COMPLETE
REACT TUTORIAL
FOR 2020

why? • props • state • Hooks • API calls • deployment

Dave Ceddia

daveceddia.com

Contents

1 Introduction

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

This article 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.

Here's what we'll cover:

1. What React is and why it's worth learning
2. How to get something on the page with components and JSX
3. Passing data to components with props
4. Making things interactive with state and hooks
5. Calling an API with useEffect
6. Deploying a standalone React app

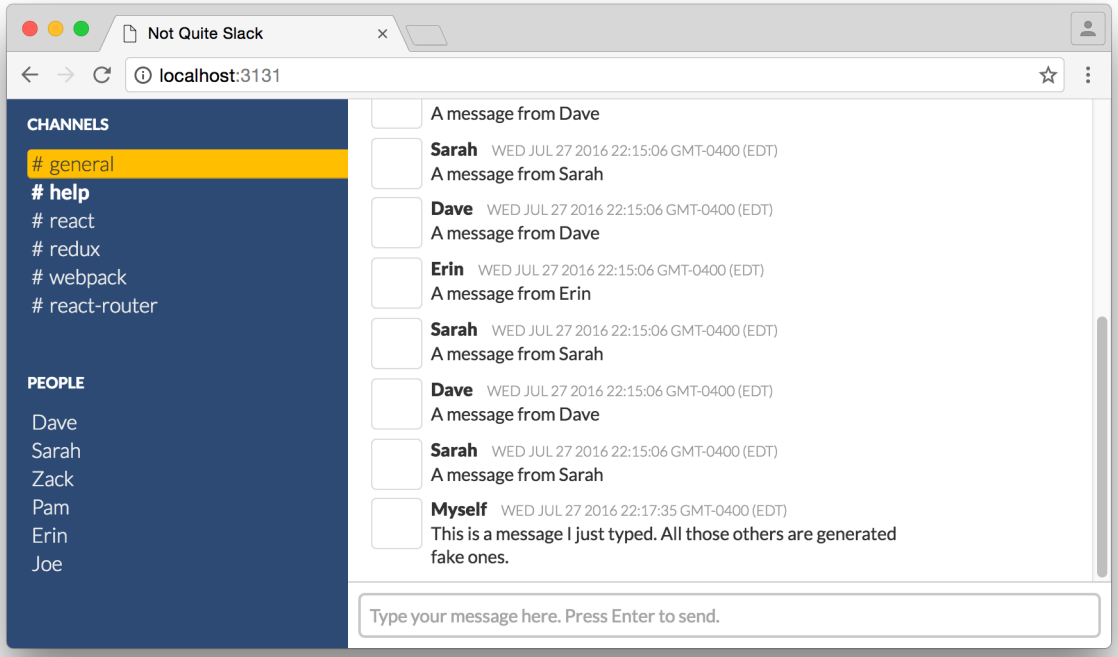
Quick warning though: this tutorial is *compleeete*. And by that I mean *looong*. I turned this into a full-fledged free 5-day course, and I made a nice-looking PDF you can read on your iPad or [whatever Android device is cool these days]. Drop your email in the box to get both. (it's spread over 5 days, but you can jump ahead whenever you want).

I want to clarify what *I* mean when I talk about learning React: **just vanilla React**. All by itself. That's what we're going to cover here.

"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 vanilla React (and some fake data to make it not look like a barren wasteland):



Neat, huh?

"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.

Tiny Wins Propel You Forward

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.

We're going to get you some **tiny wins** with a few little 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.

What is React?

React is a UI library created by Facebook. It helps you create interactive web applications made up of **components**. If you're familiar with HTML, you can think of components as fancy custom tags. That's pretty much what they are: reusable bits of content and behavior that can be put on a web page.

A **component** is written as a plain JavaScript function. And this is real JavaScript here, not a template language. React supports a special syntax called JSX, which looks a lot like HTML, but it is turned into real JavaScript code by a compiler.

A web page is made up of components laid out in a nested "tree" structure. Just like HTML elements can contain other elements, React components can contain other components (and native elements like `divs` and `buttons`). But these components are functions, remember, and they can be passed data to display.

One of the defining features of React is the idea of **one-way data flow**, and this was a big part of what set React apart when it first came out in 2013. These days, lots of other libraries (like Vue, Svelte, and Angular) have adopted the one-way data flow pattern too.

In the one-way data flow model, data is only ever passed *down* the tree, from a component to its children. Just like a waterfall: only down, not up or sideways. Unlike in some other approaches (like jQuery) where data might've been globally available and you could "plug it in" anywhere on the page, React is more explicit and more restrictive. With React you'd pass that data into the top-level component, and that one would pass it down, and so on.

React is a great way of building interactive UIs, and it's very popular right now (and has been for a few years). If you are looking to get into a career as a front end developer, React is an excellent library to learn. React developers are in high demand.

It's not the only way to build UIs though! Plenty of other libraries exist. Vue.js and Angular are the most popular alternatives, and Svelte is gaining steam. And, even now in 2020, you can still build static pages with plain HTML and CSS, and dynamic pages with plain JavaScript.

What Should You Know Before Learning React?

Learning "prerequisites" is boring, but React builds upon ideas from HTML and JavaScript. While I don't think it's important to run off and master HTML and JS for months before getting into React, it'll be a big help to have some experience with them first.

React is a library on top of JS, and unlike a lot of other libraries that are just a set of functions, React has its own "JSX" syntax that gets mixed in. Without a solid grasp of JS syntax, it can be hard to tell which parts of code are "React things" and which are Just JavaScript. Ultimately it turns into a jumbled mess in your head, and it's harder to know what to Google. React will be much easier if you learn plain JavaScript first.

And, since JSX is heavily inspired by HTML (with the you-must-close-every-tag strictness of XML), it'll be a big help to understand HTML.

React has no "preferred" way to do styling. Regular CSS works great (you'll see how to use it later in this post!), and there are a bunch of CSS-in-JS libraries if you want to go that route (styled-components is probably the most popular). Either way, you need to understand how CSS works to effectively style pages.

An awful lot of "how to do X in React" questions are actually JavaScript or HTML or CSS questions, but you can't know where those lines are without understanding the other tech too.

This tutorial will walk you through the basics of React, and I think you'll be able to get something out of it even if you're not too familiar with JS, HTML, and CSS.

2 Hello World!

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?

Cuz 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 fingers. Those fingers are gonna understand React one day. Help 'em 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 (which is 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 the days of jQuery, 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 example? 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 pretty obvious where a thing comes from when the file is tiny, but explicit imports are excellent when you're working in larger files or ones that you didn't author.

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. `<div>Hello World!</div>` becomes `React.createElement('div', null, 'Hello World!')`.

This all happens behind the scenes. Babel is the tool that does the transformation, and in the stock Create React App config, Webpack is what kicks off Babel).

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 instructs React to call 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'));
```

Babel will compile this down to code that looks like this:

```
ReactDOM.render(
  React.createElement(Hi),
  document.querySelector('#root')
);
```

I wanted to show you this for two important reasons:

1. **This code is pure JavaScript.** At the end of the day, the fancy JSX syntax turns into regular JS. You might've seen people refer to React as "just JavaScript" before: this is what they mean. Outside of the JSX syntax, there's not much "React specific" code in React apps. This is great because it means there's *no magic*. Learning React is more about learning new ways to solve problems than it is about syntax.
2. **You don't call your own components; React does.** Notice how we're passing the `Hi` function itself (aka a reference to that function) into `React.createElement`. We're not *calling* `Hi()`, we're just passing it. This is a subtle thing but important to keep in mind: *React* is responsible for calling your component function. It'll do that at render time – in other words, somewhere in the depths of the `ReactDOM.render` function, and not inside `React.createElement`.

This idea that React is responsible for calling your components means that it is able to run some setup/teardown code before and after. You'll see why that matters when we talk about Hooks in a bit.

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.

- Change the text "Hello World!" to say "Hello <your name>!"
- 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. Leave a comment with a link to your project if you make something cool :)

Learning in bite-size chunks like this is *way* more effective at making the knowledge stick (versus learning *everything* and trying to remember it all)

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

Next, we'll talk about how to display dynamic data with React components.

3 Dynamic and Reusable Components

Let's take the next step and learn how to make your React components dynamic and reusable.

Before we talk about how to do this in React, though, let's look at how to do this with *plain JavaScript*. 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"*.

And if you wanted to greet someone else? You'd pass in their name as an *argument*:

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

Now you can greet anyone you want just by calling `greet` with their name! Awesome. (I warned you this part was basic)

Using Props as Arguments to a React Component

If you want to customize a React component in the same way, the principle is the same: pass an argument with your dynamic stuff, and then the component can do what it wants with that stuff.

Let's change the `Hi` component from earlier to be able to say hi to whoever we want. If you still have the CodeSandbox tab open, 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}`:

```
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? How do we pass them in? Good question.

Pass Props to a Component

You, the developer, get to pass props to a component when it is rendered. And, in this app, we're rendering 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!

Passing props is very similar to setting attributes on an HTML tag. A lot of JSX syntax is borrowed from HTML.

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* Angular1). Remember earlier, 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.

Not Everything is a String

This is important to note: don't pass every prop as a string surrounded by quotes. Sometimes that's what you want, but not always. Be intentional about what type you're passing in, because it will come out as the same type!

For instance, one of these passes the string "false" while the other passes the boolean value `false`:

```
<Sidebar hidden="false"/>
<Sidebar hidden={false}/>
```

These will have very different effects, since the *string* "false" will be interpreted as truthy – probably not what you want here.

This is a difference from HTML (where everything is a string) and some other frameworks like Angular and Svelte that will convert the string into a native JS value. React doesn't parse props at all – they get passed through untouched, just like function arguments.

Receiving Props

Inside a component that receives multiple props, each one will be a separate property on the "props" object that's passed in. For example if we had a component called "HiFullName" that took two props, like this:

```
<HiFullName firstName="Dave" lastName="Ceddia" />
```

Then the internals of that component might look something like this:

```
function HiFullName(props) {
  return (
    <div>Hi {props.firstName} {props.lastName}!</div>
  );
}
```

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 }`. This is JavaScript's destructuring syntax (added in ES6), not a React thing.

What that says is: "I expect the first argument will be an object. Please extract the 'name' property 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. Useful for documentation purposes.

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, one change at a time:

```
// 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>;
}

// Turning the "function" into an arrow:
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, and skip the parentheses:
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 `<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).

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

It's very easy to keep on reading, feeling like it's all crystal clear…

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

Make sure to spend some time practicing the stuff you learn!

Next we'll look at how to make your app interactive, with *state*.

4 Using State in React Components

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

So far we'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 this next section 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 a New 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 or OFF. Conveniently React has a feature 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, we can either turn it into a **class component**, or add state directly to the function with **hooks**. We'll call React's `useState` hook to create a piece of state to hold the lightswitch value:

```
function Room() {
  const [isLit, setLit] = React.useState(true);

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

We've added the call to React.useState at the top, passing in the initial state of **true**. It returns an array with two entries - the first is the state value itself (so, `isLit` will be true) and the second is a function to change the state.

We're using the *array destructuring* syntax `[isLit, setLit]` to break out those two entries and give them names. This syntax is ES6 JavaScript, by the way – not React-specific. It's exactly the same as if we'd written it out like this:

```
const state = React.useState(true);
const isLit = state[0];
const setLit = state[1];
```

The destructuring syntax is just shorter.

You Can Also Import useState

Quick aside! You'll often see useState imported directly from React like this:

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

After doing that, you can call `useState` directly (no need to qualify it with `React.useState`):

```
const [isLit, setLit] = useState(true);
```

Either way is fine, but it's more common to import it. I did it the other way to show it works both ways, and to avoid interrupting the example by having you change the existing `import`, and now I've gone and added a whole section interrupting you about it. Anyway. Onward!

How useState Works

The way we're using `React.useState` might look a bit weird.

If you've been coding outside of React for any length of time, you likely know about variable scope: a variable declared at the top of a function will be wiped out, erased, forgotten as soon as that function returns.

So… how is React able to remember the state in-between calls to the component?

Before React calls your component, it sets up an array to keep track of which hooks get called. When you call `useState` in the component, React uses that array behind the scenes to keep track of its initial value, and the value as it changes over time.

Remember I mentioned earlier that React is responsible for calling your component at render time? And that that means React can do some setup work beforehand? Keeping track of hooks is part of that work.

Your component is the puppet and React is the puppeteer, pulling the strings behind the scenes to make it work.

Read more about how useState works here.

Render Based on State

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

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

As you can see, the light is still on. Now change `React.useState(true)` to `React.useState(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 {isLit ? "lit" : "dark"}
  <br />
  <button onClick={() => setLit(!isLit)}>
    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 its onClick prop.

Remember how we talked about arrow functions earlier? This is one of those. It's embedded right there, anonymously, and passed straight into onClick. That anonymous function calls setLit with a new value.

FYI!! Be careful when you pass functions to props that you don't inadvertently call the function at render time:

```
  // Wrapping with an arrow function
  // delays execution until the click
  // ✅
  <button onClick={() => setLit(!isLit)}>
    flip
  </button>

  // Unwrapped call to setLit will happen
  // before the button is even rendered!
  // 🚫
  <button onClick={setLit(!isLit)}>
    flip
  </button>
```

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 `setLit` function.

How setLit works

In the onClick 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 write `isLit = !isLit`. It's because the setLit function actually has 2 jobs:

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

If you just change the variable directly, React has no way of knowing that it changed, and it won't re-render. Remember that isLit is a regular old variable – not a special React thing! It will go out of scope at the end of the function and any changes to it would be lost.

So that's why it's important to call the setter, so that React can update the value of that hook's state behind the scenes.

Read my intro to Hooks for more about what hooks are and how they work.

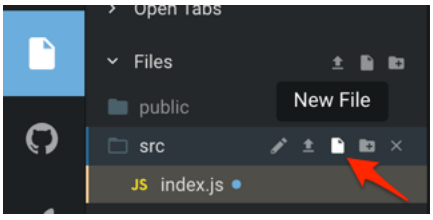
We're *alllmost* done here, but I promised that the background color would change, and it has not. Yet.

Change the background color

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

Not CSS-in-JS, not some special React CSS (not a thing), just 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 the class selectors `.lit` and `.dark`, which will set the background and text color when those CSS classes are applied.

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

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

(We can do this because of Webpack: it will see this import and take it to mean "index.js depends on index.css" and will add the CSS to the page in the correct spot)

Now all that's left is to apply the "lit" and "dark" classes to the Room component. Remember how the `<div>` in `Room` has the prop `className="room"`? We need to change that dynamically to either `room lit` or `room dark`, depending on the state.

At the top of our component, after state is created, we'll create a variable for the lightedness:

```
const brightness = isLit ? "lit" : "dark";
```

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

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

The backticks signify a template string in ES6.

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

And finally the whole string 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.

And one more thing: we could've written the className like this, all inline, without the extra variable:

```
<div className={`room ${isLit ? "lit" : "dark"}`}>
```

That would work the same (try it out!). I created the variable just to make it easier to read.

Your Turn

Run through these quick exercises to solidify your understanding:

- Add 2 more buttons: "ON" and "OFF". Turn the light either ON or OFF depending on which button is clicked by wiring up the onClick handlers to set the state.
- Add another piece of state: the room temperature. (Hint: you can call useState more than once!) Initialize it to 72 (or 22 for you Celsius types). Display the current temperature under the light status.
- Add 2 more buttons: "+" and "-". Then add onClick handlers that will increase or decrease the temperature by 1 degree when clicked,

State is one of the trickier concepts to grok in React. Don't worry if it doesn't click immediately. I put together a Visual Guide to State in React if the concept still seems a bit fuzzy (and in that post, you'll see how to represent state with **class components** instead of hooks – both valid ways to write components).

5 Fetch Data with React

So you've learned how to get something on the page with React. You've learned about **props** to pass data into components, and **state** to keep track of data within a component.

But there's one glaring question, and it's often 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 next 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 the core parts that make React tick: it's only **props** and **state**. There is no HTTP library built into React.

So the way to make this work is to either use React's *lifecycle methods* to fetch data at the appropriate time (in class components), or use the **useEffect** hook to kick off fetching data in a function component.

To a React component, fetching something from a server is a side effect. It's saying "After I'm done rendering, I'll kick off a call to get some data."

Once that data comes back it needs to go into state, and then you can render it from there.

You can complicate this process with services and data models and redux-thunk and sagas (er, "build abstractions") as much as you desire, but ultimately it all boils down to components rendering props and state.

Pick an HTTP Library

To fetch data from the server, we'll need an HTTP library. There are a ton of them out there. Fetch and Axios are probably the most popular – and Fetch is actually part of the JS standard library these days. My favorite is Axios because of how simple it is, so that's what we'll use today. It'll also let us see how to add a library and import it.

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";

function Reddit() {
  const [posts, setPosts] = React.useState([]);
  return (
    <div>
      <h1>/r/reactjs</h1>
    </div>
  );
}

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

We're creating a 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 component function at the bottom are the same as we've written the last few days. We're using the *useState hook* to create a piece of state to hold an array of Reddit posts. We're also initializing the state with an empty array, 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 component to look like this:

```
function Reddit() {
  const [posts, setPosts] = React.useState([]);

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

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

Inside the `<ul>` a JS expression is wrapped in single braces (because, remember, that's what you gotta do in JSX). `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. (more on that here)

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. `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>`.

To look at it another way, if you were to "unroll" the call to `.map` it would look like this (assuming `posts` was an array of these 3 reddit posts):

```
return (
  <div>
    <h1>/r/reactjs</h1>
    <ul>
      {[
          <li key={1}>Post one</li>,
          <li key={2}>Post two</li>,
          <li key={3}>Post three</li>
      ]}
    </ul>
  </div>
);
```

The three posts become three `<li>`'s, wrapped in a new array. React knows how to render arrays of elements as long as they each have a unique `key` prop. Not that you'd ever write code like this, but it shows what the `map` is actually doing.

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 `return` but after the call to useState, type in this code:

```
React.useEffect(() => {
  axios.get(`https://www.reddit.com/r/reactjs.json`)
    .then(res => {
      const newPosts = res.data.data.children
```

```
        .map(obj => obj.data);

      setPosts(newPosts);
    });
}, []);
```

We're using a new hook here, the **useEffect hook**. The way it works is, you pass it a function, and useEffect "queues up" that function to run after render is done.

This particular effect calls `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` state by calling setPosts.

useEffect Dependency Array

One other thing you might've noticed: we're passing an empty array `[]` as the second argument to useEffect. This is the list of variables that this effect *depends on*.

If we don't pass the array at all, then the effect will run on *every* render. As in… it'll re-run the effect after we call setPosts. As in… it will re-fetch the posts every time it renders. Infinitely. Try opening up the browser devtools, look at the Network tab, and take off that empty array argument. See what happens :)

The `useEffect` hook will only queue up the effect another time if something in this array changes, and, since the array is empty, this effect will only run ONCE after the component renders the first time.

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`. I inched up on that solution by writing a parser that printed out `res.data`, and then `res.data.data`, and so on… one level at a time, building up the final chunk of code to parse the response.

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

If you had trouble getting it working, the full working example is here: https://codesandbox.io/s/84x08lwl09 (but hey! don't just skip the work and click the working code example – the knowledge really sticks better when you write it out yourself).

Your Turn

Take a few seconds and try this out. It'll help the concepts stick!

- 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.

The fastest way to wrap your head around the "React way" of doing things (like rendering lists, or modal dialogs, or styling, or whatever) is to practice.

It's the best method I know. But you need some exercises! Here's a list of practice projects to get your mind going.

6 Deploy Your React Project

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.

So let's cover both of those now.

Local Development

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

- Node.js and NPM
- A text editor (I like Vim and VSCode)
- 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 learning. It includes a real production build, which we'll see how to use in a minute.

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, which is 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 inside. Replace it with this code, which should look familiar:

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

```
import axios from "axios";

function Reddit() {
  const [posts, setPosts] = useState([]);

React.useEffect(() => {
  axios.get(`https://www.reddit.com/r/reactjs.json`)
    .then(res => {
      const newPosts = res.data.data.children
        .map(obj => obj.data);

      setPosts(newPosts);
    });
}, []);

  return (
    <div>
      <h1>/r/reactjs</h1>
      <ul>
        {posts.map(post => (
          <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. Better yet though, try it yourself!

First we need to install the `surge` tool:

```
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 at 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 intro to React!

If you want to get *yourself* production ready on React knowledge, check out my Pure React workshop – you'll learn to "think in React", 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 it 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."