Contents

1 Welcome to the Pure React quick start course!

Over the next 5 chapters you will get started learning React.

This course is designed for total beginners to React, as well as folks who've tried to learn in the past but have had a tough time. I think I can help you figure this out. Whether or not you know much JS yet.

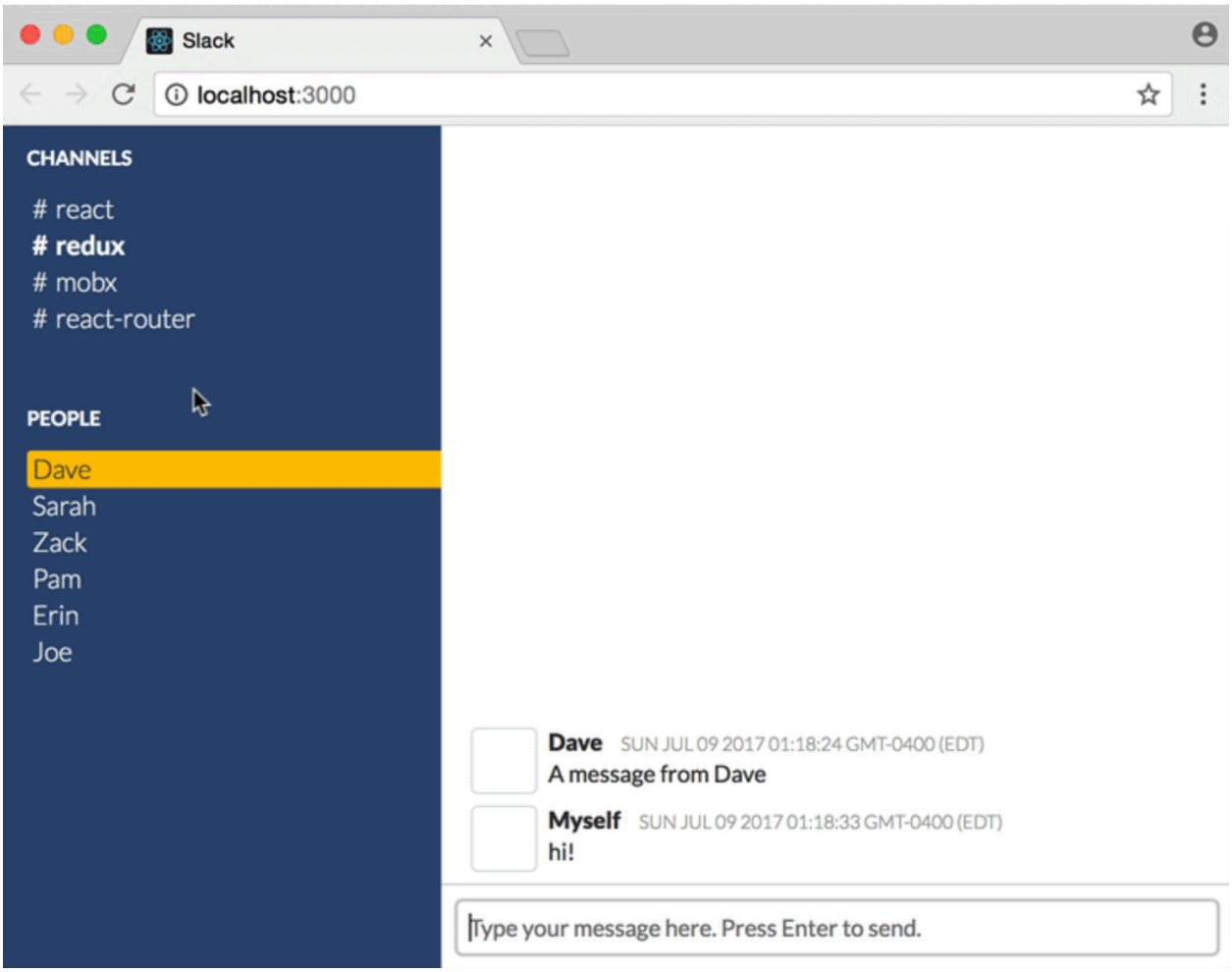
Learning React is tough. There's a lot to learn at once. You might even think of "learning React" as meaning that you have to also learn about Redux, Webpack, React Router, CSS in JS, and a pile of other stuff.

But here's what *I* mean when I talk about learning React: **just React**. All by itself. That's what we're going to cover in this course.

"Wait... is that even useful tho?"

Absolutely it is. You can build quite a bit with plain old React and the tools it gives you: just JSX, props, state, and passing some data around.

Here's a little Slack clone I put together using pure React (and some fake data to make it not look like a barren wasteland):



Neat, right?

"But won't I eventually need Redux and stuff for Real Apps? Fake data isn't gonna cut it."

I hear you. And I'm not gonna lie: you'll probably want to learn all that other stuff eventually. But that's eventually. Don't worry about building your masterpiece right now. Just worry about getting some paint on the canvas. Even if it's just 2 colors, you'll be creating something – which is way more fun than learning "prerequisites" before doing anything fun.

Think back to learning to ride a bike as a kid. Did you bike along a busy highway into town on your first day? Did anyone hand you some bowling pins and say, "Here, learn to juggle at the same time. That's what the pros at the circus do!"

No, you just focused on not falling over.

And you probably had training wheels.

If you can get to the fun stuff quickly, even if it's just a *tiny* amount of fun, it's a lot easier to keep going. So that's what we'll do here: get you those tiny wins with a few tiny projects, and get you through the basics of React.

And when I say you'll *eventually* get to Redux and that other stuff: I'm not talking *months* in the future (aka never). Whenever you understand enough React to feel like "ok, I think I got this," you're ready for more. If you've already got some programming experience, that's probably a matter of days or weeks. If you're starting fresh, it might take a bit longer.

2 Lesson 1: Get something on the screen

First let's get "Hello World" on the screen and then we'll talk about what the code is doing.

1. Start up a blank project on CodeSandbox: go here
2. Hold your breath and delete the entire contents of the index.js file.
3. Type in this code:

```
import React from 'react';
import ReactDOM from 'react-dom';

function Hi() {
  return <div>Hello World!</div>;
}

ReactDOM.render(<Hi/>, document.querySelector('#root'));
```

Now, before we move on.

Did you copy-paste the code above? Or did you type it in?

Seriously though: it's important to actually type this stuff in. Typing it drills the concepts and the syntax into your brain. If you just copy-paste (or read and nod along, or watch videos and nod along), the knowledge won't stick, and you'll be stuck staring at a blank screen like "how does that `import` thing work again?? how do I start a React component??"

Typing in the examples and doing the exercises is the "one weird trick" to learning React. It trains your brain. That brain is gonna understand React one day. Help it out ;)

Alright, let's talk about how this code works.

Imports

At the top, there are 2 import statements. These pull in the 'react' and 'react-dom' libraries.

```
import React from 'react';
import ReactDOM from 'react-dom';
```

With modern ES6 JavaScript (e.g. most React code you'll see), libraries aren't globally available; they need to be imported. This is a change if you're used to `<script>` tags, and at first it might seem like a pain. But explicitly importing things has a really nice side effect: as you read through the code, you can always tell where a variable/class/object comes from.

For instance: See the ReactDOM.render at the bottom of the code? Instead of reading that and going "what the heck is ReactDOM, where does that come from?" you can look up top and see that it's imported from the 'react-dom' library. Nifty. It's not a big benefit now, when the file is 8 lines long, but it will be.

The 'Hi' Component

Right under the imports is a function called `Hi`. This is really truly just a plain JS function. In fact, everything in this file up to and including the word "return" is plain JS syntax, nothing React-specific.

```
function Hi() {
  return <div>Hello World!</div>;
}
```

What makes this function a component is the fact that it returns something that React can render. The `<div>Hello world</div>` is a syntax called JSX, and it looks and works a lot like HTML. React calls the function, gets the JSX, and renders the equivalent HTML to the DOM.

Notice how the JSX is not a string. It's not `return "<div>Hello World</div>";`. React isn't turning these things directly into strings, either.

Before React runs, there's an extra step the code goes through that converts the JSX into function calls. This all happens under the hood (Babel is the tool that does the heavy lifting).

The fact that it's not a string might seem like an unimportant detail, but it's actually pretty cool: you can insert bits of JS code inside the JSX tags, and React will run them dynamically. We'll see that in a minute.

But how does React know where in the DOM to put this div on the page?

Rendering

The last line is what makes it all work. It calls the `Hi` function, gets the returned JSX, and inserts the corresponding HTML elements into the document under the element with id "root". document.querySelector("#root") works similar to jQuery's $("root"), finding and returning that element from the document.

```
ReactDOM.render(<Hi/>, document.querySelector('#root'));
```

Your Turn!

Now that you have a project in place, you can play around :)

Make sure to actually try these exercises. Even if they seem really simple. Even if you are 99% sure you know how to do it, prove it to yourself by typing it out and seeing the working result. Liam Neeson is watching.

- Change the text "Hello World!" to say "Hello !"
- Bold your name by wrapping it in a `<strong>` tag. It works just like HTML.
- Inside the `<div>`, and under your name, add some other HTML elements. Headings, ordered and unordered lists, etc. Get a feel for how it works. How does it handle whitespace? What happens if you forget to close a tag?
- I mentioned that you can put real JS code inside the JSX. Try that out: inside the div, insert a JS expression surrounded with single braces, like `{5 + 10}`.
- Want to style it with CSS? Instead of using the "class" property like you would in HTML, use "className". Then create a file src/index.css with the styles, and add the line `import './index.css'` to the top of index.js. Yep, you can import CSS into JS. Sorta. That's Webpack working its magic behind the scenes.

At this stage, if you already know some HTML and some CSS, you know enough about React to replicate basically any static website!

Cram all the HTML into a single component, import a CSS file, style it up, and hey – you're making web pages like it's 1995. Not too shabby for your first day!

Play around with it and see what you can come up with. Send me a screenshot if you make something cool :)

In the next chapter we'll talk about how to display dynamic data with React components.

P.S. Learning in bite-size chunks is *way* more effective at making the knowledge stick.

Read a bit, then write some code to practice. Do this enough times, and you can master React pretty painlessly.

For a bite-size approach to learning React, check out my book, Pure React.

3 Lesson 2: Make your components dynamic and reusable

Quick Recap: yesterday we got a React app onto the screen, talked a bit about how JSX works a lot like HTML, and did a few tiny exercises.

Let's get into today's lesson: how to make your React components dynamic and reusable.

Before we talk about how to do this in React, let's look at how to do this with plain functions. This might seem a bit basic but bear with me. Let's say you have this function:

```
function greet() {
  return "Hi Dave";
}
```

You can see pretty plainly that it will always return "Hi Dave". What if you want to greet someone else though? You'd pass in their name as an argument:

```
function greet(name) {
  return "Hi " + name;
}
```

Now you can greet anyone you want! Awesome. (I warned you this part was basic)

Using Arguments in a React Component

If you want to customize a React component, the principle is the same as customizing a function: pass an argument with your dynamic stuff, and then the component can change what it returns based on that stuff.

Let's change the `Hi` component from yesterday to be able to say hi to whoever we want. If you still have the CodeSandbox tab open from yesterday, great – if not, start with this one, and code along. Here's the original component:

```
function Hi() {
  return <div>Hello World!</div>;
}
```

Add a parameter called "props", and replace World with {props.name} (which is a JS expression, like we discussed yesterday):

```
function Hi(props) {
  return <div>Hello {props.name}!</div>;
}
```

What's going on here? Well, at first, it just renders "Hello" because we're not passing a name yet. But aside from that…

When React renders a component, it passes the component's props (short for "properties") as the first argument, as an object. The props object is just a plain JS object, where the keys are prop names and the values are, well, the values of those props.

You might then wonder, where do props come from? Glad you asked.

Pass Props to a Component

You, the developer, get to pass props to a component when it is invoked. And, in this app, we're invoking the `Hi` component in the last line:

```
ReactDOM.render(<Hi />, document.querySelector('#root'));
```

We need to pass a prop called "name" with the name of the person we want to greet, like this:

```
ReactDOM.render(<Hi name="Dave"/>, document.querySelector('#root'));
```

With that change in place, the app now displays "Hello Dave!" Awesome!

Here's a cool thing about props: you can pass whatever you want into them. You're not restricted to strings, or trying to guess what it will do with your string (*cough* Angular). Remember yesterday, and 30 seconds ago, how we put a JS expression inside single braces? Well, you can do the same thing with a prop's value:

```
<CustomButton
  green={true}
  width={64}
  options={{ awesome: "yes", disabled: "no" }}
  onClick={doStuffFunc}
/>
```

You can pass booleans, numbers, strings (as we saw), functions, and even objects. The object syntax looks a little weird ("what?? why are there double braces??"), but think of it as single braces surrounding an object literal, and you'll be alright.

All of that syntax, by the way, is React (specifically, JSX). It's not ES6 JavaScript. Which reminds me, I wanted to show you a couple bits of ES6 syntax that will make your components easier to write & read.

A Few Bits of ES6

Most of the components you see in the wild will not take an argument called "props". Instead, they use ES6's destructuring syntax to pull the values out of the props object, which looks like this:

```
function Hi({ name }) {
  return <div>Hello {name}!</div>;
}
```

The only thing that changed here is that the argument props became this object-looking thing { name }. What that says is: "I expect the first argument will be an object. Please extract the 'name' item out of it, and give it to me as a variable called 'name'." This saves you from having to write "props.whatever" all over the place, and makes it clear, right up top, which props this component expects.

One more bit of ES6 syntax I want to show you, and then we're done. (Not to overload you with syntax or anything, but you'll probably see example code like this and I want you to be prepared for it.) This is const and the arrow function:

```
const Hi = ({ name }) => {
  return <div>Hello {name}!</div>;
}
```

The const declares a constant, and the arrow function is everything after the first equal sign.

Compare that code with the "function" version above. Can you see what happened? Here's the transformation step-by-step:

```
// Plain function:
function Hi({ name }) {
  return <div>Hello {name}!</div>;
}

// A constant holding an anonymous function:
const Hi = function({ name }) {
  return <div>Hello {name}!</div>;
}

// Removing the "function" keyword and adding the
// arrow after the parameter list:
const Hi = ({ name }) => {
  return <div>Hello {name}!</div>;
}

// Optional step 3: Removing the braces, which makes the
// "return" implicit so we can remove that too. Leaving the parens
// in for readability:
const Hi = ({ name }) => (
  <div>Hello {name}!</div>
)

// Optional step 4: If the component is really short, you can put
// it all on one line:
const Hi = ({ name }) => <div>Hello {name}!</div>;
```

Your Turn

Now you know how to pass props into a component to make it both dynamic and reusable! Work through these exercises to try out a few new things with props. (Remember: it's important to actually *do* the exercises!)

- Write a new component called `MediaCard` that accepts 3 props: `title`, `body`, and `imageUrl`. Inside the component, render the title in an `<h2>` tag, the body in a `<p>` tag, and pass the imageUrl into an img tag like this: `<img src={imageUrl}/>`. Can you return all 3 of these things at once? Or do you need to wrap them in another element?
- Render the MediaCard with the ReactDOM.render call, and pass in the necessary props. Can you pass a JSX element as a prop value? (hint: wrap it in single braces). Try bolding some parts of the body text without changing the implementation of MediaCard.
- Make a component called Gate that accepts 1 prop called "isOpen". When isOpen is true, make the component render "open", and when isOpen is false, make it render "closed". Hint: you can do conditional logic inside JSX with the ternary (question mark, `?`) operator, inside single braces, like this: {speed > 80 ? "danger!" : "probably fine"} (which evaluates to "danger!" if speed is over 80, and "probably fine" otherwise).

Tomorrow we'll look at how to make your app interactive, with React's state feature.

P.S. Doing little exercises is an awesome way to reinforce new knowledge right away. It makes you remember.

It's very easy to read and read, and *feel* like you understand…

And then when you go to write the code yourself… *POOF* the knowledge is gone.

Exercises to reinforce each major concept are a core principle behind my book, Pure React. One of my readers said, "It put me in a good place for my new job. The exercises were excellent."

4 Lesson 3: Make your app interactive

Today we are **starting fresh**! No longer satisfied by merely saying "hello", we are launching into exciting new uncharted waters: *turning the lights on and off*! I know, it's very exciting.

Over the last couple days you've…

- written a tiny, friendly React app
- customized that app to be able to greet literally anyone in the entire world
- hopefully worked through some of the exercises I slaved over (just kidding!) (but seriously. do the exercises.)
- learned that React isn't so scary, but it's also pretty static so far

In today's lesson we're going to break away from static pages by learning how to use *state*.

Our project will be a page where the user can toggle the lights on and off by clicking a button. And by "the lights" I mean the background color (but hey, if you hook this up to an IoT thing in your house, I totally want to hear about it!).

Create the Project

We're gonna start with a brand new project. Go here: https://codesandbox.io/s/new, erase the contents of index.js, and type this in:

```
import React from 'react';
import ReactDOM from 'react-dom';

function Room() {
  return (
    <div className="room">the room is lit</div>
  );
}

ReactDOM.render(<Room />, document.getElementById('root'));
```

This is nothing you haven't seen before. Just a `Room` component rendering some JSX. But it's awfully bright. Let's turn those lights off.

Make the component stateful

I don't know if you know this about light switches, but they can be in one of two states: ON and OFF. Conveniently React has a thing called state which allows components to keep track of values that can change – a perfect place to store the state of our light.

In order to add state to our `Room` component, it has to be turned into a `class`. In React, function components are stateless. Replace the whole `Room` function with this class instead:

```
class Room extends React.Component {
  state = {
    isLit: true
  }

  render() {
    return (
      <div className="room">the room is lit</div>
    );
  }
}
```

Now that `Room` is a class, the `render` method does the same job as our old Room function. Up top is where the state is initialized. This new syntax is all ES6 JavaScript, by the way – not React-specific. (technically the `state = { ... }` bit is not part of the standard yet, but they're working hard on that).

Render according to state

Right now, the component works the same as before, because we haven't changed anything in `render`. Let's have it render differently based on the state of the light. Change the `<div>` to this:

```
<div className="room">
  the room is {this.state.isLit ? 'lit' : 'dark'}
</div>
```

As you can see, the light is still on. Now change `isLit: true` to `false`. The app will re-render saying "the room is dark."

Ok so this isn't very exciting yet; we've proved that we can change the display by changing the code But we're getting there.

Change the state when you click the button

Let's add a button and kick this thing into high gear. Change the div to look like this:

```
<div className="room">
  the room is {this.state.isLit ? "lit" : "dark"}
  <br />
  <button onClick={this.flipLight}>flip</button>
</div>
```

So we've got a plain old line break with the `<br/>`, and then a button that says "flip". When you click the button, it will call the `this.flipLight` function in the component. And if you've been paying attention, we *don't have* a flipLight function… so let's add that now.

Above the render() function, and under the state initialization, add this code:

```
flipLight = () => {
  this.setState({
    isLit: !this.state.isLit
  });
};
```

Remember how we talked about arrow functions yesterday? This is one of those. Except it's *inside a class*, which makes it a member function.

Click the "flip" button now. Does it work? Hooray! We'll fix the stark white background in a sec, but let's talk about this `setState` thing.

How setState works

In the flipLight function, we're toggling the isLit state true/false depending on what it's currently set to. You might wonder why we don't just say `this.state.isLit = !this.state.isLit`. It's because the setState function actually has 2 jobs:

- first it changes the state
- then it re-renders the component

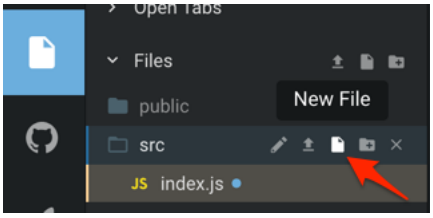
If you just change `this.state` directly, React has no way of knowing that it changed, and it won't re-render. There are more reasons why changing state directly is a bad idea, but for now, just remember to always use `this.setState`, and don't modify `this.state.anything` directly.

We're *alllmost* done here, but I promised that the background color would change, and it did not. Yet.

Change the background color

To do that we're going to use some CSS. Plain old CSS.

In the CodeSandbox editor, hover over the "src" folder on the left and click the "New File" button.



Name it `index.css`, and paste this in. We don't need TOO much style here, but we do need some.

```
html, body, #root, .room {
  height: 100%;
  margin: 0;
}

.lit {
  background-color: white;
  color: black;
}

.dark {
  background-color: black;
  color: white;
}
```

At the top, we are setting the height of absolutely-freaking-everything to 100%, so that the whole page goes dark instead of just the top 20 pixels. Then we have "lit" and "dark", which will change the background and text color accordingly.

Now click over to index.js, and at the top of the file, import the CSS file like so:

```
import "./index.css";
```

Now all that's left is to apply the "lit" and "dark" classes to the Room component. Remember how the

in Room has the prop `className="room"`? We need to change that dynamically to "room lit" or "room dark", depending on the state.

At the top of render, add a variable for the lightedness:

```
const lightedness = this.state.isLit ? "lit" : "dark";
```

Then use that variable, plus ES6's template strings, to change the className on the div. Like this:

```
<div className={`room ${lightedness}`}>
```

The backticks signify a template string in ES6.

Template strings allow you to insert variables within them, which is what the `${lightedness}` is doing.

And finally the whole thing is surrounded by single braces because it's an expression, and because React says so. I'm not sure why React allows you to just pass `someProp="string"` without the braces but requires the braces when it's `someProp={`template string`}`, but it does, so don't forget them.

Click the button now, and you'll see that the background color changes along with the text. Woohoo!

If your code isn't working, you can compare it to the final working example here.

Your Turn

Run through these quick exercises to solidify your understanding:

- Add 2 more buttons: "ON" and "OFF". Add the corresponding functions (similar to `flipLight`) to turn the light either ON or OFF depending on which button is clicked, and wire up the buttons to call those functions.
- Add another piece of state: the room temperature. Initialize it to 72 (or 22 for you Celsius types). Display the current temperature inside render().
- Add 2 more buttons: "+" and "-". Then add corresponding functions that will increase or decrease the temperature by 1 degree when clicked, and wire up the buttons.

P.S. State is one of the trickier concepts to grok in React.

I put together a Visual Guide to State in React if the concept still seems a bit fuzzy to you.

My Pure React book dedicates a couple chapters to handling state, including a shopping cart that we build step-by-step.

5 Lesson 4: Fetch data from a server

Hopefully you've had time to follow along and do the exercises from the last few days. And if I've done my job right, you're starting to understand how React works – maybe even thinking about how to use it in your next project.

But there's one glaring question, and it's one of the first things React newcomers ask:

How do you get data from an API?

A fancy front end is no good without data! So today we're going to tackle that head-on by fetching some real data from Reddit and displaying it with React.

How to Fetch Data

Before we dive in, there's one thing you need to know: React itself doesn't have any allegiance to any particular way of fetching data. In fact, as far as React is concerned, it doesn't even know there's a "server" in the picture at all. React is UI-only, baby.

You've already learned what makes React tick: it's props and state. There is no HTTP library built into React. So the way to make this work is to use React's lifecycle methods to fetch data at the appropriate time, and store the response in the component's state. You can complicate this process with services and data models (er, "build abstractions") as much as you desire, but ultimately it all boils down to components rendering props and state.

To fetch data from the server, we'll need an HTTP library. There are a ton of them out there. Fetch, Axios, and Superagent are probably the 3 most popular – and Fetch is actually part of the JS standard library now. My favorite is Axios because of how simple it is, so that's what we'll use today. If you already know and prefer another one, go ahead and use that instead.

Create the Project

Once again we're going to start with a fresh project in CodeSandbox.

1. Go to https://codesandbox.io/s/new
2. Erase everything in index.js
3. Replace it with this:

```
import React from "react";
import ReactDOM from "react-dom";

class Reddit extends React.Component {
  state = {
    posts: []
  }

  render() {
    return (
      <div>
        <h1>/r/reactjs</h1>
      </div>
    );
  }
}

ReactDOM.render(<Reddit />, document.getElementById("root"));
```

We're creating a class component called `Reddit` to display posts from the /r/reactjs subreddit. It's not fetching anything yet, though.

By now the code probably looks familiar – the imports at the top and the render at the bottom are the same as we've written the last few days. We're using a class component because it needs to have state to hold the posts that we fetch from the server (and function components can't hold state). We're also initializing the state with an empty array of `posts`, which will soon be replaced by live data.

Render the List

Next, let's add some code to render the posts we have so far (which is an empty array). Change the `render` function to look like this:

```
render() {
  return (
    <div>
      <h1>/r/reactjs</h1>
```

```
      <ul>
      {this.state.posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
      </ul>
    </div>
  );
}
```

Let's talk about what this is doing, because it probably looks a bit weird.

Inside the `<ul>` the JS expression is wrapped in single braces (because, remember, that's what you gotta do in JSX). `this.state.posts` is the empty array of posts we initialized above, and the `map` function loops over the posts and returns an `<li>` for each item in the array.

This is how you render a list in React.

Other libraries have a special template syntax, like Angular's "ngFor" or Vue's "v-for", that essentially make a copy of the element for each item in the array.

React, on the other hand, leans on JavaScript to do the heavy lifting. "this.state.posts.map" is not a React thing. That's calling the `map` function that already exists on JS arrays, and it transforms (a.k.a, "maps") each item in the array into a new thing. In this case, each array item is being turned into a JSX `<li>` element with a `key` prop and the post's title, and the resulting array of `<li>`'s is what gets rendered inside the `<ul>`.

Fetch the Data

Let's see it in action with real data. First we'll import Axios, so add this new import at the top:

```
import axios from 'axios';
```

Because we haven't installed the library yet, CodeSandbox will throw up an error saying "Could not find dependency: 'axios' " but it also provides a nice "Suggested solution" in the form of a button that says "Add axios as a dependency." Click that button, and the library will be added to the project, and the app will refresh.

Then, above the render() method, type in this code:

```
componentDidMount() {
  axios.get(`https://www.reddit.com/r/reactjs.json`)
    .then(res => {
      const posts = res.data.data.children.map(obj => obj.data);
      this.setState({ posts });
    });
}
```

We're using `axios.get` to fetch the data from Reddit's API, which returns a promise, and the `.then` handler will get called once the fetch is finished.

Reddit's API returns the posts in a pretty deeply-nested structure, so the `res.data.data.children.map...` is picking out the individual posts from the nested mess.

Finally, it updates the posts in the component state (remember that `this.setState({ posts })` is just ES6 shorthand for `this.setState({ posts: posts })`)

If you want to learn how to reverse-engineer API responses yourself, go to https://www.reddit.com/r/reactjs.json and copy the result into a pretty-printer like http://jsonprettyprint.com/, then look for something you recognize like a "title", and trace back upwards to figure out the JSON path to it. In this case, I looked for the "title", and then discovered posts were at `res.data.data.children[1..n].data`.

Once that change is made, you'll see the app re-render with real data from /r/reactjs!

Your Turn

Take a few seconds and run through the quick exercises below. It'll help the concepts stick!

- Did you notice that the first thing we did was initialize the state with an empty array of posts? Before we even fetched the data? Try taking out the initialization and watch what happens. Burn the resulting error message into your memory: when you see this again, remember to check if you've initialized state properly, and check for typos.
- Extend the UI to include more data from the Reddit posts, like their score and the submitting user. Make the title into a link to the actual post.

Tomorrow, we'll finally be free of the sandbox, and we'll finish up by deploying the app to the real live internet.

P.S. The fastest way to wrap your head around the "React way" of doing things (like rendering lists) is to practice.

It's the best method I know. But you need focused exercises.

My Pure React book comes with 26 exercises to hammer home everything you learn.

6 Lesson 5: Out of the sandbox and into production

Up until now we've been using CodeSandbox to write our projects. It's a great tool, easy to use… but sometimes writing apps in a browser feels a little… *fake*.

It's a bit more fun (and useful in the real world) if you can develop on your *own* machine. And even better if you can deploy that app to a server.

Today, we're doing both of those.

Local Development

For local React development, you need to have a few tools installed:

- Node.js and NPM: https://nodejs.org/
- Optionally, Yarn: https://yarnpkg.com/
- A text editor (I like Vim, and VSCode is nice too)
- Create React App

The last one is not strictly necessary to write React apps, but it's an awesome tool that makes for a nice development experience (DX) and it hides away all the complexity of setting up Webpack and Babel. And: it's not just for toy examples. It includes a real production build, which we'll use later today.

Install Create React App by running this from a command line:

```
npm install -g create-react-app
```

Create the Project

And then create an app for our example by running this command:

```
create-react-app reddit-live
```

When you run that, CRA will install a bunch of dependencies, and then give you further instructions. Follow what it says: switch into the new directory and start up the app.

```
cd reddit-live
npm start
```

Open up the project in your favorite editor. You'll see that CRA generated a few files for us already. There's the `src/index.js` file, similar to what we've had from CodeSandbox. And then there's an `App` component in `src/App.js`, along with tests, and some styling.

For this example we're going to ignore everything other than index.js, though.

Just as we've done before, open up index.js and delete everything within. Replace it with this code, which should look familiar from yesterday's lesson:

```
import React from "react";
import ReactDOM from "react-dom";
import axios from "axios";

class Reddit extends React.Component {
  state = {
    posts: []
  };

  componentDidMount() {
    axios.get(`https://www.reddit.com/r/reactjs.json`).then(res => {
      const posts = res.data.data.children.map(obj => obj.data);
      this.setState({ posts });
    });
  }

  render() {
    return (
      <div>
        <h1>/r/reactjs</h1>
        <ul>
          {this.state.posts.map(post => {
            return <li key={post.id}>{post.title}</li>;
          })}
        </ul>
```

```
      </div>
    );
  }
}

ReactDOM.render(<Reddit />, document.getElementById("root"));
```

The app will automatically recompile when you save, and you'll see an error because we're importing `axios` but we haven't installed it.

Back at the command line, install axios by running:

```
npm install axios --save
```

CRA should automatically pick this up if you left it running (if you didn't, start it back up now).

The app should be working if you visit http://localhost:3000/.

Deploy to Production

Now let's push it up to a server! To do that, we'll use surge.sh.

I made a 27-second video of this process if you want to see it live.

First we need to install the `surge` command:

```
npm install -g surge
```

Surge will deploy the directory we run it from, and our react-live project isn't yet ready for deployment. So let's build the project using CRA's built-in production build. Run this:

```
npm run build
```

This will output a production-ready app inside the "build" directory, so switch into that directory and run surge:

```
cd build
surge
```

It'll ask you to create an account, then prompt you for a directory (which defaults to the current one), and then a hostname (make it whatever you like), and then Surge will upload the files.

Once it finishes, visit your running site! Mine is http://crabby-cry.surge.sh (the names it comes up with are great).

That's all there is to it! You can also pair Create React App projects with other popular hosting providers like Netlify, Heroku, and Now.

I hope you had fun with this 5 day intro to React! I had fun writing it.

If you want to get *yourself* production ready on React knowledge, check out my book Pure React – you'll learn to bend JSX to your will, master the "React way" of doing things, and get up to speed on ES6 features like classes, rest & spread operators, and destructuring. One of my students, Mara, said:

> "I came to know roughly what components I need to build just by looking at the design and imagining how data flows from the root component to the children components. I would recommend the book to everyone that already had a course for beginners but they want to deepen their knowledge and really understand React by building basic UIs.
>
> Basically everyone that wants to get paid by building apps with React."

You can buy Pure React here.

Cheers, Dave