  }
}

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

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

  render() {
    return (
      <Fragment>
        { /* 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 }) }
        />
      </Fragment>
    );
  }
}

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*.

6.2 Passing Data Up

We can use the same 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, { Fragment, 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 (
      <Fragment>
        <p onClick={ this.handleClick }>{ count }</p>
      </Fragment>
    );
  }
}

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

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>`.

6.3 Additional Resources

- React: Lifting State Up
- Building Reusable Components Using React

Chapter 7

React Router

7.1 Routing

Routing is the process of taking a URL and deciding what code should run.

For example:

- `app.dev/`: go to the homepage
- `app.dev/login`: show a login form
- `app.dev/register`: show a registration form
- `app.dev/posts`: show a list of all the posts
- `app.dev/posts/24`: show the post that has the `id` of 24

There are a lot of intricacies to writing routing code, so we're better to let someone else do it for us.

That's where **ReactRouter** comes in.

7.2 Setup

First we need to install ReactRouter:

```
npm install react-router-dom
```

There are various versions of ReactRouter. We want the DOM version as we're building a single page web app.

Then we need to update our `App.jsx` to use routing.

First we need to import the ReactRouter components `<Router>` and `<Route>`. Then we need to wrap our existing `App.jsx` code with the `<Router>` component:

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

// import the ReactRouter components
import {
  BrowserRouter as Router,
  Route,
} from "react-router-dom";

import Header from "./Header";
import Buttons from "./Buttons";

// wrap the entire app with the <Router> component
// this is simply using the children prop under the hood
const App = () => (
  <Router>
    <Fragment>
      <Header subtitle="Space Wombats">App</Header>
      <Buttons />
    </Fragment>
  </Router>
);

export default App;
```

[View code on GitHub]

Now we can begin to add **routes**.

7.3 Routes

7.3.1 Basic Routes

Visiting any URL will result in seeing the `<Header>` and `<Buttons>`.

But, let's say we only wanted to show the `<Buttons>` element on the homepage. We could add a `<Route>` with a `component` prop:

```
const App = () => (
  <Router>
    <Fragment>
      { /* always show the Header component */ }
      <Header subtitle="Space Wombats">My App</Header>
      { /* if the URL is the homepage, load the Buttons component  */ }
      <Route exact path="/" component={ Buttons }/>
    </Fragment>
  </Router>
);

export default App;
```

[View code on GitHub]

Now if we visit the homepage of our app we'll see `<Buttons>`, but if we go to any other URL we'll only see the `<Header>`.

Effectively a `<Route>` is like a conditional that says "if the URL matches the given path, show the component". Anything not in a `<Route>` component will *always* render.

7.3.2 Routes with Props

How about if we wanted to show our `<Figure>` element if we visit `/figure`?

A `<Figure>` component requires specific props in order to work, so we need to provide a bit more information.

We need to use `<Route>`'s `render` prop:

```
<Route path="/figure" render={ () => (
    <Figure
        caption="A cat, strutting its stuff!"
        src="https://goo.gl/tRdW93"
    />
) } />
```

`render` lets us pass in a function that returns a fully formed component *including* props.

7.4 Matches

Say we had a list of articles and we wanted to show the user one specific article. On a standard website we might use a URL structure like the following:

- `/articles/1`: show the article with id 1
- `/articles/2`: show the article with id 2
- `/articles/3`: show the article with id 3
- &c.

We can reproduce this using ReactRouter's `match` property:

```
<Route path="/articles/:id" render={ ({ match }) => (
    <Article article={ match.params.id } />
) } />
```

The `:id` part of the URL is a **parameter**. We can call it whatever we like, as long as it starts with `:`. We can then access the value of a specific parameter by looking at the `match.params.<parameterName>` property that ReactRouter passes into the `render` function.[1]

[1] It also passes this as a prop to any components when we use the `component` prop on a `Route`.

For example, We can then use the passed value to show a particular article:

```
import React from "react";

const Article = ({ article }) => (
  <article>
    <h2>Article #{ article }</h2>
    { /* some code to show a specific article based on the id */ }
  </article>
);

export default Article;
```

[View code on GitHub]

7.5 Links

We need a way of getting round our app. Normally we'd use an `<a>` tag, but that would cause the browser to load the page from scratch, and all our app's state would be lost.

ReactRouter provides us with the `<Link>` component:

```
import React from "react";

// import Link from ReactRouter
import { Link } from "react-router-dom";

const LinkList = () => (
  <ul className="list-group">
    <li className="list-group-item">
      { /* use Link instead of <a> */ }
      <Link to="/">Home</Link>
    </li>
    <li className="list-group-item">
      { /* use Link instead of <a> */ }
      <Link to="/figures">Figures</Link>
    </li>
  </ul>
);
```

```
export default LinkList;
```

[View code on GitHub]

As long as we use `<Link>` components we'll never cause a hard-refresh, so our app's state will not be lost.

Under the hood ReactRouter is creating an `<a>` element, adding an event handler with a call to `e.preventDefault()`, and then updating the address bar and browser history for us.

7.6 404s

We need some way to handle 404 errors.

First, we'll need to create a `<FourOhFour>` component:

```
import React from "react";

const FourOhFour = () => <p>Page not found</p>;

export default FourOhFour;
```

Next we will need the `<Switch>` component:

```
import {
    BrowserRouter as Router,
    Route,
    Switch,
} from "react-router-dom";
```

We can then wrap all our routes in the `<Switch>` component and add our `<FourOhFour>` component as the last item:

```
<Switch>
    <Route exact path="/" component={ Buttons }/>
    <Route component={ FourOhFour }/>
</Switch>
```

Now, if none of the routes match it will display the last item in the `<Switch>` - like a `default` statement in a standard `switch()` conditional.

Other than your 404 component, you should only wrap `<Route>` components with `<Switch>`, otherwise things won't display as you'd expect.

7.7 Additional Resources

- React Router

Asynchronous means 'at the same time"

XML very verbose, angle bracket land <employee></employee>

Chapter 8

Working with APIs

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

8.1 AJAX

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 people still used XML to transfer data:

```
<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>
```

```
    </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.

8.2 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.[1]

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",
```

[1] 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");
```

8.2.1 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, API key, and the `Accept` 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({
    // use your own url
    baseURL: "http://username.restful.training/api/",

    // use your own key
    params: {"key": "4fdea06a65ba1491091c0db709faf0cce944067a"},

    // make sure we get JSON back
    headers: {"Accept": "application/json"},
});
```

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");
```

8.3 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 actually dealt with a similar concept when we wrote event handlers.

8.4 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.

8.4.1 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 actually 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.

8.5 Lifecycle Methods

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 gets responds successfully, clear the inputs
        this.setState({ title: "", article: "" });
    });
}
```

If we're going to *fetch* data from an API then we'll need some way to run a bit of code when a component first appears on screen.[2]

We can do this using the `componentDidMount` "Lifecycle method". 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:[3]

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

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

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

  // runs when the component first renders
  componentDidMount() {
    // make the GET request
    axios.get("https://swapi.co/api/people").then(({ data }) => {
      // once the data has come back update the component state
```

[2]This is subtly different from when it first gets created in code, `constructor`
[3]If a component is removed and then re-added, the `componentDidMount` method will run again.

```
      this.setState({
        loaded: true,
        people: data.results,
      });
    });
  }

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

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

export default StarWarsFolks;
```

[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.

8.6 Additional Resources

- MDN: Promises
- Promises: A Technical Look
- Axios
- How To Make HTTP Requests Like a Pro with Axios
- MDN: `async` functions
- Lifecycle Methods: `componentDidMount`

Chapter 9

Deploying a React App

If you've used `npm init react-app`, you can simply run `npm build` to create a copy of your site that you can run on any server. It will create a directory called `build` that contains an `index.html` file and the fully compressed and transpiled JS.

You'll need to rerun `npm build` every time you want to publish a new version of the site - it doesn't keep itself up to date.

You shouldn't add the `build` directory to Git, as it's easy to recreate by just running `npm build`.

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
- **Component**: a reusable module representing some small part of a user interface
- **Controlled Component**: a form input whose value is controlled by the component's state
- **JSX**: React's templating language. Converts a mix of HTML and JavaScript into HTML
- **Lifting State**: taking state out of a child component and putting it in the parent component
- **Moustaches**: a colloquial name for `{` and `}` when used in templating languages
- **Promise** a functional way to deal with asynchronous code
- **Props**: props are written like HTML attributes and allow us to pass values into sub-components
- **Router**: some code that decides what to do depending on the provided URL
- **Route**: a piece of code to run given a specific URL
- **Shape**: the structure of your state
- **Stateful Component**: a `class` with a `render` method and the `this.props` and `this.state` properties
- **Stateless Component**: a function that accepts props and returns JSX
- **State**: keeping track of changes using long-lived variables
- **Sub-Component**: a component that is used by another component
- **Templating Language**: converts a mixture of markup and a programming language to create pure markup

- **URL Parameter**: part of a URL that can be used like a variable

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 28, 2019