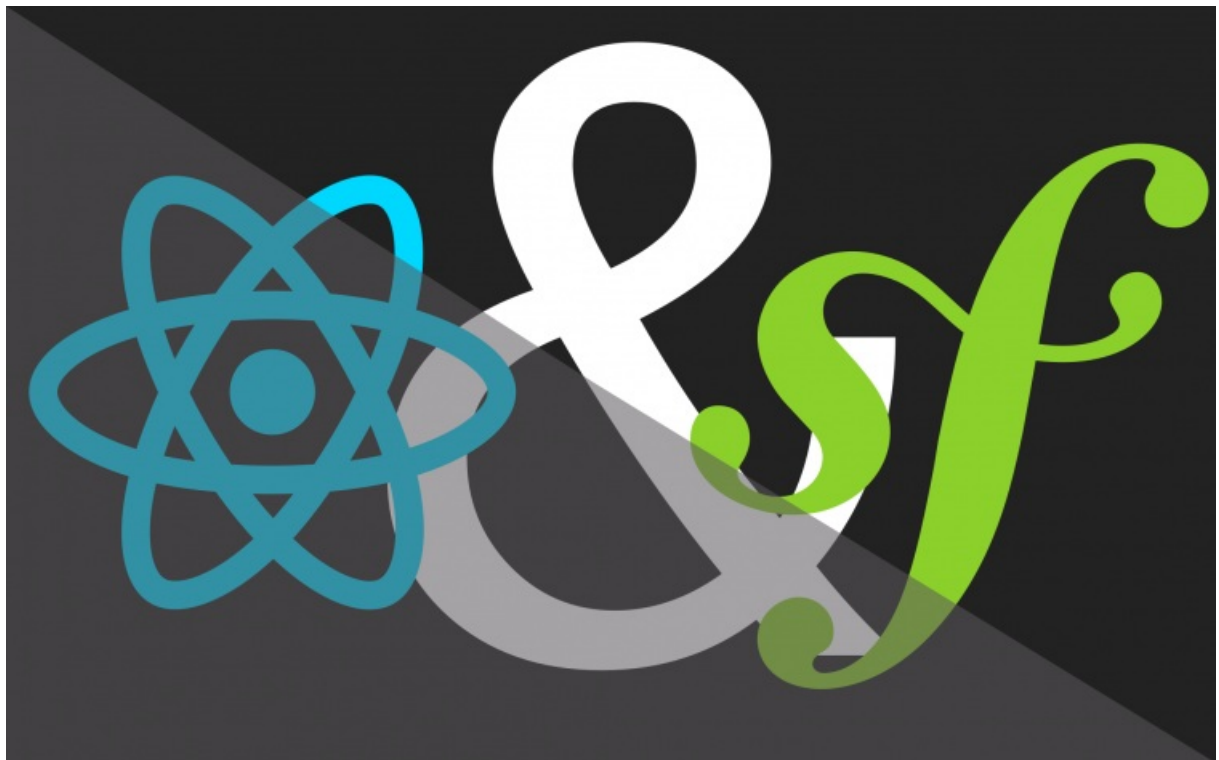


# JavaScript for PHP Geeks: ReactJS (with Symfony)



With <3 from SymfonyCasts

# Chapter 1: The World of React + ESLint

Hey friends! And welcome! Oh, I am *so* excited to talk about React. Developing in React *feels* great, and it's powerful! You can build *any* crazy frontend you want. But, honestly, writing this tutorial was a *huge* pain. Technically speaking, React is not hard. But, to *even* get started, you need to be comfortable with ES6 features *and* you need a build system, like Webpack Encore. That's why we covered both of those topics in previous tutorials.

But even then! The best practices around React are basically non-existent, especially for a backend developer, who instead of building a single page app, may just want to use React to power part of their frontend.


So our goal in this tutorial is clear: to *master* React, but *also* learn repeatable patterns you can follow to write high-quality code *while* getting your wonderful new app finished, and out the door. We won't hide anything: we'll attack the ugliest stuff and, as always, build something *real*.

Excited? Me too! And, a *huge* thanks to my co-author on this tutorial Frank de Jonge, who helped me navigate many of these important topics.

## Project Setup

If you'd like free high-fives from Frank... *or* if you want to get the most out of this tutorial, you should *totally* code along with me. Download the course code from this page. When you unzip it, you'll find a `start/` directory inside that holds the same code that you see here. Open up the `README.md` file for winning lottery numbers and instructions on how to get the project setup.


The last steps will be to open a terminal, move into the project, and run:



```
$ php bin/console server:run
```

to start the built-in web server. Our project already uses Webpack Encore to compile its CSS and JS files. If you're new to Webpack or Encore, go watch our tutorial on that first.

To build those assets, pour some coffee, open a *second* terminal tab, and run:



```
$ yarn install
```

to download our Node dependencies. And... once that finishes:



```
$ yarn run encore dev --watch
```

That will *build* our assets, and *rebuild* when we change files.

Ok cool! Let's go check out the app: find your browser, go to <http://localhost:8000> and say hello to the Lift Stuff App! Login with user `ron_furgandy` password `pumpup`.

In our effort to stay in shape... while sitting down and coding all day... we've built Lift Stuff: an app where we can record *all* the stuff we've lifted throughout the day. For example, before I started recording, I lifted my big fat cat 10 times... so let's totally log that!

In the previous tutorials, we built this JavaScript frontend using plain JavaScript and jQuery. In this tutorial, we'll re-build it with React.

## Installing ESLint

But before we dive into React, I want to install another library that will make life *much* more interesting. Move back to your terminal, open a *third* terminal tab - we're getting greedy - and run:

```
$ yarn add eslint --dev
```

ESLint is a library that can detect coding standard violations in your JavaScript. We have similar tools in PHP, like PHP-CS-Fixer. To configure exactly *which* coding standard rules we want to follow, back in our editor, create a new file at the root of the project: `.eslintrc.js`.

I'll paste in some basic configuration here: you can copy this from the code block on this page. We won't talk about ESLint in detail, but this basically imports the ESLint recommended settings with a couple of tweaks. This `jsx` part is something we'll see *very* soon in React.

```
20 lines | .eslintrc.js
1  module.exports = {
2    extends: ['eslint:recommended'],
3    parserOptions: {
4      ecmaVersion: 6,
5      sourceType: 'module',
6      ecmaFeatures: {
7        jsx: true
8      }
9    },
10   env: {
11     browser: true,
12     es6: true,
13     node: true
14   },
15   rules: {
16     "no-console": 0,
17     "no-unused-vars": 0
18   }
19 };
```

Thanks to this, we can now run a utility to check our code:

```
$ ./node_modules/.bin/eslint assets
```

where `assets/` is the directory that holds our existing JavaScript code. And... aw, boring! It looks like *all* of our code already follows the rules.

This utility is nice... but there's a more important reason we installed it. In PhpStorm, open the settings and search for `eslint` to find an ESLint section. Click to Enable this and hit Ok. Yep, PhpStorm will now *instantly* tell us when we've written code that violates our rules.

Check this out: open `assets/js/rep_log.js`: this is the file that runs our existing LiftStuff frontend. Here, add `const foo = true` then `if (foo)`, but leave the body of the `if` statement empty. See that little red error? That comes from ESLint.

This may not *seem* very important, but it's going to be *super* helpful with React.

## [Adding a new Entry](#)

As I mentioned, our app is already built in normal JavaScript. Instead of deleting our old code immediately, let's leave it here and build our React version right next to it. In the same directory as `rep_log.js`, which holds the old code, create a new file: `rep_log_react.js`. Log a top-secret, important message inside so that we can see if it's working. Don't forget the Emoji!

```
2 lines | assets/js/rep_log_react.js
1  console.log('Oh hallo React peeps! ? ');
```

Now, open `webpack.config.js`: we're going to configure this as a new "entry". Typically, you have one entry file per page, and that file holds *all* of the JavaScript you need for that page. Use `addEntry('rep_log_react')` pointing to that file: `./assets/js/rep_log_react.js`.

```
33 lines | webpack.config.js
... lines 1 - 3
4  Encore
... lines 5 - 12
13  .addEntry('rep_log_react', './assets/js/rep_log_react.js')
... lines 14 - 28
29  ;
... lines 30 - 33
```

To build this, go back to your terminal, find the tab that is running Webpack Encore, press `Ctrl+C` to stop it, and run it again: you need to restart Webpack whenever you change its config file.

Finally, to add the new JavaScript file to our page, open `templates/lift/index.html.twig`, find the `javascripts` block, and add the script tag for `rep_log_react.js`. You don't normally want *two* entry files on the same page like this. But when we finish, I plan to delete the old `rep_log.js` file.

And just like that, we can find our browser, open the dev tools, go to the console, refresh and... Hello World!

Now, it's time to go say Hello to React!

# Chapter 2: React.createElement()

Time to install React! Find your open terminal: `yarn add react` - and we also need a second package called `react-dom`:

```
$ yarn add react react-dom --dev
```

react is... um... React. `react-dom` is the library we'll use to render our React app onto the page, the DOM. These are separate libraries because you can actually use React to render things in a *non-browser* environment.

Anyways, when that finishes, go back to `rep_log_react.js`. Like with all libraries, to use React, we need to require or import it. I'll use the newer, *far* more hipster import syntax in this tutorial to import React from `react`.

## Elements & the Virtual DOM

Here's the idea behind React: it's actually really beautiful: *we* create React element objects that *represent* HTML elements, like an `h1` tag. These don't actually live on the page, they're just objects that *describe* HTML elements.

But then, we can tell React to look at our element objects, and use them to create *real* HTML elements on the page, or, on the DOM. This means that we will have a tree of element objects in React *and* a tree of element objects on the page. To say it a different way, we will have a *virtual DOM* in React and the *real DOM* on the page.

And... that's it! Of course, the *magic* is that, when we change some data on a React element object, React will update the corresponding DOM element automatically.

## Creating React Elements

So... let's create some React elements! To *really* master this, yea... we're going to do things the hard way first. But, it will be *totally* worth it.

Create a new `const el` set to `React.createElement()`. Make this an `h2` tag: we're building the title on top of the app. Pass `null` for the second argument: but this is where you could pass an array of any HTML attributes for the element. These are called "props" in React - but more on that later. For the third argument, pass *whatever* you want to put inside, like text: "Lift History!".

```
5 lines | assets/js/rep_log_react.js
1  import React from 'react';
2
3  const el = React.createElement('h2', null, 'Lift History!');
... lines 4 - 5
```

Cool! Let's `console.log(el)`: I want you to see that this is just a simple object. Go refresh the page. The element is not, *yet*, being rendered to the screen in any way. It's just a React element that describes a potential HTML element.

```
5 lines | assets/js/rep_log_react.js
... lines 1 - 3
4  console.log(el);
```

## Rendering React to the DOM

Now, go back to `index.html.twig`. To render the element onto the page, we need a target. The existing app is rendered in this `js-rep-log-table` element. Above this, add a new `<div class="row">` and inside, an empty div we can render into with `id=""`, how about, `lift-stuff-app`. Then, for a bit more separation, add a bunch of line breaks and an `<hr>`.

```

70 lines | templates/lift/index.html.twig
... lines 1 - 2
3  {% block body %}
4      <div class="row">
5          <div id="lift-stuff-app"></div>
6      </div>
7
8      <br><br><br><br><br><br>
9
10     <hr />
... lines 11 - 55
56 {% endblock %}
... lines 57 - 70

```

Awesome! Copy the id of the div. To render React to the DOM, we need to use that *other* package we installed: import ReactDOM from react-dom. Then, just, ReactDOM.render() to render our el into document.getElementById('lift-stuff-app').

```

7 lines | assets/js/rep_log_react.js
... line 1
2  import ReactDOM from 'react-dom';
... lines 3 - 5
6  ReactDOM.render(el, document.getElementById('lift-stuff-app'));

```

That's it! Step 1: create a React element object and, step 2, use ReactDOM and some boring, raw JavaScript to render it onto the page.

Let's go try it! Move over and refresh! Ha! We have our very first, but I, know very simple, React app. We deserve balloons!

## Nested Elements

Of course, in a *real* app, we're going to have more than just one element. Heck, we're going to have a big nested *tree* of elements inside of other elements, just like we do in normal HTML.

So... how could we put an element *inside* of our h2? First, break things onto multiple lines to keep our sanity. The answer is... by adding more and more arguments to the end of React.createElement(). Each argument - starting with the third argument - becomes a new child that lives inside the h2. For example, to create a nested span element, use React.createElement() with span, null and a heart Emoji.

```

13 lines | assets/js/rep_log_react.js
... lines 1 - 3
4  const el = React.createElement(
5      'h2',
6      null,
7      'Lift History! ',
8      React.createElement('span', null, '♥ ')
9  );
10
11 console.log(el);
... lines 12 - 13

```

Let's log el again. Then, flip over and... refresh!

Ha! There it is! Inspect the element: yep, the h2 tag with a span inside. Check out the logged Element: it now has two "children", which is a React term: the text and *another* React element object.

Awesome! But... you've probably already noticed a problem. Building a *real* app with many nested elements is going to get *really* ugly... *really* quickly. This React "element" idea is great in theory.... but in practice, it's a nightmare! We need another tool to save us. That tool is love. I mean, JSX.

# Chapter 3: JSX

This system of creating a "virtual DOM", or "tree" of React element objects and rendering that to the page is a really cool idea. But, it's already a total pain: imagine if we needed another `React.createElement()` inside of the span... then another element inside of that. Woof.

Because the React element objects are *meant* to represent HTML elements... it would be kinda cool if, instead, we could *actually* write HTML right here! Like, for example, what if we could say `const el =` then write an `h2` tag with Lift Stuff! and add a span tag inside with a heart. I mean, that's *exactly* what we're ultimately building with `React.createElement()`!

```
8 lines | assets/js/rep_log_react.js
... lines 1 - 3
4  const el = <h2>Lift Stuff! <span>♥ </span></h2>;
... lines 5 - 8
```

But, of course, this is *not* real JavaScript code: it's just me hacking HTML right into the middle of my JavaScript file. So, no surprise: PhpStorm is *so* angry with me. And, if you move back to the terminal tab that is running Webpack, oh, it's *furious*: Unexpected Token.

## JSX & The Babel React Preset

Well... fun fact! This crazy HTML syntax *is* actually valid. Well, it's not official JavaScript: it's something called JSX - an *extension* to JavaScript. To use it, all we need to do is teach Babel how to parse it. Remember, Babel is the tool that reads our JavaScript and "transpiles" it, or "rewrites" it, into older JavaScript that all browsers can understand.

To teach it how to parse JSX, open your `webpack.config.js` file. In normal Webpack, you need to install and enable a React Babel *preset*: a rule that understands JSX. In Encore, you can do this by adding `.enableReactPreset()`.

```
34 lines | webpack.config.js
... lines 1 - 3
4  Encore
... lines 5 - 28
29  .enableReactPreset()
30  ;
... lines 31 - 34
```

To make this take affect, go stop Encore and restart it:

```
$ yarn run encore dev --watch
```

Oh, it fails! Ah, we need to install a new package: copy the command name. Then, paste:

```
$ yarn add babel-preset-react --dev
```

By the way, the next version of this package will be called `@babel/preset-react`. So, if you see that package name in the future, don't worry, it's really the same thing.

And... done! Try Encore again:

```
$ yarn run encore dev --watch
```

Bah! It fails again - but this is my fault: I forgot to remove my extra `el` constant. After removing that, yea! Encore builds

successfully! This means that it *actually* understands our crazy JSX code! Try it - move to your browser and refresh!

```
8 lines | assets/js/rep_log_react.js
... lines 1 - 3
4  const el = <h2>Lift Stuff! <span>♥ </span></h2>;
5
6  console.log(el);
7  ReactDOM.render(el, document.getElementById("lift-stuff-app"));
```

## [JSX Vs React.createElement\(\)](#)

We get the *exact* same output as before! An h2 with a span inside. Now, here's the *really* important part. Just like before, we're using console.log() to print this el. Check this out in the browser - woh! It's a React element object! It's the *exact* same thing we had before!

This special syntax - JSX - allows you to write, what *looks like* HTML right inside JavaScript. But in reality, Babel translates this into React.createElement() calls: this JSX *generates* the React.createElement() code we had before!

Hey, we solved our problem! We found a *much* prettier and more convenient way to use the React element system.

## [Making PhpStorm also Love JSX](#)

Except... PhpStorm still *hates* me for using this. But, you can already see on top that it's asking us if we want to switch our language to support JSX. You can make the change there, or go to PhpStorm -> Preferences and search for "JSX". Under Languages & Frameworks, find JavaScript and change the language version to "React JSX".

Hit ok and... boom! PhpStorm is happy!

From now on, we will *exclusively* use JSX. But, don't forget what it really is! Just a fancy way of creating React element objects. Remembering this fact will help you understand React as things get more complex.

## [Adding the ESLint React Plugin](#)

A few minutes ago, we installed & setup the eslint package. And now that we're using React, we can activate some *new* ESLint rules that will give us all kinds of cool notices and warnings that are specific to developing in React.

To install these new ESLint rules, move over to your terminal, find your open tab and run:

```
$ yarn add eslint-plugin-react --dev
```

Once that finishes, open the .eslintrc.js file. To use the rules from this package, update the extends option to have plugin:react/recommended.

You won't notice anything immediately, but as we keep developing in React, I'll point out the warnings that come from this.

Ok, we've learned how to create & render React element objects. But to *really* use React, we need to talk about React components.



# Chapter 4: React Components

It works like this: we create React *element* objects and ask React to render them. But, React has *another* important object: a Component. Oooooooooo.

It looks like this: create a class called RepLogApp and extend something called React.Component. A component only needs to have one method render(). Inside, return whatever element objects you want. In this case I'm going to copy my JSX from above and, here, say return and paste.

```
12 lines | assets/js/rep_log_react.js
... lines 1 - 3
4 class RepLogApp extends React.Component {
5   render() {
6     return <h2>Lift Stuff! <span>♥ </span></h2>;
7   }
8 }
... lines 9 - 12
```

This is a React component. Below, I'm going to use console.log() and then treat my RepLogApp as if it were an element: <RepLogApp />.

```
12 lines | assets/js/rep_log_react.js
... lines 1 - 9
10 console.log(<RepLogApp />);
... lines 11 - 12
```

Finally, below, instead of rendering an *element*, we can render the *component* with that same JSX syntax: <RepLogApp />.

```
12 lines | assets/js/rep_log_react.js
... lines 1 - 10
11 ReactDOM.render(<RepLogApp />, document.getElementById('lift-stuff-app'));
```

Ok, go back and refresh! Awesome! We get the *exact* same thing as before! *And*, check out the console! The component becomes a React element!

## [The What & Why of Components](#)

This React component concept is something we're going to use a *lot*. But, I don't want to make it seem too important: because it's a *super* simple concept. In PHP, we use classes as a way to group code together and give that code a name that makes sense. If we organize 50 lines of code into a class called "Mailer", it becomes pretty obvious what that code does... it spams people! I mean, it emails valued customers!

React components allow you to do the same thing for the UI: group elements together and give them a name. In this case, we've organized our h2 and span React elements into a React component called RepLogApp. React components are sort of a named *container* for elements.

By the way, React components do have one rule: their names must start with a capital letter. Actually, this rule is there to help JSX: if we tried using a <repLogApp /> component with a lowercase "r", JSX would actually think we wanted to create some new hipster repLogApp HTML element, just like how a <div> becomes a <div>. By starting the component name with a capital letter, JSX realizes we're referring to our *component* class, not some hipster HTML element with that name.

## [Making our Imports more Hipster](#)

Anyways, a few minor housekeeping things. Notice that Component is a property on the React object. The way we have things now is fine. But, commonly, you'll see React imported like this: import React then { Component } from react. Thanks to this, you can just extend Component.

12 lines | [assets/js/rep\\_log\\_react.js](#)

```
1 import React, { Component } from 'react';
  ... lines 2 - 3
4 class RepLogApp extends Component {
  ... lines 5 - 7
8 }
  ... lines 9 - 12
```

This is pretty much just a style thing. And... honestly... it's one of the things that can make React frustrating. What I mean is, React developers like to use a *lot* of the newer, fancier ES6 syntaxes. In this case, the react module exports an object that has a Component property. This syntax is "object destructuring": it grabs the Component key from the object and assigns it to a new Component variable. Really, this syntax is not *that* advanced, and actually, we're going to use it *a lot*. But, this is one of the challenges with React: you may not be confused by React, you may be confused by a fancy syntax used in a React app. And we definitely don't want that!

We can do the same thing with react-dom. Because, notice, we're *only* using the render key. So instead of importing *all* of react-dom, import { render } from react-dom. Below, use the render() function directly.

12 lines | [assets/js/rep\\_log\\_react.js](#)

```
  ... line 1
2 import { render } from 'react-dom';
  ... lines 3 - 10
11 render(<RepLogApp />, document.getElementById('lift-stuff-app'));
```

This change is a *little* bit more important because Webpack should be smart enough to perform something called "tree shaking". That's not because Webpack hates nature, that's just a fancy way of saying that Webpack will realize that we only need the render() function from react-dom: not the *whole* module. And so, it will only include the code needed for render in our final JavaScript file.

Anyways, these are just fancier ways to import *exactly* what we already had.

Oh, but, notice: it *looks* like the React variable is now an *unused* import. What I mean is, we don't ever *use* that variable. So, couldn't we just remove it and only import Components?

Actually, no! Remember: the JSX code is *transformed* into React.createElement(). So, strangely enough, we *are* still using the React variable, even though it doesn't look like it. Sneaky React.

To make sure we haven't broken anything... yet, go back and refresh. All good.

## One Component Per File

Just like in PHP, we're going to follow a pattern where each React component class lives in its own file. In the assets/js directory, create a new RepLog directory: this will hold all of the code for our React app. Inside, create a new file called RepLogApp. Copy our entire component class into that file.

8 lines | [assets/js/RepLog/RepLogApp.js](#)

```
1 import React, { Component } from 'react';
2
3 class RepLogApp extends Component {
4   render() {
5     return <h2>Lift Stuff! <span>♥ </span></h2>;
6   }
7 }
```

Woh. Something weird just happened. Did you see it? We *only* copied the RepLogApp class. But when we pasted, PhpStorm auto-magically added the import for us! Thanks PhpStorm! Gold star!

But, check out this error:

ESLint: React must be in scope when using JSX.

Oh, that's what we *just* talked about! This is one of those warnings that comes from ESLint. Update the import to *also* import React.

Now, to make this class available to other files, use export default class RepLogApp.

```
8 lines | assets/js/RepLog/RepLogApp.js
1  import React, { Component } from 'react';
   ... line 2
3  export default class RepLogApp extends Component {
   ... lines 4 - 6
7  }
```

Back in rep\_log\_react.js, delete the class and, instead, import RepLogApp from ./RepLog/RepLogApp. Oh, and it's not too important, but we're actually *not* using the Component import anymore. So, trash it.

```
7 lines | assets/js/rep_log_react.js
1  import React from 'react';
   ... line 2
3  import RepLogApp from './RepLog/RepLogApp';
   ... lines 4 - 7
```

Awesome! Our code is a bit more organized! And when we refresh, it's *not* broken, which is always my favorite.

## [The Entry - Component Structure](#)

And actually, this is an important moment because we've just established a basic structure for pretty much any React app. First, we have the entry file - rep\_log\_react.js - and *it* has just one job: render our top level React component. In this case, it renders RepLogApp. That's the *only* file that it needs to render because eventually, the RepLogApp component will contain our *entire* app.

So the structure is: the one entry file renders the one top-level React component, and *it* returns all the elements we need from its render() method.

And, that's our next job: to build out the rest of the app in RepLogApp. But first, we need to talk about a super-duper important concept called props.

# Chapter 5: Props

So far, the component is 100% static. What I mean is, there are *no* variables at all. No matter what, each time we render RepLogApp, we will get the *exact* same result.

I said earlier that a React component is kind of like a PHP class. Well, when we instantiate a class in PHP, we can pass in different arguments to make different *instances* of the same class behave in different ways.

And... yea! We can do the exact same thing here: we can pass variables to a component when it's rendered, and use those in the render() method.

## Components are Instantiated

But, first, a quick note: when you use the JSX syntax for a component, React *instantiates* a new instance of our class. Yep, we can actually render multiple RepLogApp components on the page, and each of them will be a separate object. with separate data.

## Passing in a Prop

It turns out, that's hugely powerful. Suppose that we want to be able to render our RepLogApp in multiple places on the page at the same time. But, sometimes we want the heart Emoji, but sometimes we don't. To make that possible, we need to be able to pass a flag to RepLogApp that tells it whether or not to render the heart.

Inside rep\_log\_react.js, create a new const shouldShowHeart = true. I'll also update the render code to use multiple lines, just to keep things clear.

```
11 lines | assets/js/rep_log_react.js
... lines 1 - 4
5  const shouldShowHeart = true;
... lines 6 - 11
```

So, how can we pass this variable into RepLogApp? By adding what *looks* like an attribute: withHeart - I'm just making that name up - equals {shouldShowHeart}.

```
11 lines | assets/js/rep_log_react.js
... lines 1 - 6
7  render(
8    <RepLogApp withHeart={shouldShowHeart} />,
9    document.getElementById('lift-stuff-app')
10 );
```

## The JSX {JavaScript} Syntax

Woh. Wait. Something crazy just happened. We are inside of JSX on this line. And, because JSX is like HTML, we know that we could, of course, say something like withHeart="foo". That's true, but whenever you're in JSX, if you write {}, that puts you back into JavaScript mode! Once inside, You can write literally *any* valid JavaScript: like reference the shouldShowHeart variable or even add expressions. We'll do this *all* the time.

## Reading Props from Inside a Component

Now, I referred to withHeart as an "attribute". But, in React, this is actually known as a prop, and its the way that you pass "arguments" or "data" into your component. Inside RepLogApp, we can access any props that were passed to us via a this.props property.

Check this out: in render() create a new variable called heart and set it to empty quotes. Then, if this.props.withHeart - referencing the prop we passed in - say heart =, copy the span JSX from below, and paste it here.

```

15 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3  export default class RepLogApp extends Component {
4    render() {
5      let heart = "";
6      if (this.props.withHeart) {
7        heart = <span>♥ </span>;
8      }
9    }
10   }
11 }
12 }

```

Oh, and notice that when we use `this.props.withHeart`, we have an error from ESLint about some missing prop validation. That's just a warning, and we're going to talk about it later. For now, *totally* ignore it.

Below, I want to break my return statement onto multiple lines. You *can* use multiple lines to define JSX, as long as you surround it with parenthesis. I do this a lot for readability.

Finally, instead of the span, we want to print the heart variable. How? Use `{heart}`. Based on the value of the prop, this will print an empty string or a React element.

```

15 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 9
10  return (
11    <h2>Lift Stuff! {heart}</h2>
12  );
... lines 13 - 15

```

Right now, `withHeart` is equal to `true`. So let's see if this work: find your browser and refresh! Yes! We still see the heart! Change `shouldShowHeart` to `false` and try it again. The heart is gone!

## Rendering a Component Multiple Times

To *really* show off, change that value back to `true`, but let's see if we can render `RepLogApp` *multiple* times. Copy the JSX, paste, and set `withHeart` to `false`.

```

11 lines | assets/js/rep_log_react.js
... lines 1 - 6
7  render(
8    <RepLogApp withHeart={shouldShowHeart} /> <RepLogApp withHeart={false} />,
9    document.getElementById('lift-stuff-app')
10 );

```

But, as *soon* as we do this, the Webpack build fails! Find your terminal to see what it's complaining about:

Syntax Error: Adjacent JSX elements must be wrapped in an enclosing tag

This is less scary than it sounds. It's not that you can't put components next to each other like this, it just means that there must be just *one* element all the way at the *top* of our JSX tree. Each *component* also needs to follow this rule. And, `RepLogApp` already is: it has one top-level element: the `h2`.

To put just *one* element at the top of our element tree, there's a simple fix: add a `div` and render both components inside. Oh, and I completely forgot to use `{}` around my `"false"` - `false` is JavaScript.

14 lines | [assets/js/rep\\_log\\_react.js](#)

... lines 1 - 6

```
7   render(  
8     <div>  
9       <RepLogApp withHeart={shouldShowHeart} />  
10      <RepLogApp withHeart={false} />  
11    </div>,  
12    document.getElementById('lift-stuff-app')  
13  );
```

Now that Webpack is happy again, go back and refresh! Sweet! Our component is rendered *twice*: each is its own object with its own data.

Props are just about the *most* important concept in React, and they will be the *key* to us creating killer UI's that update dynamically.

Back in `rep_log_react.js`, we don't *really* need two of these components: so go back to just one. And, beautiful!

It's time to build out the rest of our app: first, by moving the table into `RepLogApp`.

# Chapter 6: Collection & Rendering a Table

The RepLogApp component eventually needs to render all the elements we see on the original app: with a table and a form. No problem! Go find the template for this page: templates/lift/index.html.twig. Hey! There's our table! And the form lives in this included template. Copy the *entire* old markup. Then, go back to RepLogApp and replace our code with this. But, I don't want to worry about the form yet... so remove that. Oh, and I kinda liked the withHeart feature, so let's make sure we're still printing {heart}.

```
39 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3  export default class RepLogApp extends Component {
4    render() {
... lines 5 - 9
10   return (
11     <div className="col-md-7 js-rep-log-table">
12       <h2>Lift Stuff! {heart}</h2>
13
14       <table className="table table-striped">
15         <thead>
16           <tr>
17             <th>What</th>
18             <th>How many times?</th>
19             <th>Weight</th>
20             <th>&nbsp;</th>
21           </tr>
22         </thead>
23         <tbody>
24           <tbody>
25             <tfoot>
26               <tr>
27                 <td>&nbsp;</td>
28                 <th>Total</th>
29                 <th className="js-total-weight"></th>
30                 <td>&nbsp;</td>
31               </tr>
32             </tfoot>
33           </tbody>
34         </table>
35       </div>
36     );
37   }
38 }
```

## [class -> className](#)

Sweet! Once again, PhpStorm just did an amazing thing - automagically - when we pasted in the code. Check out the className="table table-striped". Hmm, look at the original code in the template: it had class=""... because that's what it *should* be in HTML! Well... in React, you actually *can't* use class. You can see it's highlighted:

Unknown property class found, use className instead

React can't use class because class is a keyword inside of JavaScript. And for that reason, in JSX, you need to use className instead. But ultimately, this will render as a normal class attribute on the page.

And, don't worry, there aren't tons of weird attributes like this in React: this is basically the only one you're likely to use.

The point is: PhpStorm is smart enough to convert our pasted class props to `className` automatically. Notice that I said props: while we *think* of these as HTML attributes, they're technically props, which React ultimately renders as attributes.

## Finish our Static "Mock Up"

Let's clean a few things up! We don't need this `js-rep-log-table` class: it was used by the old JavaScript. And below, this is the column that prints the *total* weight. Remove the class that was used by the old JavaScript and, for now, just put a `TODO` inside.

And finally, *just* to see how it looks with data, let's hack in one fake row full of invented stuff. Use `...` for the last column: someday, we'll add a delete link here.

```
45 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 9
10     return (
11       <div className="col-md-7">
... lines 12 - 13
14       <table className="table table-striped">
... lines 15 - 22
23         <tbody>
24           <tr>
25             <td>Big Fat Cat</td>
26             <td>10</td>
27             <td>180</td>
28             <td>...</td>
29           </tr>
30         </tbody>
31         <tfoot>
32           <tr>
33             <th>TODO</th>
... lines 33 - 34
35
... line 36
37           </tr>
38         </tfoot>
39       </table>
40     </div>
... line 41
42   );
... lines 43 - 45
```

Cool! Building a static version of your app first is a *great* way to start. And JSX makes that really easy.

Let's go check it out: find your browser and refresh! Hey hey! This is starting to look real!

## Use Static Data First

In our old app, on page load, we make an AJAX request to load the "rep logs" - that's what we call our data - and use that to render the table.

Eventually, we'll do the same thing in React. But *before* you work with dynamic data, you should *first* make your app render using static data. Check this out, inside `render()`, create a new constant called `repLogs` and then set that to some fake data that matches the format of your API. We now have 3 fake rep logs with `id`, `itemLabel` and `totalWeight`.



```

57 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 3
4   render() {
... lines 5 - 9
10  const repLogs = [
11    { id: 1, reps: 25, itemLabel: 'My Laptop', totalWeightLifted: 112.5 },
12    { id: 2, reps: 10, itemLabel: 'Big Fat Cat', totalWeightLifted: 180 },
13    { id: 8, reps: 4, itemLabel: 'Big Fat Cat', totalWeightLifted: 72 }
14  ];
... lines 15 - 54
55  }
... lines 56 - 57

```

## Rendering a Collection

Below, inside the `tbody`, we basically want to convert each "rep log" into a `tr` React element with the data printed inside of it. To do that, we're going to use a *really* common pattern in React... which might feel a bit weird at first.

Above `render()`, create a new constant called `repLogElement` set to `repLogs.map()`. Pass this a callback function with one argument: `repLog`. I'll use the arrow syntax for the callback. Inside, we're going to return a React element via JSX: add parenthesis so we can use multiple lines. Then, just build out the row: `<tr>`, then `<td>` with `{repLog.itemLabel}`.

If you're not familiar with the `map` function, that's ok: it's much less common in PHP. Basically, it loops over each element in `repLogs`, calls our function, and then, whatever our function returns, is added to the `repLogElement` array. So, ultimately, `repLogElement` will be an array of `<tr>` React element objects.

Add the next `<td>`. Let's see... ah, this column is "How Many". Fill in the second column with `{repLog.reps}`, then another `<td>` with `{repLog.totalWeightLifted}` and finally one more with `...`: this will be the delete link... someday.

.

```

57 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 15
16  const repLogElements = repLogs.map((repLog) => {
17    return (
18      <tr key={repLog.id}>
19        <td>{repLog.itemLabel}</td>
20        <td>{repLog.reps}</td>
21        <td>{repLog.totalWeightLifted}</td>
22        <td>...</td>
23      </tr>
24    )
25  });
... lines 26 - 57

```

Great! Wait... but the `tr` has a little warning: something about a missing key prop. We'll talk about that in a minute. Until then, ignore that silly warning! What could go wrong?!

Now that we have an array of React element objects, this is pretty sweet: go down, delete the hardcoded row and - wait for it - just print `repLogElements`.

Yea, it looks a bit crazy: we're literally printing an array of React elements! But, try it - go back to your browser and refresh! It works! It prints each row!

But, we have a *big* warning from React. Let's fix that next, and introduce a new best practice to keep our code readable.

# Chapter 7: The key Prop & Inline Rendering

We just rendered an array of React `<tr>` element objects. And as a thank you, React has awarded us with a big ugly error! It says:

Warning: each child in an array should have a unique "key" prop.

Rude! Here we are, with a *perfectly* functional table, and React is ruining our magical moment. Well... ok, it's warning us for a good reason. And, PhpStorm was *also* trying to help.

## Why a Collection needs a key

Here's the deal: soon, we're going to use React to perform automagical updates to the UI when data changes. Eventually, we'll use this to add and remove rows to this table as we create and delete rep logs! But... when you have a collection of items like this, if we update the data on one of the rep logs, React isn't totally sure which row to update in the DOM. For example, if this rep log's weight suddenly changed to 150, it's possible that React would update the wrong row!

To help React, we need to give each row a unique key - kind of like how each row in a database table has a primary key. To do this, go to the outer-element of each item and literally add a key prop. This needs to be something that is unique *and* won't change. So, basically, it should be the id.

```
57 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3   export default class RepLogApp extends Component {
4     render() {
... lines 5 - 15
16    const repLogElements = repLogs.map((repLog) => {
17      return (
18        <tr key={repLog.id}>
... lines 19 - 23
24      )
25    });
... lines 26 - 54
55  }
56  }
```

Solved! This key prop isn't a big deal, it's just a chore you need to handle each time you render a collection. But don't worry: if you forget, React will remind you!

Try it now: head over and refresh! The page still works, and the warning is gone!

## Rendering Inline

There's one minor downside to this new setup. Up here, we use the map function to create an array of repLogElements. Down below, we render that.

What's the problem? Well, just that, if you're looking at render() to see your markup, when you see {repLogElements}, you need to scroll back up to see what this is. Whenever possible, it's better to keep all of your markup in one place.

And, we *can* do that... by being a bit clever. Copy the repLogs.map() code, then delete the repLogElements variable entirely. Back inside JSX, clear out the variable and... paste!

```

55 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 15
16     return (
... lines 17 - 19
20         <table className="table table-striped">
... lines 21 - 28
29             <tbody>
30                 {repLogs.map((repLog) => {
31                     return (
32                         <tr key={repLog.id}>
33                             <td>{repLog.itemLabel}</td>
34                             <td>{repLog.reps}</td>
35                             <td>{repLog.totalWeightLifted}</td>
36                             <td>...</td>
37                         </tr>
38                     )
39                 })}
40             </tbody>
... lines 41 - 48
49         </table>
... lines 50 - 51
52     );
... lines 53 - 55

```

That's it! It's really the *same* thing we had before! This loops over the `repLogs` array, builds an array of "rep log" element objects, then... prints them!

Try it! Move over and, refresh! Yes!

At first, the syntax may look weird. But, it's now *very* obvious that inside the `tbody`, we are printing a bunch of `tr` elements.

## Shorter Arrow Functions

And if this isn't fancy enough for you, well, you won't be disappointed. If you're still getting used to how the arrow functions look, you may not love this next change. But, if it's too weird, don't use it!

Because the arrow function is, um, a *function*, we usually surround the body of the function with curly braces. But, if the only line in your function is the return statement, you can *remove* the curly braces and just put the code for the return. Oops, I have one extra curly brace.

```

53 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 28
29     <tbody>
30         {repLogs.map((repLog) => (
31             <tr key={repLog.id}>
32                 <td>{repLog.itemLabel}</td>
33                 <td>{repLog.reps}</td>
34                 <td>{repLog.totalWeightLifted}</td>
35                 <td>...</td>
36             </tr>
37         ))}
38     </tbody>
... lines 39 - 53

```

We now have a function with one argument that *returns* this JSX element.

Move over and try it! Woohoo! It works! These are the types of little things in React that you don't *need* to do. And, if it makes your head spin, keep it simple. But, I want you to at least *know* about these tricks. Because, often, it's *these* types of shortcuts

that end up making React *look* hard.

Let's finish building the static version of our app next: by adding the form!

## Chapter 8: Build the Static App First

One of our *big* goals in this tutorial is to create a repeatable path to success. And, we're *already* doing that! Step 1 is always to create an entry file. That file doesn't do much *except* render a React component onto your page.

Step 2, in that React component, build out an entirely static version of your app. First do this in pure HTML. Then, create some hardcoded variables and render those. For example, we *first* built one dummy `tr` element by hand and *then* created a hardcoded `repLogs` array and used *that* to build the rows.

So, step 2 for success is to build your entire UI statically... and *then, soon*, we will make things dynamic and fancy.

### Adding the Static Form

The only UI that's missing from our app now is the form below the table. No problem! Go back into `templates/lift/index.html.twig`. Ah, the form lives in another template: `_form.html.twig`. Open that: it's just a normal, boring HTML form. That's perfect! Copy *all* of it, close the file, go back into `RepLogApp` and, under the table, paste!

```
90 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3   export default class RepLogApp extends Component {
4     render() {
... lines 5 - 15
16    return (
17      <div className="col-md-7">
... lines 18 - 48
49      <form className="form-inline js-new-rep-log-form" noValidate
50        data-url="{{ path('rep_log_new') }}">
... lines 51 - 83
84      </form>
85    </div>
... line 86
87  );
88  }
89  }
```

Except, scroll up a little bit, because we need to do some cleanup! The form doesn't need this class anymore: that was used by the old JavaScript. The same is true for the `data-url` element. And `noValidate` disables HTML5 validation. But, HTML5 validation is nice to have: it will enforce the required attribute on the fields. So, remove it.

```
89 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 15
16    return (
... lines 17 - 48
49    <form className="form-inline">
... lines 50 - 85
86  );
... lines 87 - 89
```

Oh, but I want you to notice something! The attribute was `noValidate` with a capital "V"! In the original template, it was `novalidate` with a lowercase "v": that's how the property is called in HTML. When we pasted it, PhpStorm updated it for us. This is one of those uncommon situations - like `class` and `className` where the HTML attribute is *slightly* different than what you need to use in React. I want to point that out, but don't over-think it: almost everything is the same, and React will usually warn you if it's not.

Cool! Try it out: refresh! Awesome! We have a form!

## Using form defaultValue

Hmm, but the styling is not *quite* right. *And*, we have a warning about using a defaultValue. Let's fix that first. We'll talk a lot more about forms in React later. But basically, *React* is a control freak, it *really* really\* wants to manage the values of any elements on your form, including which option is selected. So, instead of using selected="selected", you can use defaultValue="" on the select element and set it to the value of the option you want. I'll skip that part because the first option will be automatically selected anyways.

## Adding Ugly Manual Whitespace

Ok, back to the styling problem. Inspect element on the form itself, right click on it, and go to "Edit as HTML". Ah, React renders as one big line with *no* spaces in it. 99% of the time... we don't care about this: usually whitespace is meaningless. But, in this case, the form is an inline element: we *need* a space between the first two fields, and between the last field and the button. Without the space, everything renders "smashed" together.

The fix is both simple... and ugly: use JavaScript to print an extra space. Do it in both places. Yep, weird, but honestly, I *rarely* need to do this: it's just not a problem you have very often.

```
89 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 48
49     <form className="form-inline">
50     <div className="form-group">
... lines 51 - 66
67     </div>
68     { ' ' }
69     <div className="form-group">
... lines 70 - 77
78     </div>
79     { ' ' }
... lines 80 - 82
83     </form>
... lines 84 - 89
```

Try it again... yep! It looks *much* better! And we have our static app!

## Using the Dev Server

Before we go kick more butt, I want to make one adjustment to our workflow. We're using Webpack Encore. We ran it with:

```
$ yarn run encore dev --watch
```

Thanks to that, each time we update a file, it notices, and re-builds our assets. But rather than using encore dev, use encore dev-server. This is pretty interesting: instead of writing physical files to our public/build directory, this starts a new web server in the background that *serves* the built assets.

Check this out: go back and refresh the app. *No* visible differences. But now, view the page source. Suddenly, instead of pointing locally, like /build/layout.css, every asset is pointing to that new server: http://localhost:8080! This magic URL changing is thanks to Webpack Encore *and* some config changes we made to our Symfony app in the [Encore tutorial](#).

The web server - http://localhost:8080 - is the server that was just started by Webpack. When you request an asset from it, Webpack returns the latest, built version of that file. What's weird is that the built assets are no longer physically written to the filesystem. Nope, we just fetch the dynamic version from the new server.

Ultimately... this is just a different, fancier way to make sure that our code is always using the latest version of the built assets. But, as your app gets more complex, it *may* become possible for you to refresh your page *before* Webpack has been able to physically write the new files! However, if you use the dev-server and refresh too quickly, your browser will *wait* for the CSS or JavaScript files to be ready, before loading the page. And, as an added "nice thing", the dev server will cause our browser to automatically refresh whenever we make changes.

Anyways, our goal was to build our entire app with a static UI. And, we've done that! Sure, we have *some* fanciness: we

learned that we can pass "props" into our components and then use those to render things dynamically. So, our app is "kind of" dynamic, because we can control different parts of how it looks by passing different props. But... once our component is rendered, it's static. For example, once we render RepLogApp with a heart... it will have a heart *forever*.

But, the *whole* point of using React is so that our UI will automagically update when data changes! And we'll do that with something very, very important called state.

# Chapter 9: State: For Magic Updating Good Times

It turns out that a React component has *two* types of data: props, which we access with `this.props` and something different called state. Understanding the difference between props and state is, well, just about the most important thing in React.

## [Props vs State](#)

Once a component receives a prop, like, we pass `true` for the `withHeart` prop, that prop is constant. A component never changes a prop: it's just a value that it receives, *reads* and uses.

So, when you think of props, think of "data that doesn't change". We're going to *slightly* complicate that later... but for now, think: props are data that we never change, they're constant. Or, "immutable" to be more hipster.

The `withHeart` prop is a really good example: once our app renders, the heart will be there or not be there. We don't need that to ever change based on some user interaction or some data changing. It will *always* be there... or it will *always* not be there.

But when you want to make things *interactive*, when you have data that needs to *change* while your app is alive, well, then you need to store that data somewhere else: state, which you can access with `this.state`.

So... *why* exactly is there this distinction between props that must be immutable and state, which is allowed to change? We'll learn about that: it goes to the core of React's architecture.

## [Adding highlightRowId State](#)

But first, let's do some coding and add our first user interactivity! Here's the goal: when the user clicks a row in the table, I want to highlight that row.

From a data standpoint, this means that, somewhere in our `RepLogApp` component, we need to keep track of *which* of these rep log rows is currently highlighted. We can do that by keeping track of the highlighted row id. Literally, we could store that row 1, 2 or 8 is highlighted.

But because this data will *change* throughout the lifecycle of our app, we're not going to store the id in props. Nope, we're going to store it as state.

Once you've decide that you have some data that needs to be stored as state, you need to initialize that value on your state. That's always done the same way: by overriding the `constructor()` function. The constructor of React components receive a props argument. And then, you're supposed to call `super(props)` to execute the parent constructor. You'll see this pattern over and over again.

To set the initial state, just set the property directly: `this.state` equals an object, with, how about, a `highlightedRowId` key set to `null`. Nothing will be highlighted at first.

```
102 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3  export default class RepLogApp extends Component {
4    constructor(props) {
5      super(props);
6
7      this.state = {
8        highlightedRowId: null
9      };
10   }
... lines 11 - 100
101 }
```

## [Using State in render\(\)](#)

Cool! Down in `render`, we can use this data *just* like we do with props. But, let's use object destructuring to get this value as a



variable: `const { highlightedRowId } = this.state.`

```
102 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 11
12   render() {
13     const { highlightedRowId } = this.state;
... lines 14 - 99
100  }
... lines 101 - 102
```

This is another common React pattern. Instead of referring to `this.state.highlightedRowId` down below, we use destructuring so that we can be lazier later and use the shorter variable name.

I'll break the `tr` onto multiple lines. *If* this row should be highlighted, we'll give it a special class: add `className={}` and use the ternary syntax: if `highlightedRowId === this repLog's id`, then add an `info` class. Otherwise, print no class. This `info` class already exists in the CSS of our app.

```
102 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 11
12   render() {
... lines 13 - 25
26   return (
... lines 27 - 39
40     {repLogs.map((repLog) => (
41       <tr
42         key={repLog.id}
43         className={highlightedRowId === repLog.id ? 'info' : ''}
44       >
... lines 45 - 48
49     </tr>
50   )]}
... lines 51 - 98
99   );
100 }
... lines 101 - 102
```

Cool! If we try it now, we, of course, don't expect anything to be highlighted: we initialized the state to null. And, yep! It works... probably: none of the rows have the class.

## React Developer Tools

One of the most *awesome* things about developing with React is a tool called the "React Developer Tools". It's an extension for Chrome or Firefox, so you can install it from the Chrome Web Store or from Firefox Add-Ons.

After installing it, your browser's developer tools will have a new React tab! Oooooooo. Check this out: it shows your entire component and element hierarchy. *And*, you can click to see all the different props for every part.

Click on the `RepLogApp` component on top. Woh! It shows us the `withHeart` prop and the `highlightedRowId` state! *And*, we can mess with it! Remember: the rep log ids are 1, 2 and 8. Change `highlightedRowId` to 2. Boom! That row instantly updates to have the class! Change it to 1... and 8. Super fun!

## How React Re-Renders Things

This looks like magic... but really, it's just that React is really, really smart. Behind the scenes, whenever some "state" changes on a React component, React automatically re-executes the `render()` method on that component. So, if we change this number from 8 back to 2, it calls the `render()` method again and *we* return the new React element objects.

Because of this, you *might* think that *all* of this HTML is completely replaced each time we re-render. But actually, dang, nope! React is, yet again, too smart for that. Instead of replacing everything, React *compares* the React element objects from *before* the re-render to the *new* element objects after. Yep, it performs a "diff" to see what changed. And then, it only updates

the parts of the DOM that *need* to change.

We can actually *watch* this happen! Open the table but collapse the rows so we can see them all. Then, I'll re-select the RepLogApp component and scroll back down. Watch closely when we change the state: from 2 to 1. Did you see it highlight the two class attributes that changed in yellow? Watch again: 1 to 8.

That yellow highlight is my browser's way of telling us that these two attributes were *literally* the *only* thing that changed. In React, we re-render all of the elements. But in the DOM, React only updates the things it needs to.

The big takeaway is this: state is allowed to change. And each time it *does*, React calls `render()` on our component and updates the DOM as needed.

### But... who Updates State?

Of course, our ultimate goal is *not* just to update state via the cool React dev tools. Nope, we want to update the state when the user clicks a row.

Before we do that, there is *one* small housekeeping item. In addition to destructuring state, I like to do the same with props. Add `const { withHeart } = this.props`. Then below, use `if withHeart`.

```
103 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 11
12   render() {
... line 13
14     const { withHeart } = this.props;
... lines 15 - 16
17     if (withHeart) {
... line 18
19     }
... lines 20 - 100
101  }
... lines 102 - 103
```

It's a small detail, but it's nice to setup *all* the variables right on top.

Now, let's add some click magic and update our state!

# Chapter 10: Handling Events (like onClick)!

I want to do something when the user clicks the `tr` element. In React, how can we attach event listeners? What is the React version of selecting an element in jQuery and adding an on click function?

## Attaching an Event

Oh, you're going to *love*... or maybe hate the answer. I love it, because it's simple! To add a click handler to this `tr` add... `onClick` and pass this a function. I'll use an arrow function and, for now, just `console.log('OMG - an onClick!')`.

```
104 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 11
12   render() {
... lines 13 - 26
27   return (
... lines 28 - 40
41     {repLogs.map((repLog) => (
42       <tr
... lines 43 - 44
45       onClick={() => console.log('OMG an onClick!')}
46     >
... lines 47 - 51
52   )})
... lines 53 - 100
101 );
102 }
... lines 103 - 104
```

Move over, refresh, click, and... find the terminal. Boom!

## Updating with `this.setState()`

*Cool.* Let's review our goal: to highlight a row when we click on it. So... hmm... `onClick`: if we could update the `highlightedRowId` state to the correct id, React would re-render and take care of the rest! Easy! Inside the arrow function, update the state with `this.setState()`. Pass this an object with the state key or keys that you want to change. For us, `highlightedRowId` set to the id of *this* rep log: `repLog.id`.

```
104 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 40
41   {repLogs.map((repLog) => (
42     <tr
... lines 43 - 44
45     onClick={() => this.setState({highlightedRowId: repLog.id})}
46   >
... lines 47 - 51
52   )})
... lines 53 - 104
```

Coolio! But, an important note! In the constructor, we initialized the state by setting the `this.state` property directly. This is the *only* place, *ever*, that you will change or set the state property directly. *Everywhere* else, always, you need to call `this.setState()`. If you don't, puppies will stare at you with sad eyes.

And, more important, if you modify the state property directly, React won't re-render. The reason is simple: this is what React uses to *know* that you changed the state and so, to start the re-rendering.

Bah, let's go try it already! Refresh! And... click! Woohoo! We just added our *first* bit of interactivity. In the React dev tools, if you click on RepLogApp, you can watch the highlightedRowId state change as we click the rows. Pretty freaking cool.

## The SyntheticEvent

Just like with jQuery or plain JavaScript, when you add an event callback, your function is passed an event object. We don't need the event in this case, but it contains all the same information you're used to having. Actually, this isn't a native DOM "event" object. React passes you what's called a "SyntheticEvent": an event object that wraps the normal event, has all the same methods and properties, but adds a few things to make life easier.

```
104 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 41
42      <tr
... lines 43 - 44
45        onClick={(event) => this.setState({highlightedRowId: repLog.id}) }
46      >
... lines 47 - 104
```

## Moving the Handler to the Class

Putting all this logic inline is fine... but it can become hard to read. So, instead, I like to make the handler a property on my class. Start by adding a new method: handleRowClick that will accept the repLogId that was just clicked and also the event object itself... just to show that we can we pass this:

Next, steal the state-setting code and paste it here, but with highlightedRowId set to repLogId. And... we should probably close the method so Webpack isn't so mad at me!

```
108 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 11
12    handleRowClick(repLogId, event) {
13      this.setState({highlightedRowId: repLogId});
14    }
... lines 15 - 108
```

Below, call it: this.handleRowClick() with repLog.id and event.

```
108 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 15
16    render() {
... lines 17 - 30
31      return (
... lines 32 - 45
46        <tr
... lines 47 - 48
49          onClick={(event) => this.handleRowClick(repLog.id, event)}
50        >
... lines 51 - 104
105      );
106    }
... lines 107 - 108
```

I like it! Let's make sure we didn't bork our cool app: back to the browser! Refresh! Yea! It *still* works!

*This* is the power of React! It doesn't care *how* many different things in your UI need to change when some state changes, it takes care of everything.

And now, it's time to talk about organization. RepLogApp is *big*, and when things get too big, they get confusing. Let's move some code into a new child component.

# Chapter 11: Child Component

Our RepLogApp component is getting kinda big! I'm so proud! It's not only the amount of HTML, but also its complexity. We're now handling an event, updating state in a handler function *and*, below, this repLogs row stuff is pretty complex on its own!

In PHP, if you're working on a class, sometimes that class can become so big or so *complex* that, for your own sanity, you choose to create a *new* class and move some of that logic into it. Another reason you might create a new class is if you want to make some of your logic re-usable.

Well, that exact idea is true in React: when a component becomes too big or too complex & confusing, you can choose to move part of it into a *new* component. This isn't some *ground-breaking* strategy: it just simple code organization! And, in theory, you could re-use the new component in multiple places.

## Creating the RepLogList Component

Because the rep log table is already pretty complex, let's move that into its own component first. In the RepLog directory, create a new file: how about, RepLogList.js. Inside, every React component begins the same way: import React and { Component } from react. Then, export default class RepLogList extends Component. Add the *one* required method: render().

```
29 lines | assets/js/RepLog/RepLogList.js
1  import React, { Component } from 'react';
   ... line 2
3  export default class RepLogList extends Component {
4    render() {
   ... lines 5 - 26
27  }
28 }
```

So... hmm... I basically want to move my rep log rows into that component. We could move the whole table, or just the inside - don't over-think it. Let's copy all of the tbody. Then, return, add parenthesis so we can use multiple lines and, paste!

Cool! Of course, we're missing the repLogs variable! Right now, because that's still hardcoded, let's just move that variable over into the render() method of the new component.

29 lines | assets/js/RepLog/RepLogList.js

... lines 1 - 3

```
4   render() {
5     const repLogs = [
6       { id: 1, reps: 25, itemLabel: 'My Laptop', totalWeightLifted: 112.5 },
7       { id: 2, reps: 10, itemLabel: 'Big Fat Cat', totalWeightLifted: 180 },
8       { id: 8, reps: 4, itemLabel: 'Big Fat Cat', totalWeightLifted: 72 }
9     ];
10
11    return (
12      <tbody>
13        {repLogs.map((repLog) => (
14          <tr
15            key={repLog.id}
16            className={highlightedRowId === repLog.id ? 'info' : ''}
17            onClick={(event) => this.handleRowClick(repLog.id, event)}
18          >
19            <td>{repLog.itemLabel}</td>
20            <td>{repLog.reps}</td>
21            <td>{repLog.totalWeightLifted}</td>
22            <td>...</td>
23          </tr>
24        ))}
25      </tbody>
26    )
27  }
```

... lines 28 - 29

But, we *do* still have one problem: `highlightedRowId`. Um, ignore that for a minute. Back in `RepLogApp`, delete the `tbody`. At the top, this is cool: `import RepLogList from './RepLogList'`. And because `RepLogList` is a component, we can render it just like we did with `RepLogApp`: go into the middle of the markup and add `<RepLogList />`.

90 lines | assets/js/RepLog/RepLogApp.js

... line 1

```
2   import RepLogList from './RepLogList';
3
4   export default class RepLogApp extends Component {
5     ... lines 5 - 16
17   render() {
18     ... lines 18 - 25
26   return (
27     ... lines 27 - 29
30     <table className="table table-striped">
31       ... lines 31 - 38
39       <RepLogList/>
40       ... lines 40 - 47
48     </table>
49     ... lines 49 - 86
87   );
88 }
89 }
```

### Leaving State at the Top Level (for now)

That's it! We have successfully broken our big component into a smaller piece. Well, I guess we shouldn't celebrate *too* much, because, when we refresh, in the console, yep! React is always trying to bring us down: the `highlightedRowId` variable

is not defined in RepLogList!

That makes perfect sense: our child component - RepLogList - needs to know this value so that it can add the info class. But... hmm... we have a problem! The highlightedRowId state lives in a different component: our top-level RepLogApp component! So, how can access the state of our parent component?

Well, before I answer that, there is *technically* another option: we could just move the highlightedRowId state into the RepLogList component. And, technically, this would work! Look closely: RepLogApp isn't using that data anywhere else! So if we moved the state, everything would work!

But... for a reason I can't fully explain yet, I *don't* want you to do that. Nope, I want you to leave *all* of your state in the top level component of your app. That means, I want *all* of your child components to have zero state. Don't worry: we'll talk a lot more about why later.

## Passing Props down the Tree

But, because I'm being rude and *forcing* you to keep all of your state in RepLogApp, the question becomes: how can we *pass* this highlightedRowId state into RepLogList?

Guess what? We *already* know the answer! We *already* know how to pass data into a component: *props*. We have the highlightedRowId variable that's coming from state. Scroll down to RepLogList and add a new prop: highlightedRowId={} and pass that variable.

```
90 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 38
39      <RepLogList highlightedRowId={highlightedRowId}/>
... lines 40 - 90
```

And *now* we can go back into RepLogList and use this in render()! At the top, let's continue to destructure our props & state: const { highlightedRowId } = this.props. And, just like earlier, *ignore* this error about props validation: we'll talk about that soon.

```
31 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4    render() {
5      const { highlightedRowId } = this.props;
... lines 6 - 28
29  }
... lines 30 - 31
```

Ok... we're done! Move back to your browser and, refresh! It works! And if you check out the React dev tools, you can still see RepLogApp on top... but down here, hey! There is the embedded RepLogList. Now, things get fun: click back on RepLogApp and change the state to 2. This causes React to re-render that component. Check out RepLogList again - yea! You can see that its prop automatically updated!

This highlights one really, really important detail: while you may have multiple components that have *some* state, each *piece* of state like the highlightedRowId - needs to live in exactly *one* component. What I mean is: you are *not* allowed to have, for example, a highlightedRowId state in RepLogApp and also a highlightedRowId state in RepLogList. Nope! That would *duplicate* that data. Instead, each piece of state will live in just one component. And then, if a child component needs that data, we'll pass it as a prop.

## Props are Immutable... but Change

We already know that whenever something updates the state of a component, React automatically re-renders that component by calling render(). And actually, the same is true for props. When the highlightedRowId state changes, this changes the props of RepLogList and *that* causes it to *also* re-render. Which, is *exactly* what we want!

But, earlier, I told you that props are immutable: that props can never be changed. That's true, but it's maybe not the best way to explain it. In RepLogApp, when the highlightedRowId state changes, we *will* pass a *new* value to RepLogList for the highlightedRowId prop. But, here's the important part: once RepLogList receives that prop, it never changes it. You will *never* change something on this.props.

We're going to see this pattern over and over again: we hold state in one component, change the state in that component and then pass that state to any child component that needs it as a prop. And *now* we know that when that state changes, all the child components that use it will automatically re-render.

But... our click handling code is now broken! Let's fix it!



# Chapter 12: Notifying Parent Components: Callback Props

We're storing the `highlightedRowId` state in our top level component and passing it down to our child component as a prop. Inside the child, we read that prop and use it in `render()`. It's all very zen.

Until... we go to the browser and click on one of the rows. Whoops! Our console exploded:

```
handleRowClick is not a function
```

This comes from `RepLogList`. Yep... it's right! There is *no* `handleRowClick` method on `RepLogList`. That method lives on the *parent* component: `RepLogApp`.

## Communicating from Child to Parent

We've just hit another one of those very important moments. The parent component passes data down to the child via props. Thanks to this, the child component can be really dumb! It just receives props and renders them. That's it. *Importantly*, the child never communicates back *up* to the parent. Heck, the `RepLogList` component doesn't even *know* who its parent is! And, in theory, we could render `RepLogList` from 10 different parent components!

So... that leaves us in a pickle: inside `onClick`, we need to update the `highlightedRowId` state on our parent component. How can a child component update the state of a parent? The answer: it can't!

Wait wait, it's not time to panic... yet. Remember: the child component doesn't know who its parent is... heck! It doesn't even know that `highlightedRowId` is something that is stored in state! `RepLogList` just says:

```
I don't know, I'm just a React component. Somebody passes me a highlightedRowId prop and I render it. I don't know and I don't really care if it's stored in state.
```

So, how can we solve this problem? In the same way that we pass *data* from a parent component to a child component, we can also pass *callback* functions from parent to child. The child can *effectively* notify the parent when something happens by *calling* that function!

## Passing a Callback Prop

It's best to see this in action. First, just to clean things up, in the handler function, remove the event argument: we were never using this anyways. Next, down where we render `RepLogList`, let's break this into multiple lines. Then pass in another prop called `onRowClick` set to `this.handleRowClick`. But, make sure you don't *call* that function: just pass it as a reference.

```
93 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 25
26     return (
... lines 27 - 38
39         <RepLogList
40             highlightedRowId={highlightedRowId}
41             onRowClick={this.handleRowClick}
42         />
... lines 43 - 89
90     );
... lines 91 - 93
```

Thanks to this, in the child component, we have a fancy new `onRowClick` prop. Destructure this into a new variable. Then, `onClick`, we're dumb: we don't know *anything* about state, but we *do* know that we're passed an `onRowClick` prop, and that we're supposed to call this when the row is clicked! Cool! Call it and pass it `repLog.id`.

```

31 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4   render() {
5     const { highlightedRowId, onRowClick } = this.props;
... lines 6 - 12
13   return (
... line 14
15     {repLogs.map((repLog) => (
16       <tr
... lines 17 - 18
19         onClick={() => onRowClick(repLog.id)}
20       >
... lines 21 - 25
26     )})
... line 27
28   )
29 }
... lines 30 - 31

```

That's it! When the user clicks the row, our dumb component will execute the callback and pass the rep log id. The parent maintains complete control of what to do when this happens.

Let's try it! Move over and refresh. Bah! We still have an error:

Cannot read property setState() of undefined

Whoops! Whenever we have a callback handler function, we need to guarantee that the `this` keyword is bound to this object. There are a few ways to do this, but I usually fix this in one consistent way: go to the constructor and add `this.handleClick = this.handleClick.bind(this)`.

```

95 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 3
4   export default class RepLogApp extends Component {
5     constructor(props) {
... lines 6 - 11
12     this.handleClick = this.handleClick.bind(this);
13   }
... lines 14 - 93
94 }

```

Now, no matter *who* calls this method, this will always refer to this RepLogApp instance.

Refresh one more time. And, click! We got it!

We just saw a *massively* important pattern. Our state lives on our top level component. Then, we communicate *to* our children by passing data as props *and* we allow those children to communicate *back* to the parent component by passing them *callback* functions that should be executed when some interaction happens.

Oh, and by following this pattern, we've started to identify two types of components: *stateful* smart component and *stateless* dumb components. An important distinction we'll talk about next.

# Chapter 13: Smart vs Dumb Components

So far, only RepLogApp has state. But, any component is allowed to have state, as long as each specific *piece* of state like highlightedRowId lives in just one place and isn't duplicated. But, yea, in general, any component can have state.

However, I'm going to create a rule, for now. And later, we'll talk about when we can *bend* this rule. For now, I want you to keep *all* of your state on the one, top level component. This means that all of your *other* components, which, right now is just one, will be *stateless*.

Hmm, if you think about this, it's a bit like how controllers and templates work in PHP. RepLogApp is like a controller: it's the place that controls all the data and logic. It updates state, and will eventually load and save things via AJAX calls.

Then, RepLogList is like a template. It does... nothing. It's dumb! It just receives data and prints that data. This separation is intentional. It means that we have *two* types of components: smart, *stateful* components, sometimes called "container components". And dumb, *stateless* components, sometimes called presentation components.

## Stateless, Functional Components

Most dumb, stateless components also have something else in common: they usually only have one method: render(). So, *purely* for convenience, or I guess, laziness, you'll often see stateless components written, not as a *class*, but as a function.

Update this to: export default function RepLogList, but now without extends Component. When a component is a function, React passes you one arg: props. Now, we can remove one function level and... I'll unindent everything.

```
29 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 2
3   export default function RepLogList(props) {
... lines 4 - 27
28 }
```

Yep, the component is now *just* the render function... because that's all we needed! Refresh to try it! Oh, big error:

cannot read property props of undefined

Of course! Once a component is a function, there is no *this* anymore! That's fine, just change the code to use props. Hmm, destructuring everything in one place made that easy...

```
29 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4   const { highlightedRowId, onRowClick } = props;
... lines 5 - 29
```

Try it again! Move over and... reload! Yeehaw! Success!

We just saw *another* important pattern: we will have one smart component on top, and then, because all its children are stateless and dumb, we will write those components as functions. We *are* going to bend this rule later: you *can* have more than one smart component and sometimes a "dumb" component *can* have state. But, until then, internalize this rule: one smart component on top, and then all dumb, *functional*, components inside of it that just receive data.

## Smart Components should not have HTML

Ok, so, a smart component is like a controller in Symfony. Except... check out RepLogApp. It's a mixture of logic and... gasp! Markup! We would *never* put HTML code into our Symfony controllers: controllers should be *pure* logic.

And... surprise! We're going to follow that *same* rule with React components. New rule: a smart component should hold state & logic, but *no*, or, very little markup. To make this possible, a smart component should always *wrap* a dumb component.

## A Smart Component Wraps a Dumb Component

Yep, we're going to split RepLogApp into two components: one with all the logic and another will all the markup. Create a new file called RepLogs.js: this will be our stateless, dumb component. So, this will look a lot like RepLogList: import React from react. And then, export default function RepLogs.

```
73 lines | assets/js/RepLog/RepLogs.js
1  import React from 'react';
   ... line 2
3  export default function RepLogs(props) {
   ... lines 4 - 72
73 }
```

Next, go copy *all* of the code from RepLogApp's render() function and, paste it here.

```
73 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 2
3  export default function RepLogs(props) {
4    let heart = "";
5    if (withHeart) {
6      heart = <span>♥ </span>;
7    }
8
9    return (
10     <div className="col-md-7">
11       <h2>Lift Stuff! {heart}</h2>
12
13       <table className="table table-striped">
   ... lines 14 - 70
71     </div>
72   );
73 }
```

This new component has basically no logic, except for a tiny bit on top that's related to the markup itself.

Back in RepLogApp, delete all of that code! Instead, on top, import RepLogs from ./RepLogs. And then, in render, all we need is <RepLogs />.

```
27 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3  import RepLogs from './RepLogs';
   ... line 4
5  export default class RepLogApp extends Component {
   ... lines 6 - 19
20    render() {
   ... lines 21 - 23
24      return <RepLogs/>
25    }
26  }
```

That is it! Oh, it's great: look how pure & clean the top level component is! Our business logic is *much* easier to read. And all the markup responsibilities now belong to RepLogs.

By the way, *this* is why smart components are often called "container" components: they are a container around a dumb, *presentational* component. People often even use that to name their components, like RepLogsContainer instead of RepLogApp.

## Passing Props to the Dumb Component

Anyways, I'm sitting here celebrating our genius, but this won't actually work yet: RepLogs needs a few pieces of data:

withHeart and highlightedRowId. Both of these are available in RepLogApp. Oh, and we also need to pass a prop for the handleClick callback: *that* method *also* lives in RepLogApp.

But, before we fix that, add the missing import on top: import RepLogList from ./RepLogList.

```
76 lines | assets/js/RepLog/RepLogs.js
```

```
... line 1
```

```
2 import RepLogList from './RepLogList';
```

```
... lines 3 - 76
```

Then, while we're here, let's destructure the props we're about to receive:  
const { withHeart, highlightedRowId, onClick } = props.

```
76 lines | assets/js/RepLog/RepLogs.js
```

```
... lines 1 - 3
```

```
4 export default function RepLogs(props) {
```

```
5   const { withHeart, highlightedRowId, onClick } = props;
```

```
... lines 6 - 75
```

```
76 }
```

Use the new onClick variable down below: pass *this* into RepLogList.

```
76 lines | assets/js/RepLog/RepLogs.js
```

```
... lines 1 - 11
```

```
12 return (
```

```
... lines 13 - 24
```

```
25   <RepLogList
```

```
... line 26
```

```
27     onClick={onClick}
```

```
28   />
```

```
... lines 29 - 74
```

```
75 );
```

Finally, head back to RepLogApp so that we can pass these props. I'll break things onto multiple lines, then add:  
withHeart={withHeart}, highlightedRowId={highlightedRowId} and onClick={this.handleClick}... being sure not to actually *call* that function, even though PhpStorm is trying to trick us!

```
32 lines | assets/js/RepLog/RepLogApp.js
```

```
... lines 1 - 18
```

```
19 render() {
```

```
... lines 20 - 22
```

```
23   return (
```

```
24     <RepLogs
```

```
25       highlightedRowId={highlightedRowId}
```

```
26       withHeart={withHeart}
```

```
27       onClick={this.handleClick}
```

```
28     />
```

```
29   )
```

```
30 }
```

```
... lines 31 - 32
```

Oh, and I made a big ugly mistake! In RepLogs, import from RepLogList: I was trying to import myself! Strange and creepy things happen if you try that...

Ok, let's do this! Refresh! Yes! It still looks nice.

So here is our current system: one smart component on top, which acts like a controller in Symfony. Then, it renders a dumb, presentation component, just like how a controller renders a template. After that, you may *choose* to also render *other* dumb

components, just to help keep things organized. Heck, we do that same thing in Symfony: a template can *include* another template.

This "pattern" is not an absolute rule, and, we'll talk more about how and when you'll bend it. But, generally speaking, by following this pattern upfront - even if you don't completely understand *why* it's important - it will save you big time later.

# Chapter 14: Prop Validation: PropTypes

Look at RepLogList: it uses 2 props: highlightedRowId and onClick. But, what *guarantees* that RepLogList is *actually* passed these props? What prevents us from *forgetting* to pass those two *exact* props from RepLogs?

The answer is... absolutely nothing! In PHP, when you instantiate an object, if that object has a constructor with required arguments, we are forced to pass those arguments. But, with React components, it's the wild west! There is nothing like that. There is simply no guarantee that we are passed *any* of these props. There's also no way for a component to easily document, or advertise what props it needs.

This is an especially big problem with onClick, because, if we forget to pass this prop, our whole app will break when someone clicks on a row.

## Introducing PropTypes

To fix this, React leverages something called "prop types", which uses an external library. Find your terminal and install it:

```
$ yarn add prop-types --dev
```

And... ding! With prop types, we can tell React *exactly* which props a component needs, and even what they should look like. At the bottom of the component, add RepLogList.propTypes = and pass it an object. This is where you describe all the different props this component might have. But first, back on top, import PropTypes from prop-types. And... back down below, add highlightedRowId set to PropTypes.any. Then, onClick set to PropTypes.func.

```
35 lines | assets/js/RepLog/RepLogList.js
```

```
... line 1
2  import PropTypes from 'prop-types';
... lines 3 - 30
31  RepLogList.propTypes = {
32    highlightedRowId: PropTypes.any,
33    onClick: PropTypes.func
34  };
```

For highlightedRowId, we could have used PropTypes.number, because, it *is* a number right now. But later, we're going to refactor our ids to be uuid's, which are a string.

## PropTypes Give us Warnings

Let's see this in action! Go back to RepLogs and, instead of passing in a function, be devious: pass a string!

Check it out: move back to your browser! An error! Oh, and, notice: the page refreshed automatically *before* we got here. That's thanks to the Encore dev-server we're running: when we save a file, our browser automatically refreshes, which, is kinda nice.

Anyways, we see:

```
Invalid prop onClick of type string supplied to RepLogList, expected function.
```

*This* is what PropTypes give you: clear & early warnings when we mess up. I mess up a lot!

## Required PropTypes

The highlightedRowId prop in RepLogList is technically an *optional* prop: if we forget to pass it... no problem! No rows are highlighted. But the onClick prop... that's a different story: if we forget this, boom! Our code will explode in grand fashion when the user clicks a row.

By default, all propTypes are optional. To make one required, just add `.isRequired`. I'm kidding, add `.isRequired`. But, I feel like my name would have been much more awesome.

```
35 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 30
31 RepLogList.propTypes = {
... line 32
33   onClick: PropTypes.func.isRequired
34 };
```

Back in RepLogs, let's mess with stuff! "Forget": to pass that prop entirely. Move to your browser and... yep!

The prop `onClick` is marked as required... but its value is undefined.

## [PhpStorm ♥ 's propTypes!](#)

I love it! Enough fun: let's re-add the prop. Woh! We're now getting auto-complete!! Yeaaa! PhpStorm *reads* the propTypes and uses them to help us out.

And, in RepLogList, before, we had big red warnings when we referenced our props. That came from ESLint: it was telling us that we forgot to add the prop types. Now, everyone is happy.

## [Adding propTypes Everywhere Else](#)

Because propTypes are *amazing*, let's add them everywhere else, like in RepLogApp. Here, we're relying on a `withHeart` prop. And, *there* is the warning I was just talking about:

Missing props validation

You guys know the drill! First, import propTypes from `prop-types`. Then, at the bottom, `RepLogApp.propTypes` = an object with `withHeart` set to `PropTypes.bool`. The prop isn't *really* required, so I'll leave it optional.

```
36 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 2
3 import PropTypes from 'prop-types';
... lines 4 - 33
34 RepLogApp.propTypes = {
35   withHeart: PropTypes.bool
36 }
```

The *last* place we need propTypes is in RepLogs: we depend on 3. Go copy the import statement from RepLogApp, and paste! At the bottom, add the `RepLogs.propTypes` = part.

```
84 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 2
3 import PropTypes from 'prop-types';
... lines 4 - 78
79 RepLogs.propTypes = {
... lines 80 - 82
83 };
```

The 3 props are: `withHeart`, `highlightedRowId` and `onClick`. Steal `withHeart` from RepLogApp and paste. Then, more stealing! Get the other two from RepLogList and put those here too.



84 lines | [assets/js/RepLog/RepLogs.js](#)

... lines 1 - 78

```
79 RepLogs.propTypes = {  
80   withHeart: PropTypes.bool,  
81   highlightedRowId: PropTypes.any,  
82   onClick: PropTypes.func.isRequired  
83 };
```

Hmm, this shows off *another* common thing in React. Frequently, you'll pass props into one component, just so that it can pass them into *another* component. For example, in RepLogApp, we pass 3 props. But, two of them aren't even used in RepLogs! We just pass them straight to RepLogList!

This "props passing" can be kind of annoying. But, it's not necessarily a sign of bad design. It's just part of using React. There *are* ways to organize our code to help this, but many are more advanced. The point is: this is ok.

Next, let's make a small production optimization with prop types.

# Chapter 15: Removing propTypes on Production

The *whole* purpose of propTypes is to help us during development: they don't add any actual functionality to our code. And, for that reason, some people *remove* the propTypes code when they build their assets for production. It's not a big deal, it just makes your final JS files a *little* bit smaller.

This is *totally* optional, but let's do it real quick! Google for "babel-plugin-transform-react-remove-prop-types". Wow! First prize for longest name!

This is a Babel plugin that can remove propTypes. Copy the library name, find your terminal, and get it!

```
$ yarn add babel-plugin-transform-react-remove-prop-types --dev
```

While that's downloading, go back to its docs. Usually, this is configured via a .babelrc file: this activates the plugin on the production environment. Except, because we're using Webpack Encore, *it* handles the Babel configuration for us.

Fortunately, Encore gives us a hook to modify that config. Add .configureBabel() and pass this a function with one arg: call it babelConfig. Now, when Encore builds, it will create our Babel configuration, then call this function so we can modify it. We need to add a new env key, with this config below it. Copy the production, plugins part. Then, add babelConfig.env = and paste. This is safe because, if you logged the babelConfig object, you would see that Encore doesn't include an env key. So, we're not overriding anything.

```
41 lines | webpack.config.js
... lines 1 - 3
4  Encore
... lines 5 - 29
30  .configureBabel((babelConfig) => {
31    babelConfig.env = {
32      "production": {
33        "plugins": ["transform-react-remove-prop-types"]
34      }
35    }
36  })
... lines 37 - 41
```

Oh wait, actually, I made a mistake! This totally won't work! That's because we can't rely on Babel to know whether or not we're creating our production build. Instead, use if Encore.isProduction(). Then, inside, add the plugin with babelConfig.plugins.push(), copy the plugin name, and paste!

```
41 lines | webpack.config.js
... lines 1 - 29
30  .configureBabel((babelConfig) => {
31    if (Encore.isProduction()) {
32      babelConfig.plugins.push(
33        'transform-react-remove-prop-types'
34      );
35    }
36  })
... lines 37 - 41
```

Remove the stuff below. This is simpler anyways: *if* we're building for production, add this handy plugin.

We're not going to build for production right now, but to make sure we didn't break anything, go back to the terminal that runs encore, press Ctrl+C to stop it, then restart:



And... no errors! Later, when we execute `yarn run encore production`, the prop types won't be there.

# Chapter 16: Moving the Rep Logs to State

Communication always flows *down* in React: data lives in one component and is passed *down* to its children as props. Actually, both data *and* callbacks are passed from parent to child: child components use callbacks to communicate back up to the parent when something happens. For example, RepLogs passes an `onRowClick` to RepLogList. It uses that to tell its parent when that interaction occurs.

So, parents pass data *to* their children. But, parents do *not*, ask children for information. Well, it's technically possible, but it's not the normal flow.

For example, RepLogApp passes the `highlightedRowId` to RepLogs. But, RepLogApp does *not* ever ask RepLogs to give it any data that lives inside RepLogs. Information only flows down.

## Why state Lives up High

For that reason, in general, we need the state of our application to live as *high* up the component hierarchy as possible. Why? Because we can pass a piece of state *down* to all of the components that need to use it. When that state changes, that change naturally flows down and everything updates beautifully.

But, imagine if a piece of state lived in a *child* component, but we wanted to use it in the `render()` method of a parent. Well, that just won't work! The parent can't ask the child for that data: information does not flow up.

*This* is the reason why we will put *all* of our state in the top level component: RepLogApp. Again, this is not an absolute rule, but it's a great rule to follow for now. We'll talk later about when it's ok to move state lower, into a child component.

## Moving RepLogs to state

Anyways, the *most* important piece of data in our app is the rep logs themselves. And, these *will* need to change dynamically as the user adds new rep logs and deletes old ones. That means, rep logs need to be stored as *state*.

To get the static version of our app up and running, we just hardcoded these inside RepLogList. Time to move this to state! Copy the dummy rep log data and go to RepLogApp. Whenever we have new state, we need to initialize it in the constructor. Add a new `repLogs` key to this array and paste!

```
42 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 5
6   constructor(props) {
... lines 7 - 8
9     this.state = {
10       highlightedRowId: null,
11       repLogs: [
12         { id: 1, reps: 25, itemLabel: 'My Laptop', totalWeightLifted: 112.5 },
13         { id: 2, reps: 10, itemLabel: 'Big Fat Cat', totalWeightLifted: 180 },
14         { id: 8, reps: 4, itemLabel: 'Big Fat Cat', totalWeightLifted: 72 }
15       ]
16     };
... lines 17 - 18
19   }
... lines 20 - 42
```

Yea, eventually the `repLogs` state will start empty, and we'll then populate it by making an AJAX call to the server for the existing rep logs. But, until then, the dummy data makes building things easier.

## Passing the RepLogs State Down

The new state lives in the top-level component. But... we need to use it down in RepLogList. No problem! We just need to pass this down our tree. Fetch the `repLogs` out of state, then pass this as a prop to RepLogs.

42 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 24
25   render() {
... lines 26 - 28
29     return (
30       <RepLogs
... lines 31 - 33
34       repLogs={repLogs}
35     />
36   )
37 }
... lines 38 - 42
```

In RepLogs, before using the new prop, head down to the bottom: we want to define all props in propTypes. Add repLogs set to PropTypes.array.isRequired.

86 lines | [assets/js/RepLog/RepLogs.js](#)

```
... lines 1 - 79
80 RepLogs.propTypes = {
... lines 81 - 83
84   repLogs: PropTypes.array.isRequired
85 };
```

Copy that, because, RepLogList will receive the same prop.

30 lines | [assets/js/RepLog/RepLogList.js](#)

```
... lines 1 - 24
25 RepLogList.propTypes = {
... lines 26 - 27
28   repLogs: PropTypes.array.isRequired,
29 };
```

Ok! We *are* passing the repLogs prop to the RepLogs component. At the top of render(), read repLogs out of props. And then, do the prop-passing dance: send this straight into RepLogList.

86 lines | [assets/js/RepLog/RepLogs.js](#)

```
... lines 1 - 4
5 export default function RepLogs(props) {
6   const { withHeart, highlightedRowId, onClick, repLogs } = props;
... lines 7 - 12
13   return (
... lines 14 - 25
26     <RepLogList
... lines 27 - 28
29     repLogs={repLogs}
30   />
... lines 31 - 76
77   );
78 }
... lines 79 - 86
```

Finally, in that component, get repLogs out of props and... delete the hardcoded stuff.

30 lines | [assets/js/RepLog/RepLogList.js](#)

... lines 1 - 3

```
4 export default function RepLogList(props) {  
5   const { highlightedRowId, onClick, repLogs } = props;  
... lines 6 - 22  
23 }  
... lines 24 - 30
```

This is sweet! Move back to your browser and refresh! Hey hey! It's not broken! Check out the React dev tools, and look at the top RepLogApp component. Yep! You can see the repLogs state. Now... mess with it! Change the reps from 25 to 50.... boom! The UI on the child component updates instantly!

But, look back at RepLogApp, it has two pieces of state & one prop. And... it's passing *all* of that into its child as props. With a trick, we can be lazier, and do this automatically.

# Chapter 17: Smart Components & Spread Attributes

RepLogApp is a "smart", or "container" component: it holds state & logic... but no markup. Instead, smart components wrap themselves around a dumb component - like RepLogs - and *that* component renders all the elements.

Thanks to this pattern... a funny thing happens in smart components: you pretty much *always* want to pass *all* of your state and props into your child so that it can *actually* use them to render!

But, this can become tedious: each new piece of state needs to be initialized, destructured into a variable, then passed as a prop... with the same name. Lame! Let's use a shortcut.

## The Attribute Spread Operator

Delete all of the variables we destructured from state and props. Then, delete all of the props we're passing to RepLogs, except for the callback. Instead, here's the awesome part, use the spread operator: `{...this.props}` and `{...this.state}`.

```
38 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 24
25   render() {
26     return (
27       <RepLogs
28         {...this.props}
29         {...this.state}
30         onClick={this.handleClick}
31       />
32     )
33   }
... lines 34 - 38
```

That's it! Every prop and state is now being passed to the child as a prop. The only things we need to pass by hand are any callbacks we need.

Move over and refresh. No problems. Our smart component is getting simpler and simpler: it's *all* logic.

## Calculating the Total Weight Lifted

While we're here, it's time to finish one of our TODO's: fill in the total weight lifted column. This is great! I usually end up forgetting about my TODO's until I accidentally find them years later.

The value in this column will need to change whenever the repLogs state changes. But... the total weight should *not* be stored in state! Why? Simple! We can already calculate it by adding up all of the total weight values for the rep logs. No need to introduce more state: that would be duplication.

Go to RepLogs: there's the TODO! And here's the plan: loop over the repLogs and add up the weights. So, we need just a *little* bit of logic. If this component were a class, I'd probably add a new method to the class and put that logic there. But, darn! It's just a function! No problem: just go *above* the function and add... another function! Call it `calculateTotalWeightLifted()` with a `repLogs` argument: we will need to pass that in.

To do the calculation, I'll paste in some booooooring code: it loops over the repLogs, adds up the `totalWeightLifted` for each and... returns.

```

96 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 4
5 function calculateTotalWeightLifted(repLogs) {
6   let total = 0;
7
8   for (let repLog of repLogs) {
9     total += repLog.totalWeightLifted;
10  }
11
12  return total;
13 }
... lines 14 - 96

```

Copy the function name, move down to the TODO, and call it! `{calculateTotalWeightLifted()}` passing it `repLogs`. The `repLogs` live inside props, but we already destructured that into a variable.

```

96 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 26
27   <table className="table table-striped">
... lines 28 - 41
42     <tr>
... lines 43 - 44
45       <th>{calculateTotalWeightLifted(repLogs)}</th>
... line 46
47     </tr>
... line 48
49   </table>
... lines 50 - 96

```

Moment of truth: refresh! Boom! We have a total! Let's mess with the state, like change this to 200. Yes! It updates! The state change on `RepLogApp` causes both `RepLogApp` and `RepLogList` to re-render. When that happens, our code uses the *new* weights to calculate the *new* total. It's all very awesome.

## [Super-Fancy \(Confusing?\) ES6 Syntax](#)

This works *fine*. But, I'm going to write a new function that does this *same* calculation... but with some crazy, fancy syntax. Check this out:

`const calculateTotalWeightFancier = then repLogs => repLogs.reduce()`. Pass *this* another arrow function with two arguments: `total` and `log`. That callback will *return* `total + log.totalWeightLifted`. Start the reduce function with a 0 value.

```

97 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 13
14 const calculateTotalWeightFancier = repLogs => repLogs.reduce((total, log) => total + log.totalWeightLifted, 0);
... lines 15 - 97

```

Phew! Before we understand this madness, copy the new function name, move down, and paste! Find your browser - ah, the page is already reloading... and... it works!

```

97 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 42
43   <tr>
... lines 44 - 45
46     <th>{calculateTotalWeightFancier(repLogs)}</th>
... line 47
48   </tr>
... lines 49 - 97

```



This fancier function doesn't contain anything new. But wow, even for me, this is hard to understand. So, why are we doing this? Because, in the React world, you *will* see syntax like this. And I want you to be at least comfortable reading it.

Let's walk through it.

This creates a variable that is set to a function that accepts one argument: `repLogs`. Because the function doesn't have curly braces, it means the function *returns* the result of `repLogs.reduce()`. The `reduce()` function - which you may not be familiar with - *itself* accepts a callback function with two arguments. Once again, because that function doesn't have curly braces, it means that it *returns* `total + log.totalWeightLifted`.

If this makes your head spin, me too! Honestly, to me, this looks much more complex than the original. But, when you see things like this in blog posts or documentation, just break it down piece by piece: it's just boring code, dressed up in a different style. And if you like this syntax, cool! Go nuts.

# Chapter 18: Handling a Form Submit

Hey! Our repLogs live in state! And so, I think it's finally time to add some magic to our form and get it functional. Here's our *next* goal: when the user submits this form, we want to take its data and *update* the repLogs state so that a new row is rendered in the table.

The form itself lives in RepLogs, near the bottom. But, the state we need to modify lives in our parent: RepLogApp. To communicate back *up* the tree, we'll follow a familiar pattern: pass a *callback* from parent to child, just like we did with onRowClick.

## Adding the onSubmit Behavior

Start in RepLogApp: add the handler function: handleNewItemSubmit() with an event object. To prevent the form from *actually* trying to submit, use event.preventDefault(), just like normal JavaScript.

```
46 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 4
5   export default class RepLogApp extends Component {
... lines 6 - 24
25   handleNewItemSubmit(event) {
26     event.preventDefault();
... lines 27 - 29
30   }
... lines 31 - 41
42 }
... lines 43 - 46
```

For now, log some stuff! I love when a good form submits! Oh, and *also* log event.target. Because this function will handle the form element's submit, event.target will be the form itself. We're going to need that so we can read the values from its fields.

```
46 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 24
25   handleNewItemSubmit(event) {
... lines 26 - 27
28     console.log('I love when a good form submits!');
29     console.log(event.target);
30   }
... lines 31 - 46
```

Pass this callback as a new prop: onNewItemSubmit = {this.handleNewItemSubmit}. And, hey! We're starting to see a naming convention. This isn't anything official, but I like to name my *methods* "handleSomeEvent" and my *props* "onSomeEvent".

```
46 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 31
32   render() {
33     return (
34       <RepLogs
... lines 35 - 37
38       onNewItemSubmit={this.handleNewItemSubmit}
39     />
40   )
41 }
... lines 42 - 46
```

In RepLogs, head *straight* down to propTypes to describe the prop: onNewItemSubmit is a required function.

```
98 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 90
91   RepLogs.propTypes = {
... lines 92 - 94
95     onNewItemSubmit: PropTypes.func.isRequired,
... line 96
97   };
```

Love it! Back in render, destructure this into a variable. So: how can we attach a "submit" listener to the form? Ah... it's just onSubmit={onNewItemSubmit}.

```
98 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 15
16   export default function RepLogs(props) {
17     const { withHeart, highlightedRowId, onClick, repLogs, onNewItemSubmit } = props;
... lines 18 - 23
24     return (
... lines 25 - 51
52       <form className="form-inline" onSubmit={onNewItemSubmit}>
... lines 53 - 87
88     );
89   }
... lines 90 - 98
```

So simple! Go over to the browser and give it a nice refresh! Select an item... fill in a number and... we got it! Every time we submit by pressing enter or clicking the button, we see our insightful message. And as promised, the event.target that we're logging is *literally* the raw, form DOM element.

This is actually really nice. React *always* guarantees that event.target will be the element that you attached the listener to.

## [Reading the Form Data](#)

Next question! How can we read the values from our fields? Look at the form in RepLogs: there's the select element and... the text area. Check it out: it has a *name* attribute: reps. We can use that and normal, boring JavaScript to find that field and get its value.

By the way... if you've read a little bit about forms and React, this might not be what you were expecting. Don't worry. I'm going to show you a *few* different ways to get the values from form fields, including the pros and cons of each, and which method I recommend and when.

But right now, forget about React, and remember that, under the hood, there is a boring HTML form sitting on the page that we can interact with.

In RepLogApp, it's time to flex our native JavaScript muscles! To read the reps textarea, use event.target - that's the form - .elements.namedItem('reps'). This will give us the text element. Reads its value with .value.

```
46 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 24
25   handleNewItemSubmit(event) {
... lines 26 - 28
29     console.log(event.target.elements.namedItem('reps').value);
30   }
... lines 31 - 46
```

Let's go try it! Move over, refresh... select "My Laptop" and lift it 50 times. Yes! There's the 50! Victory!

## [Keep your Smart Component Unaware of Markup](#)

But, before we go further, I need to ask an important philosophical question:

If your shirt isn't tucked into your pants, are your pants tucked into your shirt?

Hmm. Thought provoking. And also: if our smart component - RepLogApp - should not be responsible for rendering *any* HTML, should its handleNewItemSubmit() method be aware that there is an HTML form and a field with a name="reps" attribute inside?

Actually... no! It makes no sense for handleNewItemSubmit() to suddenly be aware of a specific HTML structure that's rendered by its child. In fact, *all* RepLogApp should care about is that, when - *somehow* - a new rep log is created in the app, its handleNewItemSubmit() function is called so that it can update the repLogs state. If it's created with a form, or with some random fields during a 10-step process or just with black magic... RepLogApp should not care!

So, check this out: copy the inside of the function: I'm going to move most of this callback into RepLogs as a *new* handler function. *Inside* render(), add a new function: handleFormSubmit() with our normal event argument. Then, paste the logic.

```
105 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 23
24   function handleFormSubmit(event) {
25     event.preventDefault();
26
27     console.log('I love when a good form submits!');
28     console.log(event.target.elements.namedItem('reps').value);
29   }
... lines 30 - 105
```

Down in onSubmit, instead of calling the parent handler, call the new function: handleFormSubmit.

```
105 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 30
31   return (
... lines 32 - 58
59     <form className="form-inline" onSubmit={handleFormSubmit}>
... lines 60 - 94
95   );
... lines 96 - 105
```

Yep, this feels *much* better. handleFormSubmit() is responsible for calling event.preventDefault() and uses the form structure - which is created right inside this component - to read the names of the fields. Finally, at the bottom, call the parent handler: onNewItemSubmit().

```
107 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 23
24   function handleFormSubmit(event) {
... lines 25 - 29
30     onNewItemSubmit('Big Fat Cat', event.target.elements.namedItem('reps').value);
31   }
... lines 32 - 107
```

Actually, *this* is the reason why I put the new function *inside* of render() instead of above the function like I did with calculateTotalWeightFancier(): our callback needs access to the props.

Here's the last important part: instead of passing the event object or the form element to the parent onNewItemSubmit() callback, only pass it what it needs: the new rep log's raw data. For now, hardcode an item name - "Big fat cat" - but copy the number of true rep logs and paste.

110 lines | [assets/js/RepLog/RepLogs.js](#)

... lines 1 - 24

```
25   function handleFormSubmit(event) {  
    ... lines 26 - 30  
31     onNewItemSubmit('Big Fat Cat', event.target.elements.namedItem('reps').value);  
32   }  
    ... lines 33 - 110
```

Back in RepLogApp, clear out handleNewItemSubmit and give it two fresh args: itemName and reps. Log a todo below: we will eventually use this to update the state. And log those values so we can check things!

44 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 24

```
25   handleNewItemSubmit(itemName, reps) {  
26     console.log('TODO - handle this new data');  
27     console.log(itemName, reps);  
28   }  
    ... lines 29 - 44
```

I love it! RepLogApp still has a callback, but it's now unaware of the form. It doesn't care *how* rep logs are created, it only cares that its callback is executed when that happens. All the form logic is *right* where it should be.

Try it out! Refresh the page, select an item, enter 45 and... submit! The Big fat cat is hardcoded, but the 45 is our real data.

As *simple* as it is to read the values of the fields by using the name attribute, you probably won't do this in practice. Instead, we'll learn two other ways: refs & state. We'll jump into refs next.

# Chapter 19: New Component to Hold our Form

Inside RepLogs, this form logic is starting to get pretty big: we have a handler function and most of the markup comes from the form! And this is only going to get *more* complex when we finish the form stuff *and* add future things like validation.

So, to reduce complexity, I think it's time to take this form and put it into its own *new* component. Remember: our app starts with RepLogApp on top: it's our smart, good-looking, fun-loving, stateful component. Then, it renders a dumb, tries-its-best, presentation component that holds the markup. That setup is important, and in a small app, it may be all you need! But, if RepLogs gets too big or confusing, you can *choose* to break things down even more.

## Creating RepLogCreator

Inside the RepLog directory, create a new file called, how about, RepLogCreator.js. This will be another dumb, presentation component. And so, it can be a function, like RepLogList. Copy its import statements, paste, and export default function RepLogCreator with the normal props arg. Get things working by returning a hardcoded div!

I'm going to be a form when I grow up!

```
9 lines | assets/js/RepLog/RepLogCreator.js
1  import React from 'react';
2  import PropTypes from 'prop-types';
3
4  export default function RepLogCreator(props) {
5    return (
6      <div>I'm going to be a form when I grow up!</div>
7    );
8  }
```

Love it! Back in RepLogs, on top, import RepLogCreator from ./RepLogCreator. Then, down in render, above the form, use <RepLogCreator />.

```
110 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 3
4  import RepLogCreator from './RepLogCreator';
... lines 5 - 16
17 export default function RepLogs(props) {
... lines 18 - 33
34   return (
... lines 35 - 61
62     <RepLogCreator/>
... lines 63 - 99
100  );
101  }
... lines 102 - 110
```

Ok, let's go check it out! So far, so good.

Next, copy *all* of the form markup, delete it, go to RepLogCreator and... paste!

```

43 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 3
4   export default function RepLogCreator(props) {
5     return (
6       <form className="form-inline" onSubmit={handleFormSubmit}>
... lines 7 - 39
40   </form>
41   );
42 }

```

That *looks* cool... but, come on. We know that nothing ever works on the first try. Try it - yep... we are rewarded with a nice big error!

handleFormSubmit is not defined

coming from RepLogCreator line 6: it's our onSubmit! I totally forgot about that! Go grab it from RepLogs - and, by the way - check out how *small* this component is looking - then, inside RepLogCreator, paste.

```

52 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   function handleFormSubmit(event) {
6     event.preventDefault();
7
8     console.log('I love when a good form submits!');
9     console.log(event.target.elements.namedItem('reps').value);
10
11     onNewItemSubmit('Big Fat Cat', event.target.elements.namedItem('reps').value);
12   }
... lines 13 - 52

```

The *last* missing piece is the onNewItemSubmit() callback: this is passed from RepLogApp to RepLogs. And now, we need to pass it once again to RepLogCreator. Because we need a new prop, define it first at the bottom in propTypes: RepLogCreator.propTypes = an object and... go steal this code from RepLogs.

```

58 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 54
55  RepLogCreator.propTypes = {
56    onNewItemSubmit: PropTypes.func.isRequired,
57  };

```

Excellent! Now that we are *requiring* this prop, head back up to render() and destructure it: const { onNewItemSubmit } = props.

```

58 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 3
4   export default function RepLogCreator(props) {
5     const { onNewItemSubmit } = props;
... lines 6 - 52
53   }
... lines 54 - 58

```

Cool! *Finally*, in RepLogs, nice! PhpStorm is already telling us that we're missing a required prop: pass this onNewItemSubmit={onNewItemSubmit}.

67 lines | assets/js/RepLog/RepLogs.js

```
... lines 1 - 24
25   return (
... lines 26 - 52
53     <RepLogCreator
54       onNewItemSubmit={onNewItemSubmit}
55     />
... line 56
57   );
... lines 58 - 67
```

And... we're done! Probably... Let's go find out: refresh. The form renders... we can select something and... it *does* print. Awesome!

## Introducing: Refs

And *now*.... it's time to start *bending* some of the nice rules that we've been talking about. RepLogCreator is a "dumb" component. It's like a template: its main job is to render markup, *not* to contain state or a lot of logic. Because "dumb", presentation components only really render things, we usually create them as *functions* instead of a class... just because we can and we're lazy!

Well... that's a good rule, but it's not always true. Right now, our handler function uses the *name* attribute of the input element to get a reference to the underlying DOM element. Then, it reads its value. It turns out that React has its *own* system for allowing you to reference the corresponding DOM element for *any* of our React elements. It's called "refs".

## Refactoring our Dumb Component to a Class

There are a few ways to use this "refs" system. But, the recommended way requires your component to be a *class*. And, that's ok! There *are* some legitimate situations where a dumb component needs a class. Refs are one of those.

No big deal: let's change this to a class! First, we need to also import Component from React. Then, export default class RepLogCreator extends Component. And, of course, we *now* need to put all of this inside a render() function. Let's indent everything one level, and close the function.

60 lines | assets/js/RepLog/RepLogCreator.js

```
1  import React, { Component } from 'react';
... lines 2 - 3
4  export default class RepLogCreator extends Component {
5    render() {
6      const { onNewItemSubmit } = props;
... line 7
8      function handleFormSubmit(event) {
... lines 9 - 14
15     }
... line 16
17     return (
18       <form className="form-inline" onSubmit={handleFormSubmit}>
... lines 19 - 51
52     </form>
53   );
54 }
55 }
... lines 56 - 60
```

Yep! Webpack is happy! Now that we have a proper class, we don't need to put handleFormSubmit() *inside* of render() anymore. Nope, we can access the props from anywhere as this.props.

So, copy that function & the const, paste it in the class, turn it into a property, and move the const *into* the function. Oh, and try not to mess up the syntax like I just did.



```

60 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 5
6   handleFormSubmit(event) {
7     event.preventDefault();
8     const { onNewItemSubmit } = this.props;
9
10    console.log('I love when a good form submits!');
11    console.log(event.target.elements.namedItem('reps').value);
12
13    onNewItemSubmit('Big Fat Cat', event.target.elements.namedItem('reps').value);
14  }
... lines 15 - 60

```

Better! Back in render(), now we'll call this.handleFormSubmit.

```

60 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 15
16  render() {
17    return (
18      <form className="form-inline" onSubmit={this.handleFormSubmit}>
... lines 19 - 52
53    );
54  }
... lines 55 - 60

```

Let's go check it out! Head back to the browser refresh! It loads... and when you submit... error! Woh! And then the page reloaded! Oh no!

There are *two* evil problems working together against us! For the first, my bad! We need to make sure that event.preventDefault() is always the *first* line in a handle function. You'll see why when we refresh and try the form again.

Ah yes: here is the *real* error:

Cannot read property props of undefined

coming from line 7. We know this problem: we forgot to *bind* our handler function to this... so this is *not* our RepLogCreator instance. When the page refreshed, it was because this error killed our code even *before* we called event.preventDefault().

We know the fix: *whenever* you have a handler function that's a property on your class, we need to create a constructor, call super(props), then bind that function with this.handleFormSubmit = this.handleFormSubmit.bind(this).

```

65 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   constructor(props) {
6     super(props);
7
8     this.handleFormSubmit = this.handleFormSubmit.bind(this);
9   }
... lines 10 - 65

```

That should do it! Move back, refresh, fill in the form and... yes! Our app logs correctly again. It's time to use refs to finish our form logic so we can update the repLogs state.

# Chapter 20: Refs

Right now, we're using the name attribute of each form field to get the underlying DOM element. We use *that* to fetch its value.

This is interesting: *most* of the time in React, you communicate down to your elements - and your child components - via props. You use props when you render the element objects, and React handles creating the *real* DOM elements from that. The DOM is *not* something we normally touch directly.

But occasionally, you *will* want to access the underlying DOM elements. For example, you might want to read a value from a form field, call focus() on an element, trigger media playback if you're rendering a video tag or integrate with a third-party JavaScript library that needs you to pass it a DOM element.

## Creating the refs

Because these are *totally* valid use-cases, React gives us a *great* system to access any DOM element called refs. We need to access two elements: the select element and the input. Cool! In the constructor, create 2 new properties: this.quantityInput = React.createRef() and this.itemSelect = React.createRef().

```
70 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   constructor(props) {
... lines 6 - 7
8     this.quantityInput = React.createRef();
9     this.itemSelect = React.createRef();
... lines 10 - 11
12  }
... lines 13 - 70
```

This just, "initialized" these two properties. The real magic is next: on the select, replace the name attribute with ref={this.itemSelect}. Do the same thing on the input: move the props onto their own lines, then add ref={this.quantityInput}.

```
70 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 24
25  render() {
26    return (
... lines 27 - 32
33      <select id="rep_log_item"
34            ref={this.itemSelect}
... lines 35 - 51
52      <input type="number" id="rep_log_reps"
53            ref={this.quantityInput}
... lines 54 - 62
63    );
64  }
... lines 65 - 70
```

To really *get* what this does, you need to see it. Comment out the onNewItemSubmit() call for a minute: it's temporarily broken. Then, let's console.log(this.quantityInput) and also this.itemSelect.

```

70 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 13
14   handleFormSubmit(event) {
... lines 15 - 18
19     console.log(this.quantityInput);
20     console.log(this.itemSelect);
... line 21
22     //onNewItemSubmit('Big Fat Cat', event.target.elements.namedItem('reps').value);
23   }
... lines 24 - 70

```

Moment of truth! Move over, Encore already refreshed the page. Fill out the fields, hit enter... cool! Each "ref" is an *object* with one property called *current* that is *set* to the underlying DOM element! Yea, I know, the fact that it sets the DOM element to a current key is a little weird... but it's just how it works.

## Using the Refs

Thanks to this, let's set the DOM element objects onto two new variables: `const quantityInput = this.quantityInput.current` and `const itemSelect = this.itemSelect.current`. Below, log the field values: `quantityInput.value` and, this is a bit harder, `itemSelect.options[itemSelect.selectedIndex].value`, then, `.value`.

```

73 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 13
14   handleFormSubmit(event) {
... lines 15 - 17
18     const quantityInput = this.quantityInput.current;
19     const itemSelect = this.itemSelect.current;
... lines 20 - 21
22     console.log(quantityInput.value);
23     console.log(itemSelect.options[itemSelect.selectedIndex].value);
... lines 24 - 25
26   }
... lines 27 - 73

```

This finds *which* option is selected, then returns its value attribute. Try it: refresh, select "Big Fat Cat", enter 50 and... boom! People, this is *huge*! We can *finally* pass *real* information to the callback. Uncomment `onNewItemSubmit`. Pass the options code, but, change to `.text`: this is the *display* value of the option. And, until we *actually* starting saving things via AJAX, *that* is what we'll pass to the callback. Next, use `quantityInput.value`.

```

72 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 13
14   handleFormSubmit(event) {
... lines 15 - 20
21     onNewItemSubmit(
22       itemSelect.options[itemSelect.selectedIndex].text,
23       quantityInput.value
24     );
25   }
... lines 26 - 72

```

## Updating the repLogs State

*Finally*, go back to `RepLogApp` and find `handleNewItemSubmit`: *this* is the function we just called. I'm going to change the argument to `itemLabel`, then clear things out. Ok, our job here is simple: add the new rep log to the `repLogs` state. Read the existing state with `const repLogs = this.state.repLogs`. Then, create the `newRep` set to an object. This needs the same properties as the other rep logs. So, add `id` set to... hmm. We don't have an `id` yet! Set this to "TODO". Then, `reps`: `reps`, `itemLabel`: `itemLabel` and, the last field is `totalWeightLifted`. Hmm, this is another tricky one! Our React app doesn't know how "heavy" each item is... and so we don't know the `totalWeightLifted`! Later, we'll need to ask the server for this info. For now,

just use a random number.

```
51 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 24
25   handleNewItemSubmit(itemLabel, reps) {
26     const repLogs = this.state.repLogs;
27     const newRep = {
28       id: 'TODO-id',
29       reps: reps,
30       itemLabel: itemLabel,
31       totalWeightLifted: Math.floor(Math.random() * 50)
32     };
... lines 33 - 34
35   }
... lines 36 - 51
```

And finally, let's update the state! `repLogs.push(newRep)` and `this.setState()` with `repLogs` set to `repLogs`.

```
51 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 24
25   handleNewItemSubmit(itemLabel, reps) {
... lines 26 - 32
33     repLogs.push(newRep);
34     this.setState({repLogs: repLogs});
35   }
... lines 36 - 51
```

Um... there *is* a teeny problem with *how* we're updating the state here. But, we'll talk about it next. For now, gleefully forget I said *anything* was wrong and refresh! Fill out the form and... boo! A familiar error:

Cannot read property state of undefined in RepLogApp line 26

I've been *lazy*. *Each* time we create a handler function in a class, we need to *bind* it to this! In the constructor, add `this.handleNewItemSubmit = the same thing .bind(this)`.

```
52 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 5
6   constructor(props) {
... lines 7 - 18
19     this.handleNewItemSubmit = this.handleNewItemSubmit.bind(this);
20   }
... lines 21 - 52
```

## Using uuids

Try it again! We got it! It updates the state and *that* causes React to re-render and add the row. But... if we try it a second time, it *does* update the state, but, ah! It yells at us:

Encountered two children with the same key

Ah! The `id` property is eventually used in `RepLogList` as the `key` prop. And with the hardcoded `TODO`, it's not unique. Time to fix that temporary hack.

But, hmmm. How *can* we get a unique `id`? There are always two options. First, you can make an AJAX request and wait for the server to send back the new `id` before updating the state. We'll do that later. *Or*, you can generate a `uuid` in JavaScript. Let's do that now. And later, when we start talking to the server via AJAX, we'll discuss how `UUIDs` *can* still be used, and are a great idea!

To generate a `UUID`, find your terminal and install a library:

```
$ yarn add uuid --dev
```

## Tip

In the latest version of uuid, you should import the uuid package like this:

```
import { v4 as uuid } from 'uuid';
```

Wait for that to finish... then go to RepLogApp and import uuid from uuid/v4. There are a few versions of UUID that behave slightly differently. It turns out, we want v4.

Down in constructor(), use UUID's everywhere, even in our dummy data. Then, move to the handle function and use it there.

53 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 3
4   import uuid from 'uuid/v4';
... lines 5 - 6
7   constructor(props) {
... lines 8 - 9
10    this.state = {
... line 11
12    repLogs: [
13      { id: uuid(), reps: 25, itemLabel: 'My Laptop', totalWeightLifted: 112.5 },
14      { id: uuid(), reps: 10, itemLabel: 'Big Fat Cat', totalWeightLifted: 180 },
15      { id: uuid(), reps: 4, itemLabel: 'Big Fat Cat', totalWeightLifted: 72 }
16    ]
17  };
... lines 18 - 20
21  }
... lines 22 - 26
27  handleNewItemSubmit(itemLabel, reps) {
... line 28
29    const newRep = {
30      id: uuid(),
... lines 31 - 33
34    };
... lines 35 - 36
37  }
... lines 38 - 53
```

Let's see if this fixes things! Move over, make sure the page is refreshed and start adding data. Cool: we can add as many as we want.

## Clearing the Form

Which... is actually kinda weird: when the form submits, we need the fields to reset. No problem: in RepLogCreator, in addition to *reading* the values off of the DOM elements, we can also *set* them. At the bottom, use `quantityInput.value = ''` and `itemSelect.selectedIndex = 0`.

75 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 13

```
14   handleFormSubmit(event) {
```

... lines 15 - 25

```
26     quantityInput.value = "";
```

```
27     itemSelect.selectedIndex = 0;
```

```
28   }
```

... lines 29 - 75

Try it! Refresh... fill in the form and... sweet! Whenever you need to work directly with DOM elements, refs are your friend.

# Chapter 21: Immutability / Don't Mutate my State!

Ok... so.... there's this annoying... but super important rule in React that we're *totally* violating. The rule is: the *only* time you're allowed to set or change the state property directly is when you're initializing it in the constructor. *Everywhere* else, you *must* call `setState()` instead of changing it directly.

Here's another way to say it: each piece of data on `this.state` should be *immutable*. And *that* is the part we're violating. It was really subtle! First, unlike PHP, in JavaScript arrays are *objects*. And so, like all objects, if you modify `repLogs`, that also modifies `this.state.repLogs` because... they're the same object!

And that's *exactly* what we did when we called `repLogs.push`: this changed, or *mutated*, the `repLogs` key on `this.state`! Yep! We *changed* the state before calling `this.setState()`.

## Do I really need to Avoid Mutating State?

Now, is that *really* a problem? I mean, everything seems to work. Basically... yes, but, honestly, it's subtle. There are two problems with mutating the state. First, `setState()` is actually asynchronous: meaning, React doesn't handle your state change immediately. For example, if two parts of your code called `setState()` at almost the same moment, React would process the first state change, re-render React, and *then* process the second state change. Because of this, if you mutate the state accidentally, it's possible that it will get overwritten in a way you didn't expect. It's unlikely, but we're trying to avoid a WTF moment.

The second reason is that if you mutate your state, it may prevent you from making some performance optimizations in the future.

Honestly, when you're learning React, the reasons for "why" you shouldn't mutate your state are hard to understand. The point is: you should *avoid* it, and we'll learn how. Well, if you're updating a scalar value like `highlightedRowId`, it's simple! But when your state is an object or an array, which, *is* an object, it's harder.

If you need to "add" to an array without, updating it, here's how: `const newRepLogs =` , create a *new* array, use `...this.state.repLogs` to put the existing `repLogs` into it and then, add `newRep`. Yep, this is a *new* array: we did *not* change state. This solves our problem.

52 lines | `assets/js/RepLog/RepLogApp.js`

```
... lines 1 - 26
27   handleNewItemSubmit(itemLabel, reps) {
... lines 28 - 33
34     const newRepLogs = [...this.state.repLogs, newRep];
35     this.setState({repLogs: newRepLogs});
36   }
... lines 37 - 52
```

## Using the `setState()` Callback

Except... there is *one* other tiny, annoying rule. Most of the time, when you set state, you set it to some new, specific value. But, if the *new* state *depends* on the old state - like our new `repLogs` depends on the current `repLogs` - then you need to use `setState()` as a callback.

Check it out: call `this.setState()`, but instead of passing data, pass a callback with a `prevState` argument. Inside, create the array: `const newRepLogs = [...prevState.repLogs, newRep]`, and *return* the new state: `repLogs` set to `newRepLogs`.

56 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 26

```
27   handleNewItemSubmit(itemLabel, reps) {  
    ... lines 28 - 34  
35     this.setState(prevState => {  
36       const newRepLogs = [...prevState.repLogs, newRep];  
37  
38       return {repLogs: newRepLogs};  
39     })  
40   }
```

... lines 41 - 56

Why the heck are we doing this? Remember how I said that `setState()` is asynchronous? Because of that, if you call `setState()` now, React may not *use* that state until a few milliseconds later. And, if something *else* added a new `repLog` between now and then... well... with our previous code, our new state would override and *remove* that new `repLog`!

I know, I know! Oof, again, it's subtle and probably won't bite you, and you'll probably see people skip this. To keep it simple, just remember the rule: *if* setting new state involves you *using* data on `this.state`, pass a callback instead. Then, you'll *know* you're safe.

### [Smarter Method & Prop Names](#)

While we're here, something is bothering me. Our callback method is named `handleNewItemSubmit()`. But... we purposely designed `RepLogApp` so that it doesn't know or care that a form is being used to create rep logs. So let's rename this method: `handleAddRepLog()`.

56 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 26

```
27   handleAddRepLog(itemLabel, reps) {  
    ... lines 28 - 39  
40   }
```

... lines 41 - 56

Yea. Make sure to also update the `bind()` call in the constructor. Below, when we pass the prop - update it here too. But... I think we should also rename the prop: `onAddRepLog()`.

56 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 6

```
7   constructor(props) {  
    ... lines 8 - 19  
20     this.handleAddRepLog = this.handleAddRepLog.bind(this);  
21   }
```

... lines 22 - 41

```
42   render() {  
    ... line 43
```

```
44     <RepLogs
```

... lines 45 - 47

```
48       onAddRepLog={this.handleAddRepLog}
```

```
49     />
```

... line 50

```
51   }
```

... lines 52 - 56

And, if we change that, we need to update a few other spots: in `RepLogs`, change the `propType`. And, up where we destructure, PhpStorm is highlighting that this prop doesn't exist anymore. Cool! Change it to `onAddRepLog`, scroll down, and make the same change `onAddRepLog={onAddRepLog}`.



```

67 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 16
17 export default function RepLogs(props) {
18   const { withHeart, highlightedRowId, onClick, repLogs, onAddRepLog } = props;
... lines 19 - 24
25   return (
... lines 26 - 52
53     <RepLogCreator
54       onAddRepLog={onAddRepLog}
55     />
... line 56
57   );
58 }
... line 59
60 RepLogs.propTypes = {
... lines 61 - 63
64   onAddRepLog: PropTypes.func.isRequired,
... line 65
66 };

```

Repeat this process in RepLogCreator: rename the propType, update the variable name, and use the new function.

```

75 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 13
14   handleSubmit(event) {
... line 15
16     const { onAddRepLog } = this.props;
... lines 17 - 20
21     onAddRepLog(
... lines 22 - 23
24   );
... lines 25 - 27
28 }
... lines 29 - 75

```

Oh, also, in RepLogs, the destructuring line is getting *crazy* long. To keep me sane, let's move each variable onto its own line.

```

73 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 16
17 export default function RepLogs(props) {
18   const {
19     withHeart,
20     highlightedRowId,
21     onClick,
22     repLogs,
23     onAddRepLog
24   } = props;
... lines 25 - 63
64 }
... lines 65 - 73

```

### Moving the "itemOptions" onto a Property

Finally, we need to make one other small change. In RepLogCreator, all of our options are hardcoded. And, that's not necessarily a problem: we'll talk later about whether or not we should load these dynamically from the server.

But, to help show off some features we're about to work on, we need to make these a *little* bit more systematic. In the constructor, create a new property: `this.itemOptions` set to a data structure that represents the 4 items.

```
81 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   constructor(props) {
... lines 6 - 10
11   this.itemOptions = [
12     { id: 'cat', text: 'Cat' },
13     { id: 'fat_cat', text: 'Big Fat Cat' },
14     { id: 'laptop', text: 'My Laptop' },
15     { id: 'coffee_cup', text: 'Coffee Cup' },
16   ];
... lines 17 - 18
19 }
... lines 20 - 81
```

Notice, I'm not making this props or state: we don't need these options to actually *change*. Nope, we're just taking advantage of the fact that we have a class, so, if we want to, we can store some data on it.

Back in `render()`, delete the 4 options and replace it with one of our fancy map structures: `this.itemOptions.map()` with an `item` argument. In the function, return an `<option>` element with `value={option.id}`, `key={option.id}` - we need that for any array of elements - and, for the text, use `{option.text}`.

```
81 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 36
37 render() {
38   return (
... lines 39 - 44
45     <select id="rep_log_item"
... lines 46 - 51
52       {this.itemOptions.map(option => {
53         return <option value={option.id} key={option.id}>{option.text}</option>
54       })}
55     </select>
... lines 56 - 73
74   );
75 }
... lines 76 - 81
```

Nice! Let's make sure it works - refresh! It works and... yea - the options are still there.

When we submit... woh! All our state disappears! This smells like a Ryan bug, and it will be *something* wrong with how we're setting the state. Ah, yep! This should be `prevState.repLogs`.

Ok, try it again. Refresh, fill out the form and... we're good!

Let's talk about some validation!

# Chapter 22: Dumb Components with State

Our form works... but has no validation. Well, that's not *completely* true. In general, there are *three* types of validation. First, server-side validation, which we absolutely need and will talk about later. Second, client-side validation via JavaScript, which is optional, but a nice way to give quick feedback. And third, client-side HTML5 validation, which isn't flexible at all, but is *super* easy to add. In fact, we already have some: the required attributes on the form: these prevent the user from submitting the form empty.

But, right now, you *can* enter a negative number. And... that makes no sense! So let's add some client-side validation in our React app to prevent this. Oh, and by the way, the input field *already* disallows entering letters - that's another HTML5 validation feature thanks to this `type="number"`.

## Preventing the Form Submit

The form handling-logic lives in `RepLogCreator.handleFormSubmit()`. So, if we want to prevent the form from submitting or add an error message, that code needs to live here.

And the first half is easy! If `quantityInput.value <= 0` then... somehow, we need to add an error message. And because we *don't* want to continue processing, just return.

```
88 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 20
21   handleFormSubmit(event) {
... lines 22 - 27
28     if (quantityInput.value <= 0) {
29       // TODO - print some validation error!
30
31       // don't submit, or clear the form
32       return;
33     }
... lines 34 - 41
42   }
... lines 43 - 88
```

Let's see how this looks so far. Try negative 10, select an item and... yep! The form does not clear.

## Dumb Components with State

Great! Now, how can we add an error message? Think about it: *sometimes* our form will need an error message and sometimes it will *not*. The error is a piece of data that we need to *change*. In other words, it needs to be stored as state!

Ya! If we had an "error message" state, we could use that down in `render()`. And then, whenever the state changed, React would re-render this component and print out the new message.

But, hmm. Right now, *all* our state lives in the top-level component: `RepLogApp`. That's on purpose! `RepLogApp` is a smart, stateful component. And, because it holds *all* of the state, all the other components can be dumb, *stateless* components that render markup with little or no logic.

This is a *good* distinction. But, in the real world, there *are* some situations when a dumb, presentation component - like `RepLogCreator` - *should* hold some state. This is one of them!

Why? Well, `RepLogApp`'s job is to be worried about the *business* logic of our app, independent of markup. So, it keeps track of things like the `repLogs`. But, a form validation error is not *really* business logic: it's state that just exists to support the form's user interface. Heck, as I keep mentioning, `RepLogApp` isn't even aware that our app has a form!

This was a hard distinction for me to *fully* understand. So, here's a different explanation, entirely stolen from our brave co-author Frank:

RepLogCreator is concerned about the creation process. It's like a bouncer at the club and the input field is the front door. The input only gets into the club if it meets certain criteria. By handling that logic in RepLogCreator, we allow the rest of our application to be unaware of this: it's taken care of for them. It also prevents RepLogApp - the manager of the club - from *needing* to know how RepLogCreator is doing its job.

Here's the point: I want your dumb components to, at first, *not* have state. But if you *do* need some state in order to power the user interface for that component, that's totally fine. And if you're *totally* confused, don't sweat it. If you *do* put your state in the wrong place, you'll either realize it eventually, or it'll just mean a bit more work for you. It's not the end of the world, nothing is permanent.

So, let's give RepLogCreator some state!

# Chapter 23: Form Validation State

When the user tries to submit a negative quantity, we need to show an error! On a React level, this means that we'll need new state to store that message. And, like we just discussed, because this state is *really* tied to our form and is meant to power the UI, the best place for it is inside RepLogCreator.

To add state, we always start the same way: initialize it in the constructor: `this.state = an object`, and call it `quantityInputError` set to empty quotes.

```
99 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   constructor(props) {
... lines 6 - 7
8     this.state = {
9       quantityInputError: ""
10    };
... lines 11 - 22
23  }
... lines 24 - 99
```

Below, remove the `todo` and add `this.setState()` with `quantityInputError` set to `Please enter a value greater than zero`.

```
100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 24
25  handleFormSubmit(event) {
... lines 26 - 31
32    if (quantityInput.value <= 0) {
33      this.setState({
34        quantityInputError: 'Please enter a value greater than 0'
35      });
... lines 36 - 38
39    }
... lines 40 - 50
51  }
... lines 52 - 100
```

And when the form submit *is* successful, we need to make sure any existing error is removed. Copy the `setState()` line and set it to an empty string.

```
100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 24
25  handleFormSubmit(event) {
... lines 26 - 47
48    this.setState({
49      quantityInputError: ""
50    });
51  }
... lines 52 - 100
```

State, check! Down in render, as always, start by destructuring the variable: `const { quantityInputError } = this.state`. Let's use this to do *two* things: add a class to make the form element look red *and* print the message.

100 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 52

```
53   render() {
54     const { quantityInputError } = this.state;
    ... lines 55 - 93
94   }
    ... lines 95 - 100
```

Because we're using Bootstrap, to make the field red, the form-group div needs a new has-error class. Empty out className, enter into JavaScript mode and use "ticks" to use a "template literal". This makes using multiple classes with logic a bit easier: re-add form-group, then type `${}` to do "string interpolation". Oooo. Inside, *if* quantityInputError then print has-error, else print nothing.

100 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 55

```
56   return (
    ... lines 57 - 75
76     <div className={`form-group ${quantityInputError ? 'has-error' : ''}>
    ... lines 77 - 86
87     </div>
    ... lines 88 - 92
93   );
    ... lines 94 - 100
```

## [The Cool Conditional String Printing Syntax](#)

And to actually print the message, go down after the input. Here, I don't just want to print the error, I want to surround it in a red div, but *only* if there actually *is* an error. We could use the ternary syntax here... and use some inline JSX. That's fine.

But instead, I want to show you a cool, but weird, shortcut syntax that we can use *whenever* we want to print a string *only* when that string is not empty. Here it is: quantityInputError && and then the JSX: `<span className="help-block">`, print quantityInputError, close the tag, and exit from JavaScript mode.

100 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 75

```
76     <div className={`form-group ${quantityInputError ? 'has-error' : ''}>
    ... lines 77 - 85
86         {quantityInputError && <span className="help-block">{quantityInputError}</span>}
87     </div>
    ... lines 88 - 100
```

Woh. Before we talk about this, try it! Move over, make sure the page is *fully* refreshed, select an item, be annoying and use a negative number and... there it is! Oh, it's ugly: we'll fix that soon.

But first, about this syntax! It's weird because this bit of code works *different* in JavaScript versus PHP! In JavaScript, if quantityInputError is empty, or "falsey", this return false and we print nothing. But if quantityInputError has some text in it, so, it's "truthy", then JavaScript returns the *second* part of the expression: our JSX. So, this entire block will either return false or this JSX. In PHP, this would always return false or true.

So... yes, this is another fancy syntax. If you love it, nice! If you hate it, use the ternary syntax instead.

## [Cleaning up the Form Design](#)

Before we move on, oof, we *need* to make this look less ugly. Go back into RepLogs and scroll down. Add a few divs: className="row" and another className="col-md-6". Move RepLogCreator inside.

```

77 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 30
31   return (
... lines 32 - 58
59     <div className="row">
60       <div className="col-md-6">
... lines 61 - 63
64     </div>
65   </div>
... line 66
67   );
... lines 68 - 77

```

Then, back in RepLogCreator, find the form element and... remove the form-inline class.

```

100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 55
56   return (
57     <form onSubmit={this.handleFormSubmit}>
... lines 58 - 92
93   );
... lines 94 - 100

```

Give your browser a nice refresh. This puts the form elements onto their own lines. And *that* makes our validation error look *much* nicer.

We'll handle server-side validation later. But right now, let's talk about a totally *different* way of handling form data: controlled components.

# Chapter 24: Controlled Form Input

I have good news and bad news. The bad news? React has *two* totally different options for interacting with forms. And *nobody* likes extra choices. The good news? We *are* going to learn both, but then I'll tell you *exactly* which choice *I* think you should make, and when.

The first option is what we've been doing so far: you render the form field then interact with the DOM element directly to read or set its value. In this world, React is basically unaware that this field exists after it's rendered.

The *second* option is quite different. When you render the field, you *bind* its value to a piece of *state*. Then, instead of working with the DOM element to read and set its value, you read and set that *state*. And, of course, when you update the state, React re-renders the field with the new value.

In the first approach, the *DOM* is the source of truth for the value of a field. In the second approach, the *state* is the source of truth.

## Adding the Field State

To show off the second approach, we're going to add a *really* important new feature: a form field at the top of the page where the user can control the *number* of hearts they want to see. If they enter 50, we'll heart them 50 times!

In RepLogs, right after the h1, add a simple `<input type="number" />`.

```
79 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 30
31   return (
32     <div className="col-md-7">
... lines 33 - 34
35       <input type="number" />
... lines 36 - 67
68     </div>
69   );
... lines 70 - 79
```

Nothing interesting yet. Yep, hello, new, empty field. Next, we need to know how *many* hearts the user wants. That means we need some new *state*. We *could* put that state inside RepLogs. After all, the number of hearts is a pretty not-important, UI-related state. But, to keep RepLogs simple, let's put it in RepLogApp.

Initialize numberOfHearts to 1. As *soon* as we do this, thanks to *how* we're rendering RepLogs, this new state is automatically passed as a prop.

```
57 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 6
7   constructor(props) {
... lines 8 - 9
10     this.state = {
... lines 11 - 16
17       numberOfHearts: 1
18     };
... lines 19 - 21
22   }
... lines 23 - 57
```

Awesome! Copy numberOfHearts and head down to add it as a new prop type: numberOfHearts set to PropTypes.number.isRequired.



```

84 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 75
76 RepLogs.propTypes = {
... lines 77 - 81
82   numberOfHearts: PropTypes.number.isRequired
83 };

```

Above, destructure this value out, and then, this looks a little crazy, copy the heart, enter JavaScript, paste the heart in some quotes and `.repeat(numberOfHearts)`.

```

84 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18   const {
... lines 19 - 23
24     numberOfHearts
25   } = props;
... lines 26 - 27
28   if (withHeart) {
29     heart = <span>{'♥ '.repeat(numberOfHearts)}</span>;
30   }
... lines 31 - 84

```

We haven't bound the state to the field yet, but we should *already* be able to play with this! Refresh the page. One heart. Find the React tools and change the state to 10. Yay!

## Binding the Input to State

And *this* is where the two options diverge. If we were to do things the same as before, we would add an `onChange` handler to the input, read the value from the DOM directly, and use *that* to set some state. But, the state and the input wouldn't be connected directly. Oh, and, to be 100% clear, if you want to read a value off of a DOM element directly, you don't *necessarily* need to use refs. Inside the `onChange` handler, you could use `event.target` to get the element.

Refs are just a tool: they're handy if you need to find several fields inside a form, or, in general, whenever you need to work with a DOM element and you don't have access to it.

Anyways, to use the *second*, state-based approach, literally say `value={numberOfHearts}`.

```

84 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 35
36   <input
... line 37
38     value={numberOfHearts}
39   />
... lines 40 - 84

```

Try it! Refresh. And, hey! We see a value of 1! But in the console... a huge error! Wah, wah. Oh, and the field is *stuck* at 1: I can't change it. The error explains why:

You provided a value prop to a form field without an `onChange` handler. This will render as a read-only field.

## Updating the State

This *new* strategy - where you set the value of a field to some state - is called a controlled component. React will *always* make sure that this value is set to the value of this prop... which means that it won't allow us to change it! If we want the value to change, we need to update the underlying state: `numberOfHearts` in `RepLogApp`.

To do this, add another handler function: `handleHeartChange()`. And remember: our top-level component is *all* about changing state: it shouldn't be aware of, or care, that there is a form input that's used to change this. So, give it just one argument: the new `heartCount`.

Inside, set the state! `this.setState()` with `numberOfHearts` set to `heartCount`.

```
65 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 43
44   handleHeartChange(heartCount) {
45     this.setState({
46       numberOfHearts: heartCount
47     });
48   }
... lines 49 - 65
```

And because we just added a handler function, don't forget to go up to the constructor and add `this.handleHeartChange = this.handleHeartChange.bind(this)`.

```
65 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 6
7   constructor(props) {
... lines 8 - 21
22     this.handleHeartChange = this.handleHeartChange.bind(this);
23   }
... lines 24 - 65
```

Back down in `render`, all our state and props are automatically passed. The only things we need to pass manually are the handlers: `onHeartChange={this.handleHeartChange}`.

```
65 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 49
50   render() {
51     return (
52       <RepLogs
... lines 53 - 56
57         onHeartChange={this.handleHeartChange}
58       />
59     )
60   }
... lines 61 - 65
```

Finally, open `RepLogs` and scroll down to `propTypes`: we're now expect an `onHeartChange` function that is required. Back up, destructure that new variable.

```
89 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 16
17  export default function RepLogs(props) {
18    const {
... lines 19 - 24
25      onHeartChange
26    } = props;
... lines 27 - 77
78  }
... line 79
80  RepLogs.propTypes = {
... lines 81 - 86
87    onHeartChange: PropTypes.func.isRequired
88  };
```

We need to update the state *whenever* the field changes. This means we need an `onChange`. Set it to an arrow function with an `e` argument. Inside, it's so nice: `onHeartChange(e.target.value)`.

89 lines | [assets/js/RepLog/RepLogs.js](#)

```
... lines 1 - 32
33   return (
... lines 34 - 36
37     <input
... lines 38 - 39
40       onChange={(e) => {
41         onHeartChange(e.target.value);
42       }}
43     />
... lines 44 - 76
77   );
... lines 78 - 89
```

We *do* reference the DOM element - `e.target` - but just for a moment so that we can call the handler & update the state.

And... we're done! Let's try it - refresh! Change this to 10. Ha! It works! We are happy!

## Casting to an Integer

Oh, except, hmm, we just failed prop validation?

numberOfHearts of type string supplied, expected number.

Interesting. In `RepLogs`, we expect the `numberOfHearts` prop to be a number... which makes sense. But apparently, it's now a string! This isn't *that* important... but it is interesting!

When you read a value from a field, it is, of course, *always* a string! That means the `numberOfHearts` state becomes a string and *that* is passed down as a prop. Let's clean that up: we could do that right here, or inside the handler function. To do it here, oh, this is bizarre, add a `+` before the variable.

89 lines | [assets/js/RepLog/RepLogs.js](#)

```
... lines 1 - 39
40     onChange={(e) => {
41       onHeartChange(+e.target.value);
42     }}
... lines 43 - 89
```

That will *change* the string to a number. There are other ways to do this - JavaScript is weird - but this is one way.

Try it again: change the value and... no error!

Welcome to the world of "controlled components"! It feels really good... but it *can* be a bit more work. Don't worry: in a few minutes, we'll talk about when to use this strategy versus the original.

Oh, but to make this a little bit more fun, change this to `input type="range"`.

89 lines | [assets/js/RepLog/RepLogs.js](#)

```
... lines 1 - 36
37     <input
38       type="range"
... lines 39 - 42
43   />
... lines 44 - 89
```

Try it! Super-fun-heart-slider!!!

Next, let's refactor `RepLogCreator` to use controlled components. This will be the *best* way to see the difference between each approach.

# Chapter 25: Controlled Component Form

With *controlled* components, the value for *each* field in your form needs to be set to state. And then, you need to make sure to *update* that state whenever the field changes. But once you do this, everything else is super nice! The input automatically renders with the correct value, and it's dead-simple to read that state and use it in other places.

In RepLogCreator, we did *not* use this strategy. Nope, we took advantage of refs to access the DOM elements directly, and then read the values from there.

To *really* compare these two approaches, let's see how it would look to use "controlled components" inside of RepLogCreator. Then, later, I'll give you *my* clear recommendation on when to use each.

Copy RepLogCreator and create a *new* file: RepLogCreatorControlledComponents.js. Next, in RepLogs, copy the import statement, comment it out and, instead, import RepLogCreator from this new file.

90 lines | [assets/js/RepLog/RepLogs.js](#)

... lines 1 - 3

```
4 //import RepLogCreator from './RepLogCreator';
5 import RepLogCreator from './RepLogCreatorControlledComponents';
```

... lines 6 - 90

100 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

... lines 1 - 3

```
4 export default class RepLogCreator extends Component {
5   constructor(props) {
6     super(props);
7
8     this.state = {
9       quantityInputError: ""
10    };
11
12    this.quantityInput = React.createRef();
13    this.itemSelect = React.createRef();
14
15    this.itemOptions = [
16      { id: 'cat', text: 'Cat' },
17      { id: 'fat_cat', text: 'Big Fat Cat' },
18      { id: 'laptop', text: 'My Laptop' },
19      { id: 'coffee_cup', text: 'Coffee Cup' },
20    ];
21
22    this.handleFormSubmit = this.handleFormSubmit.bind(this);
23  }
24  ... lines 24 - 94
95 }
```

... lines 96 - 100

## [Adding the new State](#)

Perfect! Because our form has two fields - the select & the input - we need two new pieces of state. On top, add selectedItemId set to empty quotes and quantityValue set to 0. Delete the old refs stuff.

99 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

```
... lines 1 - 4
5     constructor(props) {
... lines 6 - 7
8         this.state = {
9             selectedItem: "",
10            quantityValue: 0,
... line 11
12        };
... lines 13 - 21
22    }
... lines 23 - 99
```

In `render()` destructure these out of state, and use them below: instead of `ref=`, use `value={selectedItemId}`. On the input, the same thing: `value={quantityValue}`.

99 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

```
... lines 1 - 51
52    render() {
53        const { quantityInputError, selectedItem, quantityValue } = this.state;
... line 54
55        return (
... lines 56 - 61
62            <select id="rep_log_item"
63                value={selectedItemId}
... lines 64 - 79
80            <input type="number" id="rep_log_reps"
81                value={quantityValue}
... lines 82 - 91
92        );
93    }
... lines 94 - 99
```

Oh, this is cool: when you use a controlled component with a select element, you add the `value=` to the *select* element itself! That's *not* how HTML works. Normally, you need to add a `selected` attribute to the correct option. But in React, you can pretend like the select itself holds the value. It's pretty nice.

## [Adding the Handler Functions](#)

As *soon* as you bind the value of a field to state, you *must* add an `onChange` handler. Above, create `handleSelectedItemChange()` with an event argument. Inside, all we need to do is set state: `this.setState()` with `selectedItemId` set to `event.target.value`. `event.target` gives us the *select* DOM element, and then we use `.value`. We don't need to read the `selectedIndex` like before.

113 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

```
... lines 1 - 51
52    handleSelectedItemChange(event) {
53        this.setState({
54            selectedItem: event.target.value
55        });
56    }
... lines 57 - 113
```

Copy this function, paste, and call it `handleQuantityInputChange`. This time, update `quantityValue`... but the `event.target.value` part can stay the same. Nice!

113 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

... lines 1 - 57

```
58   handleQuantityInputChange(event) {
59     this.setState({
60       quantityValue: event.target.value
61     });
62   }
```

... lines 63 - 113

Before we use these functions in render, head up to the constructor and bind both of them to this.

Finally, head back down to hook up the handlers: `onChange={this.handleSelectedItemChange}` and for the input, `onChange={this.handleQuantityInputChange}`.

113 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

... lines 1 - 63

```
64   render() {
... lines 65 - 66
67     return (
... lines 68 - 73
74       <select id="rep_log_item"
... line 75
76         onChange={this.handleSelectedItemChange}
... lines 77 - 92
93       <input type="number" id="rep_log_reps"
... line 94
95         onChange={this.handleQuantityInputChange}
... lines 96 - 105
106     );
107   }
```

... lines 108 - 113

Ok: the controlled components are setup! Move over, refresh, inspect element to find the text input, click it, and *then* go over to React. The dev tools show us this *exact* element... which is nice because we can scroll up to find `RepLogCreator` and see its state!

Select a new item. New state! Change the input. New state again!

## Using the new State

The hard work is *now* behind us. Find `handleSubmit()`. Instead of looking at the DOM elements themselves... we can just read the state! On top, destructure what we need: `const { selectedItemId, quantityValue } = this.state`. Delete the old refs stuff.

113 lines | [assets/js/RepLog/RepLogCreatorControlledComponents.js](#)

... lines 1 - 25

```
26   handleSubmit(event) {
... lines 27 - 28
29     const { selectedItemId, quantityValue } = this.state;
... lines 30 - 49
50   }
```

... lines 51 - 113

Then, in the if statement, it's just if `quantityValue`. That *is* nice.

```

113 lines | assets/js/RepLog/RepLogCreatorControlledComponents.js
... lines 1 - 25
26   handleFormSubmit(event) {
... lines 27 - 30
31     if (quantityValue <= 0) {
... lines 32 - 37
38   }
... lines 39 - 49
50 }
... lines 51 - 113

```

Use that again below for onAddRepLog. For the first argument, put a *TODO just* for a minute. Then, at the bottom, *clearing* the form fields is also easier: delete the old code, then re-set the selectedItemId and quantityValue state back to their original values.

```

113 lines | assets/js/RepLog/RepLogCreatorControlledComponents.js
... lines 1 - 39
40   onAddRepLog(
41     'TODO - just wait a second!',
42     quantityValue
43   );
... line 44
45   this.setState({
46     selectedItemId: "",
47     quantityValue: 0,
... line 48
49   });
... lines 50 - 113

```

Ok, back to that onAddRepLog() call. The first argument is the item *label*: that's the visual part of the option, not its value. But our state - selectedItemId *is* the value. We're going to change this to use the value later, once we introduce some AJAX. But, thanks to the itemOptions property we created earlier, we can use the option id to find the text. I'll create a new itemLabel variable and paste in some code. This is *super* not important: it just *finds* the item by id, and, at the end, we call .text to get that property.

```

117 lines | assets/js/RepLog/RepLogCreatorControlledComponents.js
... lines 1 - 30
31   const itemLabel = this.itemOptions.find((option) => {
32     return option.id === this.state.selectedItemId
33   }).text;
... lines 34 - 117

```

Use that below: itemLabel.

```

117 lines | assets/js/RepLog/RepLogCreatorControlledComponents.js
... lines 1 - 43
44   onAddRepLog(
45     itemLabel,
... line 46
47   );
... lines 48 - 117

```

And... I think we're ready! Move over and refresh. Lift our big fat cat 25 times. We got it! Try some coffee while we're at it.

## Controlled Versus Uncontrolled Components

Ok, let's finally judge these two approaches. The old RepLogCreator uses the first strategy, called "Uncontrolled Components". It's about 100 lines long. RepLogCreatorControlledComponents is a bit longer: 116 lines. And that reflects the

fact that controlled components require more setup: each field needs its own state *and* a handler to update state. Sure, there *are* some clever ways to make one handler that can update everything. But, the point is, the added state and handlers means a bit more setup & complexity. On the bright side, when you need to read or update those values, it's super easy! Just use the state. Oh, even this is too much code: I forgot to use the local `selectedItemId` variable.

Controlled components are the React officially-recommended approach to forms. However, because of the added complexity & state, we recommend using "uncontrolled components" instead... most of the time. But, this is subjective, and you'll be fine either way. No decision is permanent, and switching from uncontrolled components to controlled is easy.

So, when *do* we recommend *controlled* components? The biggest time is when you want to do render something as *soon* as a field changes - not *just* on submit. For example, if you wanted to validate a field as the user is typing, disable or enable the submit button as the user is typing or reformat a field - like a phone number field... once again... as the user is typing. This is why the `heartCount` input was *perfect* as a controlled component: we want to re-render the hearts *immediately* as the field changes.

If you're not in one of these situations, you can totally still use controlled components! But we usually prefer uncontrolled components.

Oh, and remember another downside to controlled components is that they *do* require your component to have state. And so, if your dumb component is a function, like `RepLogs`, you'll need to refactor it to a class. No huge deal - just something to think about.

In `RepLogs`, let's change the import to use our original component.



## Chapter 26: Deleting Items

Our app is looking great! But I know, we're missing *one* big piece: actually making AJAX requests so that all of this saves to the server. That is coming *very* soon. But, we have one more piece of homework first: adding the ability to delete rep logs.

Open up RepLogList. This is where we have a little "..." TODO. Turn this into an anchor tag with a span inside: `className="fa fa-trash"`.

```
34 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 17
18     <td>
19         <a href="#">
20             <span className="fa fa-trash"></span>
21         </a>
22     </td>
... lines 23 - 34
```

Cool! That should get us a fancy new trash icon. Awesome.

To hook this up, we're going to go through a process that should be starting to feel familiar... hopefully boring! Here it is: when the user clicks this link in RepLogList, we ultimately need to update the state that lives in RepLogApp. *That* means we need to pass a handler callback from RepLogApp into RepLogs and again into RepLogList.

### Adding & Calling the Delete Handler Function

In RepLogApp, create that new function: `handleDeleteRepLog`, which is a *great* name, because this component doesn't know and doesn't care that a *link* will be used to delete rep logs. Nope, it's all about the data. Give this an `id` argument so we know *which* rep log to delete. Be lazy and log a todo.

```
71 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 50
51     handleDeleteRepLog(id) {
52         console.log('todo');
53     }
... lines 54 - 71
```

Next, because we have a new handler method, make sure to bind it to this.

```
71 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 6
7     constructor(props) {
... lines 8 - 22
23         this.handleDeleteRepLog = this.handleDeleteRepLog.bind(this);
24     }
... lines 25 - 71
```

And finally, pass this as a new prop: `onDeleteRepLog={this.handleDeleteRepLog}`.

```

71 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 54
55   render() {
56     return (
57       <RepLogs
... lines 58 - 62
63         onDeleteRepLog={this.handleDeleteRepLog}
64       />
65     )
66   }
... lines 67 - 71

```

Our work here is done. Now, move to RepLogs. First, at the bottom, add this to propTypes: onDeleteRepLog is PropTypes.func.isRequired.

```

90 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 82
83   highlightedRowId: PropTypes.any,
... lines 84 - 90

```

Above in the function, destructure onDeleteRepLog, find RepLogList, and pass this again as a prop: onDeleteRepLog={onDeleteRepLog}.

```

90 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 18
19   const {
... lines 20 - 26
27     } = props;
28
... lines 29 - 34
35   <div className="col-md-7">
... lines 36 - 55
56     highlightedRowId={highlightedRowId}
... lines 57 - 58
59   />
... line 60
61   <tr>
... lines 62 - 79
80
... lines 81 - 90

```

Finally, move to RepLogList. Start the same: add the new prop to propTypes and destructure the variable.

```

41 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4   export default function RepLogList(props) {
5     const { highlightedRowId, onClick, onDeleteRepLog, repLogs } = props;
... lines 6 - 32
33   }
... line 34
35   RepLogList.propTypes = {
... lines 36 - 37
38     onDeleteRepLog: PropTypes.func.isRequired,
... line 39
40   };

```

Ultimately, we need to execute this callback onClick() of the link. We have a choice here: create an inline arrow function, or

add a function above render. If the logic is simple, both are fine. Add a new `handleDeleteClick` function with two arguments: the event and `repLogId`. Start with `event.preventDefault()` so the browser doesn't try to follow the link. Then, yep, just `onDeleteRepLog(repLogId)`.

```
41 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 6
7   const handleDeleteClick = function(event, repLogId) {
8     event.preventDefault();
9
10    onDeleteRepLog(repLogId);
11  };
... lines 12 - 41
```

Scroll down to hook this up: `onClick={}`. Hmm, we can't call `handleDeleteClick` directly... because we *also* need to pass it the id. No worries: use an arrow function with `(event) => handleDeleteClick()` passing it event and - because we're inside the loop, `repLog.id`.

```
41 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 12
13   return (
... line 14
15     {repLogs.map((repLog) => (
... lines 16 - 24
25       <a href="#" onClick={(event) => handleDeleteClick(event, repLog.id)} >
... lines 26 - 29
30     )})
... line 31
32   );
... lines 33 - 41
```

Let's try it! Refresh! It looks good... and click delete. Nothing happens, but check the console. Got it! There is our todo.

### Updating State: Delete from an Array without Mutating

Now for the fun part! Go back to `RepLogApp`. Inside the handler, we need to remove *one* of the `repLog` objects from the `repLogs` state. But... we do *not* want to *modify* the state. So, the question is: how can we *remove* an item from an array without *changing* that array?

Here's one great way: call `this.setState()` and pass it the key we want to set: `repLogs`. Assign *this* to `this.state.repLogs.filter()`, passing this a callback with a `repLog` argument. For the body, because I didn't add curly braces, we are *returning* `repLog.id !== id`.

```
75 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 50
51   handleDeleteRepLog(id) {
52     // remove the rep log without mutating state
53     // filter returns a new array
54     this.setState({
55       repLogs: this.state.repLogs.filter(repLog => repLog.id !== id)
56     });
57   }
... lines 58 - 75
```

The filter function loops over each `repLog`, calls our function, and if it returns *true*, that `repLog` is added to the new array. This will give us a new, *identical* array... except without the *one* item.

This will work... but! You might also notice another, familiar problem. Because the *new* state depends on the existing state, we should pass `setState()` a callback to avoid a *possible* race condition with state being set at almost the same moment.

Call, `this.setState()` again, but with a callback that receives a `prevState` argument. Copy the object from below, delete all of that code, and *return* this from our callback.

```
77 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 53
54     this.setState((prevState) => {
55         return {
56             repLogs: prevState.repLogs.filter(repLog => repLog.id !== id)
57         };
58     });
... lines 59 - 77
```

That's it! Let's try it! Refresh and... click that trash! It's gone! We got it! And because React is *awesome*, there is *no* doubt that if I add a new item and try to delete it... yep - it works too. Because everything is based on state, there are no surprises.

Ok - it's finally time to start using AJAX to communicate with the server.

# Chapter 27: API Setup & AJAX with fetch()

Ok people: it's time to make our React app *legit*, by loading and saving data to the server via AJAX. Our Symfony app already has a functional set of API endpoints to load, delete and save rep logs. We're not going to spend a lot of time talking about the API side of things: we'll save that for a future tutorial. That's when we'll also talk about other great tools - like ApiPlatform - that you *won't* see used here.

Anyways, I want to at least take you through the basics of my simple, but very functional, setup.

## The API Setup

Open `src/Controller/RepLogController.php`. As you can see, we already have API endpoints for returning all of the rep logs for a user, a single rep log, deleting a rep log and adding a new rep log. Go back to the browser to check this out: `/reps`. Boom! A JSON list of the rep logs.

This endpoint is powered by `getRepLogsAction()`. The `findAllUsersRepLogModels()` method lives in the parent class - `BaseController`, which lives in this same directory. Hold Command or Ctrl and click to jump into it.

The *really* important part is this: I have *two* rep log classes. First, the `RepLog` entity stores all the data in the database. Second, in the `Api` directory, I have *another* class called `RepLogApiModel`. *This* is the class that's transformed into JSON and used for the API: you can see that it has the same fields as the JSON response.

The `findAllUsersRepLogModels` method first queries for the `RepLog` entity objects. Then, it loops over each and *transforms* it into a `RepLogApiModel` object by calling another method, which lives right above this. The code is *super* boring and not fancy at all: it simply takes the `RepLog` entity object and, piece by piece, converts it into a `RepLogApiModel`.

Finally, back in `getRepLogsAction()`, we return `$this->createApiResponse()` and pass it that array of `RepLogApiModel` objects. This method also lives inside `BaseController` and it's dead-simple: it uses Symfony's serializer to turn the objects to JSON, then puts that into a `Response`.

That's it! The most interesting part is that I'm using *two* classes: the entity and a separate class for the serializer. Having 2 classes means that you need to do some extra work. However... it makes it *really* easy to make your API look *exactly* how you want! But, in a lot of cases, serializing your entity object directly works great.

## Using fetch()

So here's our first goal: make an API request to `/reps` and use that to populate our initial `repLogs` state so that they render in the table.

In the `assets/js` directory, create a new folder called `api` and then a new file called `rep_log_api.js`. This new file will contain all of the logic we need for making requests related to the rep log API endpoints. As our app grows, we might create *other* files to talk to *other* resources, like "users" or "products".

You probably also noticed that the filename is lowercase. That's a minor detail. This is because, instead of exporting a class, this module will export some functions... so that's just a naming convention. Inside, export function `getRepLogs()`.

```
12 lines | assets/js/api/rep_log_api.js
```

```
... lines 1 - 5
```

```
6 export function getRepLogs() {
```

```
... lines 7 - 12
```

The question now is... how do we make AJAX calls? There are several great libraries that can help with this. But... actually... we don't need them! All modern browsers have a built-in function that makes AJAX calls *super* easy. It's called `fetch()`!

Try this: `return fetch('/reps')`. `fetch()` returns a `Promise` object, which is a *super* important, but kinda-confusing object we talked a lot about in our ES6 tutorial. To *decode* the JSON from our API into a JavaScript object, we can add a success handler: `.then()`, passing it an arrow function with a `response` argument. Inside, return `response.json()`.

12 lines | [assets/js/api/rep\\_log\\_api.js](#)

... lines 1 - 5

```
6 export function getRepLogs() {
7   return fetch('/reps')
8     .then(response => {
9       return response.json();
10    });
11 }
```

With this code, our `getRepLogs()` function will *still* return a Promise. But the "data" for that should now be the decoded JSON. Don't worry, we'll show this in action.

By the way, I mentioned that `fetch` is available in all modern browsers. So yes, we *do* need to worry about what happens in older browsers. We'll do that later.

Go back to `RepLogApp`. Ok, as *soon* as the page loads, we want to make an AJAX call to `/reps` and use that to populate the state. The constructor seems like a good place for that code. Oh, but first, bring it in: `import { getRepLogs } from ../api/rep_log_api`. For the first time, we're not exporting a default value: we're exporting a *named* function. We'll export more named functions later, for inserting and deleting rep logs.

83 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 4

```
5 import { getRepLogs } from '../api/rep_log_api';
```

... lines 6 - 83

Oh, and, did you see how PhpStorm auto-completed that for me? That was awesome! And it wasn't video magic: PhpStorm was cool enough to guess that correct import path.

Down below, add `getRepLogs()` and chain `.then()`. Because we decoded the JSON already, this should receive that decoded data. Just log it for now.

83 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 7

```
8 constructor(props) {
  ... lines 9 - 10
11   getRepLogs()
12     .then((data) => {
13       console.log(data);
14     });
  ... lines 15 - 29
30 }
  ... lines 31 - 83
```

Ok... let's try it! Move over and, refresh! Oof! An error:

Unexpected token in JSON at position zero

Hmm. It *seems* like our AJAX call might be working... but it's having a problem *decoding* the JSON. It turns out, the problem is authentication. Let's learn how to debug this and how to authenticate our React API requests next.

# Chapter 28: API Auth & State via AJAX

When we used `fetch()` to make an AJAX call to `/reps...` we were "rewarded" with this big ugly error. This tells me that `fetch` is *probably* having problems... for *some* reason... parsing the JSON in the response.

Let's go see what happened! Click on the Network tab in your browser tools and filter for only XHR requests. Ah, here is one for `/reps` that was successful. BUT! That's the *wrong* AJAX call: this is the AJAX call made by our *old* code. So... where the heck is the *other* `/reps` AJAX call that was just made by `fetch()`?

Click instead to filter by the "Other" tab. There it is! Why is it here? Well... because... something went wrong. Look at the response: 302. And if you look at the response headers... woh! It is a redirect to the login page, which is why you see a *second* request below for `/login`.

Let's back up. First, for some reason, authentication is failing for the API request. We'll get to that in a minute. Second, `fetch()` requests will *normally* show up under the XHR network filter. We'll see that later. But, if something goes wrong, the request *may* show up under "Other". Just be aware of that: it's a gotcha!

## [Sending the Cookie Headers](#)

So, why the heck is authentication failing? If we go directly to `/reps`, it works! What's wrong with you `fetch`!

This, in my opinion, is one of the really cool things about `fetch`. Look at our controller. Ah, every endpoint requires us to be logged in! This works in our browser because our browser automatically sends the session cookie. But `fetch()`, on the other hand, does *not* automatically send any cookies when it makes a request.

## [What Type of Authentication to Use?](#)

I like this because it *forces* you to ask yourself:

Hey, how *do* I want to authenticate my API requests?

API authentication is a big topic. So we're going to skip it! I'm kidding: it's too important.

One way or another, *every* API request that needs authentication will have *some* sort of authentication data attached to it - maybe a session cookie or an API token set on a header.

So... what type of authentication *should* you use for your API? Honestly, if you're building an API that will be consumed by your own JavaScript front-end, using session cookies is an *awesome* option! You don't need anything fancier. When we login, that sets a session cookie. In a moment, we'll tell `fetch` to *send* that cookie, and everything will be solved. If you want to build your login page in React and send the username and password via AJAX, that's totally fine: when your server sends back the session cookie, your browser will see it & store it. Well, as long as you use the `credentials` option that I'm about to show you for that AJAX call.

Of course, if you want, you can also create an API token authentication system, like JWT or OAuth. That's totally fine, but that truly *is* a separate topic.

Whatever you choose, when it's time to make your API call, you will *attach* the authentication info to the request: either by sending the session cookie or your API token as a header.

## [Sending the Session Cookie](#)

To send the session cookie, `fetch` has a second options argument. Add `credentials` set to `same-origin`. Thanks to this, `fetch()` will send cookies to any requests made back to *our* domain.

14 lines | assets/js/api/rep\_log\_api.js

... lines 1 - 6

```
7   return fetch('/reps', {
8     credentials: 'same-origin'
9   })
```

... lines 10 - 14

## Tip

The default value of credentials may change to same-origin in the future.

Ok, let's see if this fixes things! Move over and refresh. No errors! Check out the console. Yes! *There* is our data! Notice, the API wraps everything inside an items key. Yep, inside: the 4 rep logs, which have the same fields as the state in our app. That was no accident: when we added the static data, I made sure it looked like the *real* data from the API so that we could swap it out later.

## Processing the fetch data via Promises

In rep\_log\_api, I really want my getRepLogs() API to return a Promise that contains the *array* of rep logs... without that items key. To do that, it's a bit weird. The .json() method returns another Promise. So, to do further processing, chain a .then() from it and, inside the callback, return data.items.

14 lines | assets/js/api/rep\_log\_api.js

... lines 1 - 9

```
10   .then(response => {
11     return response.json().then(data => data.items)
12   });
```

... lines 13 - 14

Promises on top of promises! Yaaaay! When fetch() finishes, it executes our first callback. Then, when the JSON decode finishes, it executes our *second* callback, where we read off the .items key. Ultimately, getRepLogs() returns a Promise object where the data is the array of rep logs. Phew!

And because the browser already refreshed while I was explaining all of the promises, yep! You can see the logged data is now the array.

## componentDidMount

Awesome! Let's use this to set our initial state! First, set the initial repLogs state to an empty array. Next, copy the getRepLogs() call and remove it. Instead, create a new method called componentDidMount() and paste this there. In the callback, use this.setState() to set repLogs to data.

83 lines | assets/js/RepLog/RepLogApp.js

... lines 1 - 22

```
23   componentDidMount() {
24     getRepLogs()
25       .then(data => {
26         this.setState({
27           repLogs: data
28         })
29       });
30   }
```

... lines 31 - 83

Before we talk about this, let's try it. Refresh! Woh! We have *real* data! Yes, yes, yes! We're showing the *same* data as our original app!

Back to the code! Until this moment, render() was the only "special", React-specific, method in our class. But there are a *few* other special methods called "lifecycle" methods. The componentDidMount() method is one of those: if this exists, React



calls it right *after* our component is rendered to the DOM. And *this* is the best place to make any AJAX requests needed to populate your initial state.

Actually, we *could* have left this code in the constructor(). Because we're in a browser, they're *almost* the same. But, `componentDidMount()` is generally the recommended place.

## Chapter 29: Loading Messages

Hmm. Refresh the page and watch closely. See it? When it first loads, the table is empty *just* for a moment. Then, when our AJAX call finishes, it gets filled in. That makes perfect sense, but it does make the page feel momentarily "broken"... and it may not always load this quickly.

No worries! This is why the Internet invented loading messages and animations! How can we add these in React? I have no idea! Let's find out!

Here's our goal: before our AJAX call finishes, I want to render one row that just says "Loading". Hmm... this will need to happen inside RepLogList: that's where the tbody lives. But, hmm again: this component somehow needs to know whether or not the AJAX call has finished... and *that* is something that only RepLogApp knows.

To keep track of whether or not the AJAX call is finished, we need new state. On top, add some new state isLoading: false.

```
85 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 7
8   constructor(props) {
... lines 9 - 10
11   this.state = {
... lines 12 - 14
15     isLoading: false
16   };
... lines 17 - 21
22 }
... lines 23 - 85
```

Then, down below, when fetch() finishes, set isLoading to true!

```
85 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 23
24 componentDidMount() {
25   getRepLogs()
26     .then((data) => {
27     this.setState({
... line 28
29       isLoading: true
30     })
31   });
32 }
... lines 33 - 85
```

State, done! And thanks to how we're rendering RepLogs, this state is automatically pass as a prop. And *now* we start the prop-passing dance! In RepLogs, add the new prop type at the bottom: PropTypes.bool.isRequired. Oh, and you've probably noticed that I like to make pretty much *everything* required. That's a personal preference. Because this is *my* app, if I forget to pass a prop, it's probably a typo and I want to know.

```
96 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 84
85 RepLogs.propTypes = {
... lines 86 - 93
94   isLoading: PropTypes.bool.isRequired,
95 };
```

Next, scroll up, destructure the `isLoading` variable, find `RepLogList`, and pass that prop: `isLoading={isLoading}`.

```
96 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18 export default function RepLogs(props) {
19   const {
... lines 20 - 27
28     isLoading
29   } = props;
... lines 30 - 35
36   return (
37     <div className="col-md-7">
... lines 38 - 56
57     <RepLogList
... lines 58 - 61
62       isLoading={isLoading}
63     />
... lines 64 - 80
81   </div>
82 );
83 }
... lines 84 - 96
```

Finally, do the same in that component: I'll steal the prop type and go up to destructure the variable.

```
52 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4 export default function RepLogList(props) {
5   const { highlightedRowId, onRowClick, onDeleteRepLog, repLogs, isLoading } = props;
... lines 6 - 42
43 }
... line 44
45 RepLogList.propTypes = {
... lines 46 - 49
50   isLoading: PropTypes.bool.isRequired,
51 };

```

Ok, this is interesting: if the app is *not* loaded yet, we don't need to run *any* of this code down here. So, we can short-circuit the entire process: if `isLoading`, then return a completely new set of JSX, with a `tbody`, `tr` and `<td colSpan="4" className="text-center">`. Say, "Loading...".

```
52 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 6
7   if (isLoading) {
8     return (
9       <tbody>
10        <tr>
11          <td colSpan="4" className="text-center">Loading...</td>
12        </tr>
13      </tbody>
14    );
15  }
... lines 16 - 52
```

Oh, and notice that this is `colSpan` with a *capital* "S". This is another, uncommon, case where the prop is slightly different than the HTML attribute. PhpStorm made it easy by auto-completing the correct version for React.

And... yea! That's it! Let's go refresh... but watch closely. There it was! And because React's model is so flexible, if you ever needed to, for some reason, *reload* the data, you could re-render that loading message simply by updating one piece of state. Nice.

We're on a roll! So let's make the delete link talk to our API.

# Chapter 30: Hitting the DELETE Endpoint

We did *all* the hard work in the beginning: setting up our components and passing around state & callbacks. So now that it's time to make our React app talk to an API, dang, life is fun!

Let's hook up the delete link to our API next. On RepLogController, we already have an endpoint for this: a DELETE method to `/reps/{id}`.

Symfony queries for the RepLog entity object and we delete it. Oh, and then we return an *empty* Response.

In JavaScript, find `rep_log_api.js`: *this* is our home for *all* API requests related to rep logs. Create a second function: `export function deleteRepLog()` with an `id` argument. Let's cheat and copy the code from `getRepLogs()`. But, for the URL, use `ticks` and say `/reps/${id}`.

```
21 lines | assets/js/api/rep_log_api.js
... lines 1 - 14
15 export function deleteRepLog(id) {
16   return fetch(`/reps/${id}`, {
17     credentials: 'same-origin',
... line 18
19   });
20 }
```

## Hardcoding URLs?

If you're a hardcore Symfony user... you might hate this! We're *hardcoding* our URLs! Ah! In Symfony, we *never* do this. Nope, we always *generate* a URL by using its route - like with the `path()` function in Twig.

When you're working in React - or inside any JavaScript - you have two options when it comes to URLs. Either, (A) hardcode the URLs like I'm doing or (B) somehow generate them dynamically. To generate them, you could use `FOSJsRoutingBundle`, which is a great option, or set them to a JavaScript variable in Twig and pass them as props. You'll learn how to pass data from Twig to JavaScript later.

But honestly, hardcoding URLs in JavaScript is fine. Your API and your JavaScript are partners: they work together. And that means, if you change something in your API, like a URL - or even a field name - you need to realize that something will probably *also* need to change in JavaScript. As long as you keep this in mind, it's no big deal. It's even *less* of a big deal because we're organizing all of our API calls into one spot.

## Calling the Endpoint

Anyways, the *other* change is that we need to make a DELETE request. Do that with another option: `method: 'DELETE'`.

```
21 lines | assets/js/api/rep_log_api.js
... lines 1 - 15
16   return fetch(`/reps/${id}`, {
... line 17
18     method: 'DELETE'
19   });
... lines 20 - 21
```

Alright! Back to RepLogApp to put this in action! When a rep log is deleted, `handleDeleteRepLog` is called and that removes it from state. Now, we also need to call our endpoint. Head to the top and *also* import `deleteRepLog`. Down below, do it: `deleteRepLog(id)`.

```

87 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 4
5 import { getRepLogs, deleteRepLog } from '../api/rep_log_api';
... line 6
7 export default class RepLogApp extends Component {
... lines 8 - 58
59   handleDeleteRepLog(id) {
60     deleteRepLog(id);
... lines 61 - 68
69   }
... lines 70 - 82
83 }
... lines 84 - 87

```

That, is, nice! Try it: move over, refresh and... click delete! Check it out!

Fetch loading finished: DELETE /reps/27

I think it worked! Because this "fetch" call was *successful*, you can see it under the XHR filter. To make sure it *really* deleted: refresh. Yep! Just these 3 brave rep logs remain.

## Optimistic UI Updating

I want to point something out: notice that we *start* the AJAX request, but then *immediately* update the state... even before it finishes. This is called an "optimistic UI update": it's where you update your state & UI *before* your server *actually* saves or deletes the data.

I think this is great, but in some situations, you might want to *wait* to update the state, until the AJAX call finishes. For example, if the AJAX call might fail due to some failed validation. We'll talk more about that later.

## Centralizing the fetch Call()

But first, it's time to centralize some code! In `rep_log_api.js`, we're starting to repeat ourselves! We now have credentials: 'same-origin' in two places. That may not *seem* like a big deal. But, if you were sending an API token and *always* needed to set a header, centralizing this code would be super important.

Let's create a new utility function that *everything* else will use. At the top of the file, create a function called `fetchJson()` with the two arguments `fetch` needs: the URL and options. Inside, return `fetch()`, the URL, and, for the options, use `Object.assign()` passing it an object with credentials set to `same-origin`, comma, options.

```

25 lines | assets/js/api/rep_log_api.js
1 function fetchJson(url, options) {
2   return fetch(url, Object.assign({
3     credentials: 'same-origin',
4   }, options))
... lines 5 - 7
8 }
... lines 9 - 25

```

`Object.assign()` is JavaScript's equivalent of `array_merge()` when dealing with objects: it takes any options *we* might pass in and merges them into this object. So, credentials will always be in the final options.

Then, because every endpoint will return JSON, we can `.then()` to transform the Promise data from the response object into JSON.

### Tip

This introduces a bug when the response is null. We'll handle it in chapter 35  
<https://symfonycasts.com/screencast/reactjs/deep-state-update>

25 lines | assets/js/api/rep\_log\_api.js

... lines 1 - 4

```
5      .then(response => {  
6        return response.json();  
7      });
```

... lines 8 - 25

And just like that, we have a nice utility function that will set our credentials *and* JSON-decode the response. We're awesome! In `getRepLogs()`, simplify: `fetchJson('/reps')`. To *only* return the items key, add `.then(data => data.items)`. This function now returns the same thing as before.

25 lines | assets/js/api/rep\_log\_api.js

... lines 1 - 14

```
15 export function getRepLogs() {  
16   return fetchJson('/reps')  
17     .then(data => data.items);  
18 }
```

... lines 19 - 25

For `deleteRepLog()`, use `fetchJson()` and remove the credentials key.

25 lines | assets/js/api/rep\_log\_api.js

... lines 1 - 19

```
20 export function deleteRepLog(id) {  
21   return fetchJson(`/reps/${id}`, {  
22     method: 'DELETE'  
23   });  
24 }
```

Ok, try it out! Refresh! Yep! Everything works fine. Time to connect our *form* with the rep log *create* API endpoint.

# Chapter 31: The POST Create API

We now GET *and* DELETE the rep logs via the API. The last task is to *create* them when the form submits. Look back at RepLogController: we can POST to /reps to create a new rep log. I want to show you *just* a little bit about how this works.

## About the POST API Code

The endpoint expects the data to be sent as JSON. See: the *first* thing we do is `json_decode` the request content. Then, we use Symfony's form system: we have a form called `RepLogType` with two fields: `reps` and `item`. This is bound directly to the `RepLog` entity class, not the model class.

Using the form system is optional. You could also just use the raw data to manually populate a new `RepLog` entity object. You could *also* use the serializer to *deserialize* the data to a `RepLog` object.

These are all great options, and whatever you choose, you'll ultimately have a `RepLog` entity object populated with data. I attach this to our user, then flush it to the database.

For the response, we *always* serialize `RepLogApiModel` objects. So, after saving, we convert the `RepLog` *into* a `RepLogApiModel`, turn that into JSON and return it.

I also have some data validation above, which we'll handle in React later.

## Fetching to POST /reps

To make the API request in React, start, as we *always* do, in `rep_log_api.js`. Create a *third* function: `export function createRepLog`. This needs a `repLog` argument, which will be an object that has all the fields that should be sent to the API.

```
34 lines | assets/js/api/rep_log_api.js
... lines 1 - 25
26 export function createRepLog(repLog) {
... lines 27 - 33
34 }
```

Use the new `fetchJson()` function to `/reps` with a method set to `POST`. This time, we *also* need to set the body of the request: use `JSON.stringify(repLog)`. Set one more option: a `headers` key with `Content-Type` set to `application/json`. This is optional: my API doesn't actually read or care about this. But, because we *are* sending JSON, it's a best-practice to say this. And, later, our API *will* start requiring this.

```
34 lines | assets/js/api/rep_log_api.js
... lines 1 - 26
27 return fetchJson('/reps', {
28   method: 'POST',
29   body: JSON.stringify(repLog),
30   headers: {
31     'Content-Type': 'application/json'
32   }
33 });
```

Ok, API function done! Head back to `RepLogApp` and scroll up: `import createRepLog`. Then, down in `handleAddRepLog`, use it! `createRepLog(newRep)`. To see what we get back, add `.then()` with `data`. `console.log()` that.



```

93 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 4
5   import { getRepLogs, deleteRepLog, createRepLog } from '../api/rep_log_api';
... line 6
7   export default class RepLogApp extends Component {
... lines 8 - 37
38   handleAddRepLog(itemLabel, reps) {
... lines 39 - 45
46     createRepLog(newRep)
47       .then(data => {
48         console.log(data);
49       })
50     ;
... lines 51 - 56
57   }
... lines 58 - 88
89 }
... lines 90 - 93

```

Well... let's see what happens! Move over and refresh. Okay, select "Big Fat Cat", 10 times and... submit! Boo! The POST failed! A 400 error!

### Matching the Client Data to the API

Go check it out. Interesting... we get an error that this form should not contain extra fields. Something is not right. In Symfony, you can look at the profiler for *any* AJAX request. Click into this one and go to the "Forms" tab. Ah, the error is attached to the *top* of the form, not a specific field. Click ConstraintViolation to get more details. Oh... this value key holds the secret. Our React app is sending id, itemLabel and totalWeightLifted to the API. But, look at the form! The only fields are reps and item! We shouldn't be sending *any* of these other fields!

Actually, itemLabel is *almost* correct. It *should* be called item. And instead of being the *text*, the server wants the value from the selected option - something like fat\_cat.

Ok, so we have some work to do. Head back to RepLogApp. First: remove the stuff we *don't* need: we don't need id and we're not responsible for sending the totalWeightLifted. Then, rename itemLabel to item. Rename the argument too, because this *now* needs to be the option value.

```

91 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 37
38   handleAddRepLog(item, reps) {
39     const newRep = {
40       reps: reps,
41       item: item
42     };
... lines 43 - 54
55   }
... lines 56 - 91

```

This function is eventually called in RepLogCreator as onAddRepLog. Instead of text, pass value.

```

100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 40
41   onAddRepLog(
42     itemSelect.options[itemSelect.selectedIndex].value,
... line 43
44   );
... lines 45 - 100

```

## Updating State *after* the AJAX Call

In RepLogApp, newRep *now* contains the data our API needs! Woohoo! But... interesting. It turns out that, at the moment the user submits the form, we don't have all the data we need to update the state. In fact, we never did! We were just faking it by using a random value for totalWeightLifted.

*This* is a case where we *can't* perform an optimistic UI update: we *can't* update the state *until* we get more info back from the server. This is no big deal, it just requires a bit more work.

Comment out the setState() call.

```
91 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 37
38   handleAddRepLog(item, reps) {
... lines 39 - 49
50     // this.setState(prevState => {
51     //   const newRepLogs = [...prevState.repLogs, newRep];
52     //
53     //   return {repLogs: newRepLogs};
54     // })
55   }
... lines 56 - 91
```

Let's refresh and *at least* see if the API call works. Lift my big fat cat 55 times and hit enter. Yes! No errors! The console log is coming from the POST response... it looks perfect! Id 30, *it* returns the itemLabel and also calculates the totalWeightLifted. Refresh, yep! There is the new rep log!

Ok, let's update the state. Because our API rocks, we know that the data is actually a repLog! Use this.setState() but pass it a callback with prevState. Once again, the *new* state depends on the *existing* state.

```
89 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 37
38   handleAddRepLog(item, reps) {
... lines 39 - 43
44     createRepLog(newRep)
45     .then(repLog => {
46       this.setState(prevState => {
... lines 47 - 49
50       })
51     })
52   ;
53   }
... lines 54 - 89
```

To add the new rep log without mutating the state, use const newRepLogs = an array with ...prevState.repLogs, repLog. Return the new state: repLogs: newRepLogs. Remove all the old code below.

```
89 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 44
45     .then(repLog => {
46       this.setState(prevState => {
47         const newRepLogs = [...prevState.repLogs, repLog];
48
49         return {repLogs: newRepLogs};
50       })
51     })
... lines 52 - 89
```

Let's try it! Make sure the page is refreshed. Lift our normal cat this time, 10 times, and boom! We've got it!

### Using UUID's?

This was the *first* time that our React app did *not* have all the data it needed to update state immediately. It needed to wait until the AJAX request finished.

Hmm... if you think about it, this will happen *every* time your React app *creates* something through your API... because, there is *always* one piece of data your JavaScript app doesn't have before saving: the new item's database id! Yep, we will *always* need to create a new item in the API first so that the API can send us back the new id, so that *we* can update the state.

Again, that's no *huge* deal... but it's a bit more work, and it will require you to add more "loading" screens so that it looks like your app is saving. It's just *simpler* if you can update the state immediately.

And *that* is why UUID's can be awesome. If you configure your Doctrine entities to use UUID's instead of auto-increment ids, you *can* generate valid UUID's in JavaScript, update the state immediately, and send the new UUID on the POST request. The server would then make sure the UUID has a valid format and use it.

If you're creating a lot of resources, keep this in mind!

# Chapter 32: Polyfills: fetch & Promise


The `fetch()` function is built into all modern browsers... which is cool because we didn't need to install *any* outside libraries to make AJAX requests. Yay! Except... what about older browsers? I'm looking at you IE! Google for "caniuse fetch".

The *good* news is that `fetch()` is *almost* universal... ahem, IE 11. So... you might be ok to do nothing! But, for the rest of us that *do* have a few users on IE, yea, using `fetch()` will be a problem. But, a problem we can fix!

When we use new JavaScript syntaxes - like the arrow function - behind the scenes, Babel *transpiles* - basically *rewrites* - that code into old, boring syntax that all browsers support. So, that's already handled. But, when there is a totally new *feature*, like `fetch()`, Babel doesn't handle that. Instead, you need a polyfill: a fancy word for a library that *adds* a feature if it's missing.

## [The fetch Polyfill](#)

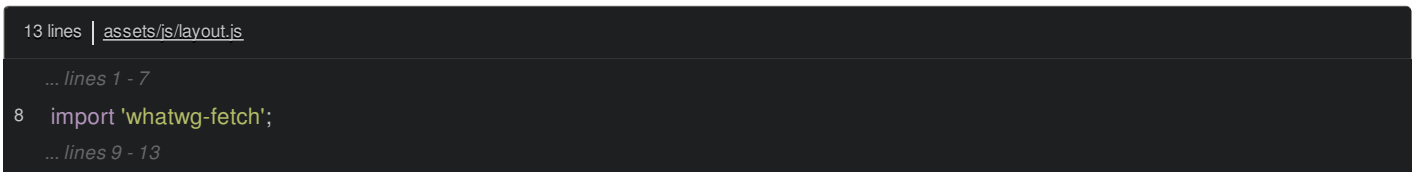
Google for "fetch polyfill" to find a library from GitHub. Scroll down to find the library name: `whatwg-fetch`. Copy that, find your open terminal and:



```
$ yarn add whatwg-fetch --dev
```

To use a JavaScript polyfill, all *we* need to do is import the file. Internally, *it* will figure out whether or not the current browser has the feature, like the global `fetch()` function. If it does *not*, it will add it.

Super easy! To make sure `fetch()` is available everywhere, we can import it from our entry file: `rep_log_react.js`. Then, it will *definitely* be available in `rep_log_api.js`. But... I like to get *even* crazier! Open `layout.js`. Then look inside `webpack.config.js`. `layout` is configured as my "shared" entry... which is a fancy way of saying that `layout.js` is included on *every* page and so its code will *always* run. Inside that file, import 'whatwg-fetch'.



```
13 lines | assets/js/layout.js
```

```
... lines 1 - 7
```

```
8 import 'whatwg-fetch';
```

```
... lines 9 - 13
```

*Every* part of my app can now safely rely on the fact that the global `fetch()` function will be available.

Oh, and because I *love* digging in to see how things work, let's go check out the polyfill code! Open `node_modules`, search for `whatwg`, expand its directory and open `fetch.js`. *This* is the file we just imported.

Look *all* the way at the bottom: it's a self-executing function. It passes this into the function, which in a browser environment, is the global `window` object. Then, on top, that `window` object is passed as a `self`. And because all global variables & functions are actually *properties* on the `window` object, it's able to ask:


Hey! Does the `window` variable have the global `fetch()` function on it?

If it does, it just returns. If it does not, the rest of this code works to define and add it. There it is: `self.fetch =`. This is a polyfill in action - kinda cool.

## [The Promise Polyfill](#)

Go back to the `fetch` polyfill docs. Ah, it says that we *also* need to use a `Promise` polyfill for older browsers. We can *literally* see this inside of the `fetch` polyfill: it assumes that a `Promise` class is available.

Let's polyfill it to be safe: click into the library they recommend. Cool: copy the name, move over, and:



```
$ yarn add promise-polyfill --dev
```

When it finishes, head back to the docs. Interesting: this shows two different import options. You can use the *second* one to *import* a Promise object, but *without* adding a new global variable. Because we *do* want to guarantee that a global Promise variable exists, copy the first one. In layout.js, paste!


14 lines | [assets/js/layout.js](#)

... lines 1 - 8

9 import 'promise-polyfill/src/polyfill';

... lines 10 - 14

To make sure we didn't break anything, go back to the tab that's running encore and restart it:

A terminal window with a dark background and light gray window controls (three circles) at the top. The terminal shows a command prompt followed by the command to restart the dev-server.

```
$ yarn run encore dev-server
```

Perfect! We now support older browsers... ahem, IE.

## Chapter 33: Success Messages + The Style Attribute

When you have a super-fancy, AJAX-powered app like we do, success messages and loading animations are *essential* to having a beautiful user experience. Of course, you will choose *how* fancy you want to get: more fancy just means more complexity.

Let's look at one rough spot. Watch carefully: there's a delay between when we submit the form and when the new row appears. Sure, that was pretty quick - but if it is ever *any* slower, it's going to look broken.

The delay is because we are *not* doing an optimistic UI update: we don't set the state until *after* the AJAX call finishes. Let's smooth this out: let's add a "loading" row at the bottom of the table while we're saving.

### Adding the "saving" State

To do this, our app needs to know whether or not a new rep log is currently being saved. So, we need new state! And this state will *definitely* live in RepLogApp, because it's the only component that is even *aware* that AJAX is happening. Call it `isSavingNewRepLog` and initialize it to `false`.

```
97 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 7
8   constructor(props) {
  ... lines 9 - 10
11   this.state = {
  ... lines 12 - 15
16     isSavingNewRepLog: false
17   };
  ... lines 18 - 22
23 }
  ... lines 24 - 97
```

Down below, before we call `createRepLog()`, add `this.setState()` to change `isSavingNewRepLog` to `true`. And *after* the AJAX call finishes, let's break this onto multiple lines and then set this same key *back* to `false`.

```
97 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 38
39 handleAddRepLog(item, reps) {
  ... lines 40 - 44
45   this.setState({
46     isSavingNewRepLog: true
47   });
  ... line 48
49   createRepLog(newRep)
50     .then(repLog => {
51       this.setState(prevState => {
  ... lines 52 - 53
54         return {
55           repLogs: newRepLogs,
56           isSavingNewRepLog: false
57         };
58       })
59     })
60   ;
61 }
  ... lines 62 - 97
```

That felt *great*! Adding and managing new state in our smart component continues to be *very* simple.

## Passing & Using the Prop

Next: where do we need to use this value? The tbody lives in RepLogList: this is where we'll add a temporary row. So, we need to, once again, do the fancy prop-passing dance. First, all state is already passed from RepLogApp to RepLogs. Inside that component, define a new prop type: isSavingNewRepLog as a required bool. Above, destructure it and then, find RepLogList and pass it down.

```
99 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18 export default function RepLogs(props) {
19   const {
... lines 20 - 28
29     isSavingNewRepLog
30   } = props;
... lines 31 - 36
37   return (
... lines 38 - 48
49     <table className="table table-striped">
... lines 50 - 57
58       <RepLogList
... lines 59 - 63
64         isSavingNewRepLog={isSavingNewRepLog}
65       />
... lines 66 - 73
74     </table>
... lines 75 - 83
84   );
85 }
... line 86
87 RepLogs.propTypes = {
... lines 88 - 96
97   isSavingNewRepLog: PropTypes.bool.isRequired,
98 };
```

Copy the new prop type and *also* put it into RepLogList. In render(), destructure the new variable.

```
64 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 3
4 export default function RepLogList(props) {
5   const { highlightedRowId, onRowClick, onDeleteRepLog, repLogs, isLoading, isSavingNewRepLog } = props;
... lines 6 - 53
54 }
... line 55
56 RepLogList.propTypes = {
... lines 57 - 61
62   isSavingNewRepLog: PropTypes.bool.isRequired
63 };
```

Ok! *Now* we're ready. Move down to *after* the map function so that our new tr appears at the bottom of the table. To print the new row *only* when we need it, use the trick we learned earlier: isSavingNewRepLog &&, then open a set of parentheses. Now, just add the tr and td: "Lifting to the database...". Give that a colSpan=4 and className="text-center".

```

64 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 22
23   return (
... lines 24 - 40
41     {isSavingNewRepLog && (
42       <tr>
43         <td
44           colSpan="4"
45           className="text-center"
... lines 46 - 48
49       >Lifting to the database ...</td>
50     </tr>
51   )}
... line 52
53 );
... lines 54 - 64

```

## The style Prop

And, hmm... it might look better if we lower the opacity a bit. Do that with a style prop. But, the style prop works a bit different than the style HTML attribute: instead of being a string of styles, React expects an *object* of the styles we want. This is actually easier, but the syntax looks a bit nuts. First, we use `{}` to move into JavaScript mode. Then, we add *another* set of `{}` to define an object, with opacity: .5.

```

64 lines | assets/js/RepLog/RepLogList.js
... lines 1 - 45
46     style={{
47       opacity: .5
48     }}
... lines 49 - 64

```

The double `{{` almost looks like Twig code. But really, we're doing two separate things: entering JavaScript and then creating an object.

Try it! Move over, refresh, fill out the form and... watch closely. There it was! It was *beautiful*!

## Success Message

While we're adding some little "touches" to make the UI better, let's add a new success message when the new rep log API call finishes.

Once again, in `RepLogApp`, we need new state for this message. Give it a generic name - `successMessage`. We may be able to use this in a few other places, like when deleting a rep log.

```

99 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 7
8   constructor(props) {
... lines 9 - 10
11     this.state = {
... lines 12 - 16
17       successMessage: ""
18     };
... lines 19 - 23
24   }
... lines 25 - 99

```

Below, after `createRepLog()` finishes, update this state: `successMessage` set to "Rep Log Saved!".



```

99 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 49
50     createRepLog(newRep)
51     .then(repLog => {
52         this.setState(prevState => {
... lines 53 - 54
55         return {
... lines 56 - 57
58             successMessage: 'Rep Log Saved!'
59         };
60     })
61 })
... lines 62 - 99

```

Cool! This time, I want to print the message right on top of the app, above the table. That markup lives in RepLogs. Go straight into that component and define the new prop type: successMessage as a string that's required.

```

109 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 95
96     RepLogs.propTypes = {
... lines 97 - 106
107     successMessage: PropTypes.string.isRequired
108 };

```

Destructure that variable... then, after the input, use our trick: successMessage && open parentheses. Render a div with a few bootstrap classes: alert alert-success text-center. Inside, print the text!

```

109 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18 export default function RepLogs(props) {
19     const {
... lines 20 - 29
30     successMessage
31     } = props;
... lines 32 - 37
38     return (
... lines 39 - 51
52     {successMessage && (
53         <div className="alert alert-success text-center">
54             {successMessage}
55         </div>
56     )}
... lines 57 - 92
93 );
94 }
... lines 95 - 109

```

I love it! Head back to your browser and refresh! Let's delete a few rep logs to clean things up. Then, lift your coffee cup 12 times and, submit! Boom! There is our new message.

The only problem is that the message stays up there... forever! That should probably disappear after a few seconds. Let's do that next!

# Chapter 34: Temporary Messages & componentWillUnmount

That success message is cool... but it should probably disappear after a few seconds. No problem! We can use the native `setTimeout()` function to change the state *back* to empty after a few seconds.

Go back to `RepLogApp`. Let's refactor things first: create a new method called `setSuccessMessage` with a message argument. Inside, set the state.

```
112 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 83
84   setSuccessMessage(message) {
85     this.setState({
86       successMessage: message
87     });
... lines 88 - 93
94   }
... lines 95 - 112
```

We're making this change so that we can re-use our cool success message feature in the future. Above, instead of setting the `successMessage` in the object, use `this.setSuccessMessage()` and paste the text there.

```
112 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 39
40   handleAddRepLog(item, reps) {
... lines 41 - 49
50     createRepLog(newRep)
51     .then(repLog => {
... lines 52 - 60
61       this.setSuccessMessage('Rep Log Saved!');
62     })
63   ;
64 }
... lines 65 - 112
```

## [Watch out for Multiple Re-Renders](#)

But! There is a downside to what we just did! *Every* time you change the state, React re-renders our component. Thanks to this change, it's going to re-render once for this state and then *again* right after. That is *probably* not something you need to worry about. But, as your applications grow bigger and bigger and bigger, you *should* be aware when you're triggering your app to re-render. This is *especially* important because, when a component like `RepLogApp` re-renders, *all* of its children are *also* re-rendered, even if the props being passed to them don't change. And yes, there *are* ways to optimize this. But, for now, just be aware that re-rendering requires CPU. If you re-render a big app too often, it could slow down. But, there *are* ways to optimize.

## [Clearing the Message with `setTimeout\(\)`](#)

Back in `setSuccessMessage()`, to clear the message, use `setTimeout()`, pass it an arrow function, and use `this.setState()` to reset `successMessage` back to empty quotes. Let's do that after 3 seconds.

112 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 84
85     this.setState({
... lines 86 - 88
89     setTimeout(() => {
90         this.setState({
91             successMessage: "
92         });
93     }, 3000)
94 }
... lines 95 - 112
```

Ok! Let's give this a try! Refresh and lift my big fat cat 5 times. Success! And... gone!

## Clearing the Timeout

So easy! Except... yes... it was *too* easy. There's an edge case: if `setSuccessMessage()` is called once, then called again 2 seconds later, the second message will disappear after only 1 second! It's a minor detail, but we can code this better.

Basically, *before* we call `setTimeout`, we want to make sure to *clear* any previous timeout that may be waiting to fire. The `setTimeout()` function returns an integer, which we can use to clear it. To keep track of that value, in the constructor, initialize a new property: `this.successMessageTimeoutHandle = 0`.

119 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 7
8     constructor(props) {
... lines 9 - 18
19     this.successMessageTimeoutHandle = 0;
... lines 20 - 24
25 }
... lines 26 - 119
```

This has *nothing* to do with React: we're just taking advantage of our object to store some data. Oh, and the value 0 is just a "null" value in disguise: if we pass this to `clearTimeout()`, nothing will happen.

Back down in `setSuccessMessage`, before `setTimeout`, add `clearTimeout(this.successMessageTimeoutHandle)`. To set that property, add `this.successMessageTimeoutHandle =` before `setTimeout()`.

119 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 88
89     setSuccessMessage(message) {
... lines 90 - 93
94         clearTimeout(this.successMessageTimeoutHandle);
95         this.successMessageTimeoutHandle = setTimeout(() => {
... lines 96 - 100
101     }
... lines 102 - 119
```

And finally, to be *completely* on top of things, inside the callback, after we reset the state, set the timeout handle back to 0.

119 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 94
95     this.successMessageTimeoutHandle = setTimeout(() => {
... lines 96 - 98
99         this.successMessageTimeoutHandle = 0;
100     }, 3000)
... lines 101 - 119
```

## Cleaning up Your Component: componentWillUnmount()

And... we're done! Our message will always disappear a full 3 seconds after the *last* success message has been set. Except... yea... there is *still* one teenie, tiny problem... and *this* time, it's special to React.

Right now, RepLogApp will *always* be rendered on the page. But, that's not true of React components in general. For example, we could choose to *only* render the RepLogCreator component after clicking a button. Or, if you're using React Router so that users can navigate to different "pages", then even RepLogApp would be rendered and unrendered as the user navigates.

Because of this, *if* your component is removed from the page, you need to ask yourself:

Is there anything I need to clean up?

The answer is usually... no! But, setTimeout() *would* cause a problem. Why? Basically, if setState() is called on a component that is *not* rendered to the page, React kinda freaks out. Thanks to setTimeout(), that could happen if the component was removed right after setting a success message.

It's not a big deal, but let's clean this up. Scroll up to componentDidMount() and add a new method: componentWillUnmount().

```
119 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 36
37   componentWillUnmount() {
... line 38
39   }
... lines 40 - 119
```

This is another one of those magic lifecycle functions: componentDidMount is called right after your component is rendered to the page. componentWillUnmount is called right before it's *removed*. It's your chance to clean stuff up.

Let's do that: clearTimeout(this.successMessageTimeoutHandle).

```
119 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 36
37   componentWillUnmount() {
38     clearTimeout(this.successMessageTimeoutHandle);
39   }
... lines 40 - 119
```

Honestly, this isn't that common. But, keep it in mind. Another example could be if you used an external library to add some cool feature directly to a DOM element. If you want to clean that up, this is the place to do it.

# Chapter 35: Updating Deep State Data

Oh man, I let a bug crawl into our app. When we delete a rep log, it goes away, but, yuck, we get a big error:

Unexpected end of JSON input

This comes from `rep_log_api.js`. We call `response.json()`... which works *great* when the response is *actually* JSON. But, our delete endpoint returns *nothing*.

To fix this, we could create two different functions: one that decodes JSON and one that doesn't. But, I'll just make our code a bit fancier so it doesn't explode.

Use `return response.text()`: this returns a Promise where the data is the raw response content. Chain `.then` and use an arrow function with a `text` argument. Here, if `text`, return `JSON.parse(text)`, else empty quotes.

```
36 lines | assets/js/api/rep_log_api.js
... lines 1 - 4
5     .then(response => {
6         // decode JSON, but avoid problems with empty responses
7         return response.text()
8         .then(text => text ? JSON.parse(text) : "")
9     });
... lines 10 - 36
```

Go over, refresh and... delete! Ok, much better.

## Success Message on Delete

We have this cool system where we can lift our cat 26 times and see this temporary success message. So, we might as well do the same thing when we delete. And this is easy! Inside of `RepLogApp`, down in `handleDeleteRepLog`, chain off the `delete: .then()`, an arrow function, and this `setSuccessMessage()`: Item was Un-lifted.

```
122 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 76
77     handleDeleteRepLog(id) {
78         deleteRepLog(id)
79         .then(() => {
80             this.setSuccessMessage("Item was Un-lifted!");
81         });
... lines 82 - 89
90     }
... lines 91 - 122
```

Cool! Move back and try it! Success... message.

## "Ghosting" the Deleting Row

We *could* be satisfied with our loading & success message setup. But... if you want... we can get fancier! Right now, we delete the rep log state immediately, but we don't show the success message until *after* the AJAX call finishes. If you want that to feel more synchronized, we *could* move the `setState()` call so that it fires only when the rep log is *actually* deleted.

122 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 77

```
78   deleteRepLog(id)
79   .then(() => {
80     // remove the rep log without mutating state
81     // filter returns a new array
82     this.setState((prevState) => {
83       return {
84         repLogs: prevState.repLogs.filter(repLog => repLog.id !== id)
85       };
86     });
87
88     this.setSuccessMessage("Item was Un-lifted!");
89   });
```

... lines 90 - 122

But, we're *trading* problems. Refresh again. When you click delete, there's a slight pause before the user gets *any* feedback. I'll add a few more items to the list real quick so that we can keep deleting.

Anyways, here's an idea of how we could improve this: when the user clicks delete, let's immediately change the *opacity* on the row that's being deleted, as a sort of "loading" indication.

Go into RepLogList: *this* is where we render the tr elements. So, imagine if there were a field on each repLog called isDeleting. If there were, we could say style={}, create an object, and set opacity: if isDeleting is true, use .3 else 1.

67 lines | [assets/js/RepLog/RepLogList.js](#)

... lines 1 - 22

```
23   return (
... line 24
25     {repLogs.map((repLog) => (
26       <tr
... lines 27 - 29
30         style={{
31           opacity: repLog.isDeleting ? .3 : 1
32         }}
33       >
... lines 34 - 42
43     )})
... lines 44 - 55
56   );
... lines 57 - 67
```

*This* was easy. The interesting part of this problem is *how* we can add that new isDeleting field. Well, it *looks* simple at first: at the top of handleDeleteRepLog, before we call deleteRepLog(), we want to set the state of *one* of our rep logs to have isDeleting: true.

But... hmm... this is tricky. First, we need to find the *one* rep log by its id. Then, we need to set this flag, but without *mutating* that object *or* the array that it's inside of! Woh!

Here's the trick: use this.setState(), but pass it an arrow function with the prevState arg. We're doing this because our new state will *depend* on the old state. Return the new state we want to set, which is the repLogs key.

137 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 76

```
77   handleDeleteRepLog(id) {
78     this.setState((prevState) => {
79       return {
80         ... lines 80 - 86
87       };
88     });
89     ... lines 89 - 137
```

To *not* mutate the state, we basically want to create a *new* array, put all the existing rep logs inside of it, and update the *one* rep log... um... without actually updating it. Sheesh.

This is another one of those moments where you can understand why React can be so darn hard! But, the fix is easy, and it's an old friend: `map`! Use `prevState.repLogs.map()` with a `repLog` argument to the arrow function.

137 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 77

```
78   this.setState((prevState) => {
79     return {
80       repLogs: prevState.repLogs.map(repLog => {
81         ... lines 81 - 85
86       })
87     };
88   });
89   ... lines 89 - 137
```

The `map` function will return a *new* array, so that handles *part* of the problem. Inside, if `repLog.id !==` the id that's being deleted, just return `repLog`. And finally, we need to basically "clone" this last rep log and set the `isDeleting` flag on the new object. The way to do that is with `return Object.assign()` passing it an empty object, `repLog`, then the fields to update: `isDeleting: true`.

137 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 79

```
80     repLogs: prevState.repLogs.map(repLog => {
81       if (repLog.id !== id) {
82         return repLog;
83       }
84
85       return Object.assign({}, repLog, {isDeleting: true});
86     })
87     ... lines 87 - 137
```

As I mentioned earlier, `Object.assign()` is like `array_merge` in PHP: the 3rd argument is merged into the second, and then that's merged into the first. The *key* is the strange first argument: the empty object. Thanks to that, we're creating a *new* object, and then all the data is merged into *it*. The `repLog` is *not* modified.

Phew! But... awesome! We've now learned how to *add* to an array, remove from an array, and even *change* an *object* inside an array, *all* without mutation. If your state structure is deeper than a simple object inside an array, it's probably too deep. In other words, you now know how to handle the most common, tough, state-setting situations.

Let's temporarily add a `return` statement below so we can *really* see if this is working. Ok, move over and refresh! Hit delete: that looks awesome! Our update worked perfectly.

Go back and remove the `return`.

# Chapter 36: Server Validation & fetch Failing

Earlier, we talked about the three types of validation. First, HTML5 validation with things like the required attribute. It's dead-simple to setup, but limited. Second, custom client-side validation, which we added because we wanted to make sure the users enter a *positive* quantity. And third, *of course*, the *one* type of validation you *must* have: server-side validation.

## Our Server Side Validation

Look at the RepLog entity. We already have a few important validation constraints: reps cannot be blank and needs to be a positive number, and the item also cannot be blank.

Thanks to HTML5 validation & client-side validation, we are *already* preventing these bad values from *even* being submitted to the server. And, of course, if some annoying hacker wants to send bad values to the API, sure, they totally could. But *then* our server-side validation would be there to tell them to bugger off.

However, a lot of time, I either skip client-side validation entirely, or just add it for a *few* things, but not *everything*. And, in that case, if an API request fails because of failed server-side validation, our React app needs to read those errors from the server and tell the user.

Check out RepLogController. We're using the form system, but that's not important. Nope, the *really* important thing is that we, somehow, get a RepLog object that's populated with data, and run it through the validation system. The form does this for us. But if you were manually setting up the object or using the serializer to deserialize, you could pass the object directly to the validation system to get back a collection of errors.

In this application, I added a shortcut method called getErrorsFromForm, which lives in BaseController. This recursively loops over my errors to create a big array of errors, where the key is the name of the field. This is what's returned from our API.

When you use the form system, there is *one* other way to add validation, which is often forgotten: on the form itself: RepLogType. The ChoiceType is normally used to render a select element where choices is where you define the valid options. When used in an API, if we submit a value that is *not* in choices, the form will fail validation.

## Sending a Bad "Item"

For testing purposes, let's purposely make this easy to do. In RepLogCreator, find the itemOptions array: these items match what's configured inside the form. Add a fake one: invalid\_item with text Dark Matter.

```
101 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 4
5   constructor(props) {
... lines 6 - 14
15   this.itemOptions = [
... lines 16 - 19
20     { id: 'invalid_item', text: 'Dark Matter' }
21   ];
... lines 22 - 23
24   }
... lines 25 - 101
```

The server will *not* like this value. Let's try it anyways! Move over, select "Dark Matter", 10 and... ah! Run! Woh!

Ok, two things. First, you can see the request failed: 400 bad request. Great! Our server-side validation is working and you can see the message in the response. But, second, React exploded in a crazy way! Something about how each child in an array should have a unique "key" prop, from RepLogList.

We know that error... but why is it suddenly happening?

## Fetch is Afraid of Failure



There's *one* simple explanation and it's hiding in `rep_log_api.js`. If you used jQuery's AJAX function, you might remember that if the server returns an error status code, a failure callback is executed instead of your `.then()`, success callback. That makes sense: the request failed!

But... `fetch()` does *not* do this. *Even* if the server sends back a 400 or 500 error... `fetch` thinks:

We did it! We made a request! Yaaaay! Let's execute the `.then()` success callbacks!

Thanks to that, our app parsed the JSON, thought it contained a rep log, tried to add it to state, and things went bananas.

## Making fetch Fail

This behavior... isn't great. So, I like to fix it: I'll paste in a new function called `checkStatus()`. If we call this function and the status code is not 200 or 300, it creates an `Error` object, puts the response onto it, and *throws* it. By the way, you could change this logic to *also* throw an error for 300-level status codes, that's actually how jQuery works.

```
48 lines | assets/js/api/rep_log_api.js
... lines 1 - 12
13 function checkStatus(response) {
14   if (response.status >= 200 && response.status < 400) {
15     return response;
16   }
17
18   const error = new Error(response.statusText);
19   error.response = response;
20
21   throw error
22 }
... lines 23 - 48
```

To use this, back up in `fetchJson()`, add this handler: `.then(checkStatus)`.

```
48 lines | assets/js/api/rep_log_api.js
1 function fetchJson(url, options) {
2   return fetch(url, Object.assign({
... line 3
4     }, options))
5     .then(checkStatus)
6     .then(response => {
... lines 7 - 9
10   });
11 }
... lines 12 - 48
```

Let's try it! Refresh, select our bad item, a number and... yes! This is a much more obvious message:

Uncaught (in promise) Error: Bad Request at checkStatus

Now that `fetch` is behaving better, let's *use* this error response to add a message for the user.

# Chapter 37: Displaying Server Validation Errors

When server-side validation fails, the API returns a 400 status code with the *details* of the error in the response. And thanks to the change we just made, `fetch()` *now* throws an error, which we can handle!

Open `RepLogApp`: inside of `handleAddRepLog`, if the call to `createRepLog` fails, we need to grab the validation error messages and put them on the screen. And now that our Promise can fail, we can do this with a `.catch()`. Pass this an arrow function. For now, just `console.log(error.response)`.

```
137 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 44
45   handleAddRepLog(item, reps) {
... lines 46 - 54
55     createRepLog(newRep)
56     .then(repLog => {
... lines 57 - 66
67     })
68     .catch(error => {
69       console.log(error.response);
70     })
... line 71
72   }
... lines 73 - 137
```

Let's see what that looks like: refresh, try dark matter again and... cool! We have the Response object!

Let's decode this *just* like we did before with success: `error.response.json()`. This returns *another* Promise, so add `.then` with an `errorsData` argument for the next arrow function. Log that... then let's go see what it looks like: dark matter, 10 times... perfect!

```
139 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 67
68     .catch(error => {
69       error.response.json().then(errorsData => {
70         console.log(errorsData);
71       })
72     })
... lines 73 - 139
```

It has an `errors` key, and a list of errors below where the *key* tells us which field this is for. So, how can we print this onto the screen? Well... it depends on how fancy you want to get. You *could* use the key of each error to find the field it belongs to, and render the error in that place. Or, you could print all of the errors on *top* of the form. Or, you could be *even* lazier like me and just print the first error above the form.

## Adding the Error State

To do that, we need new state inside of `RepLogApp` to keep track of the current "rep log validation" error message. Add one called `newRepLogValidationErrorMessage`. set to empty quotes.

146 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 7
8    constructor(props) {
... lines 9 - 10
11    this.state = {
... lines 12 - 17
18      newRepLogValidationErrorMessage: "
19    };
... lines 20 - 25
26  }
... lines 27 - 146
```

But wait, this is interesting. When we added client-side validation, we stored it in RepLogCreator and *it* handled *all* of that logic. But because RepLogApp is the only component that's aware of the server, this is state that *it* needs to handle. And, it's not really *form* validation logic: remember RepLogApp doesn't even *know* our app uses a form. Nope, it's really a *business* logic validation failure: something, *somehow* tried to save a rep log with invalid data.

Copy that name and go down to handleAddRepLog(). First, *if* we're successful, in case there was *already* a validation message on the screen, we need to set it *back* to empty quotes to remove it.

146 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 55
56    createRepLog(newRep)
57      .then(repLog => {
58        this.setState(prevState => {
... lines 59 - 60
61          return {
... lines 62 - 63
64            newRepLogValidationErrorMessage: "",
65          };
66        });
... lines 67 - 68
69      })
... lines 70 - 146
```

Down in catch(), add const errors = errorsData.errors. Then, to get *just* the first error... um... it's actually a bit tricky. Silly JavaScript! Use const firstError = errors[Object.keys(errors)[0]].

Wow! We need to do this because errors isn't an *array*, it's an object with keys. Use this in the setState() call: newRepLogValidationErrorMessage set to firstError.

146 lines | [assets/js/RepLog/RepLogApp.js](#)

```
... lines 1 - 69
70    .catch(error => {
71      error.response.json().then(errorsData => {
72        const errors = errorsData.errors;
73        const firstError = errors[Object.keys(errors)[0]];
74
75        this.setState({
76          newRepLogValidationErrorMessage: firstError
77        });
78      })
79    })
... lines 80 - 146
```

## Passing & Using the Error State

Done! As you know, the new state is instantly passed down to the child as a prop. But we need to *use* this state in

RepLogCreator: I want to put the message right above the form.

Ok! Time to pass some props around! Step 1: define the new prop type as a required string. Step 2: destructure this to a new variable. And step 3, pass to RepLogCreator. But wait! Let's change the name `validationErrorMessage` to then the variable.

```
112 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18 export default function RepLogs(props) {
19   const {
... lines 20 - 30
31     newRepLogValidationErrorMessage
32   } = props;
... lines 33 - 38
39   return (
... lines 40 - 87
88     <RepLogCreator
... line 89
90       validationErrorMessage={newRepLogValidationErrorMessage}
91     />
... lines 92 - 94
95   );
96 }
... line 97
98 RepLogs.propTypes = {
... lines 99 - 109
110   newRepLogValidationErrorMessage: PropTypes.string.isRequired,
111 };
```

Why the name change? Well... if you think about it, even though we called this component `RepLogCreator`, there's nothing about the component that's specific to *creating* rep logs. We could easily reuse it later for editing *existing* rep logs... which is awesome!

All `RepLogCreator` cares about is that we're passing it *the* validation error message: it doesn't care that it's the result of creating a *new* rep log versus editing an existing one.

Anyways, let's go use this: add the prop type: `validationErrorMessage` as a string that's required. Then, destructure it. Oh, we don't have any props destructuring yet. No problem - use `const {} = this.props` and *then* add `validationErrorMessage`. I typed that a bit backwards so PhpStorm would auto-complete the variable name.

```
108 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 3
4 export default class RepLogCreator extends Component {
... lines 5 - 53
54   render() {
... line 55
56     const { validationErrorMessage } = this.props;
... lines 57 - 100
101   }
102 }
... line 103
104 RepLogCreator.propTypes = {
... line 105
106   validationErrorMessage: PropTypes.string.isRequired,
107 };
```

Finally, *just* inside the form, use our trick: `validationErrorMessage &&`, some parenthesis and a div with `className="alert alert-danger"` and the message inside.

108 lines | [assets/js/RepLog/RepLogCreator.js](#)

```
... lines 1 - 53
54   render() {
... lines 55 - 57
58     return (
59       <form onSubmit={this.handleFormSubmit}>
60         {validationErrorMessage && (
61           <div className="alert alert-danger">
62             {validationErrorMessage}
63           </div>
64         )}
... lines 65 - 98
99       </form>
100     );
101   }
... lines 102 - 108
```

We're only printing the first message. But if you wanted to print a list instead, it's no big deal: just use the `map()` function like normal to render all of them.

Let's see if it works! Move over and make sure everything is refreshed. Select dark matter, 10, and... yes! We got it! Hmm, except, because we're not printing the error *next* to the field... it's not super obvious what I need to fix! *If* you're going to be lazy like us, you need to make sure the errors are descriptive.

Go into `src/Form/Type/RepLogType.php`. Most validation comes from our `RepLog` entity. But the form fields *themselves* can add some "sanity" validation. To customize the message, add an option: `invalid_message` set to "Please lift something that is understood by our scientists."

32 lines | [src/Form/Type/RepLogType.php](#)

```
... lines 1 - 13
14   public function buildForm(FormBuilderInterface $builder, array $options)
15   {
16     $builder
... line 17
18     ->add('item', ChoiceType::class, array(
... lines 19 - 20
21       'invalid_message' => 'Please lift something that is understood by our scientists.'
22     ))
... line 23
24   }
... lines 25 - 32
```

Much easier to understand! Try that: refresh, choose dark matter and... got it!

Except... hmm: when you get a validation error, the "Lifting to the database" loading message is still there! It... shouldn't be. Let's fix that and learn about a super-useful language feature called "object rest spread".

# Chapter 38: ...Object Rest Spread

When a user submit an invalid form, we get a nice error message... but our cool "lifting to the database" message stays! Totally confusing! It looks like we're *still* trying to save the new rep log. Let's fix that!

In RepLogApp, the state that controls this is called `isSavingNewRepLog`. When the AJAX call is successful, we set this back to false. We need to *also* set this back to false inside the catch. And yes, fixing this is as *easy* as just copying this key and pasting it below. Sure, this would duplicate that key in two places... but, that's *super-minor* duplication: no big deal.

Except... I want to learn a super-fun new language feature. To show that, let's fix this in a *slightly* fancier way. Above the AJAX call, set `const newState =` an object with `savingNewRepLog` set to false.

```
148 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 45
46   handleAddRepLog(item, reps) {
... lines 47 - 55
56     const newState = {
57       isSavingNewRepLog: false
58     };
... lines 59 - 82
83   }
... lines 84 - 148
```

This represents the new state that we want to apply in *all* situations: success or failure. In other words, we want to *merge* this state into whatever is being set in success and also down in catch.

How can you merge objects in JavaScript? We've seen it before: `Object.assign()`. Check it out: `return Object.assign()`. For the first argument, copy the new state and paste. For the second argument, use `newState`.

```
148 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 58
59   createRepLog(newRep)
60     .then(repLog => {
61       this.setState(prevState => {
... lines 62 - 63
64         return Object.assign({
65           repLogs: newRepLogs,
66           newRepLogValidationErrorMessage: "",
67         }, newState);
68       });
... lines 69 - 70
71     })
... lines 72 - 148
```

`Object.assign()` will merge the data from `newState` *into* the first object and return it. Perfect!

Repeat this in catch: add `Object.assign()`, then `newState`.

```

148 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 71
72     .catch(error => {
73         error.response.json().then(errorsData => {
... lines 74 - 76
77         this.setState(Object.assign({
78             newRepLogValidationErrorMessage: firstError
79         }, newState));
80     })
81 })
... lines 82 - 148

```

Let's go make sure this works: refresh, select our bad data and... cool. It shows for just a second, then disappears.

## [Installing & Configuring babel-plugin-transform-object-rest-spread](#)

`Object.assign()` is really great. We also used it earlier to merge two objects *without* modifying the original object. *That* was *super* important.

The only problem with `Object.assign()` is that it's... kinda confusing to look at, *especially* if you need to use it to avoid mutation.

Ok, idea time: what if we could do this: remove the `Object.assign()`, return a normal object, but then, add `...newState`.

```

150 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 59
60     .then(repLog => {
61         this.setState(prevState => {
... lines 62 - 63
64         return {
65             ...newState,
... lines 66 - 67
68         }
69     });
... lines 70 - 71
72 })
... lines 73 - 150

```

That would be cool, right? I mean, we *already* do this for arrays! But... Webpack explodes: the "spread" syntax does *not* work for objects.

Or does it?! Google for "babel plugin transform object rest spread" and find the Babel documentation page. The feature we're "dreaming" about is called "object rest spread". It is *not* an official ECMAScript feature. But, it's currently a proposed, "draft" feature that's in a late stage. There's no promises, but that means it will *likely* become a real feature in a future ECMAScript version.

And, because the JS world is a bit nuts, you don't *need* to wait! We can *teach* Babel how to understand this syntax. Copy the package name, find your terminal and run:

```
$ yarn add babel-plugin-transform-object-rest-spread --dev
```

Oh, and as I mentioned before, most of these Babel plugins will have a slightly new name in the future: `@babel/plugin-transform-object-rest-spread`. But, it's really the same library.

When you work with Babel, you typically configure it with a `.babelrc` file. But, Encore does this for us! Open `webpack.config.js`: the `configureBabel()` function allows us to *extend* its configuration. Add `babelConfig.plugins.push()` and paste the name.

43 lines | [webpack.config.js](#)

... lines 1 - 3

4 Encore

... lines 5 - 29

30 .configureBabel((babelConfig) => {

... lines 31 - 36

37 babelConfig.plugins.push('transform-object-rest-spread');

38 })

... lines 39 - 43

In the future, if you download the new `@babel/plugin-transform-object-rest-spread` library, the plugin name will be the full library name, starting with the `@babel` part. Just follow the docs.

Head back to the tab that's running Encore. Yep, it's *super* angry. Stop and re-run this command:

```
$ yarn run encore dev-server
```

And... it works! That's awesome! Babel now understands this syntax.

But... PhpStorm is still angry: ESLint parsing error. No worries: we just need to tell ESLint that this syntax is cool with us. Open `.eslintrc.js`. Under `ecmaFeatures`, add `experimentalObjectRestSpread` set to `true`.

### Tip

On ESLint version 6 or higher, you only need to change the `ecmaVersion` to 2018. You do not need to add the `experimentalObjectRestSpread` option because it's already enabled.

21 lines | [.eslintrc.js](#)

1 module.exports = {

... line 2

3 parserOptions: {

... lines 4 - 5

6 ecmaFeatures: {

... line 7

8 experimentalObjectRestSpread: true

9 }

10 },

... lines 11 - 19

20 };

Deep breath: go back to `RepLogApp`. And... sweet! The error is gone!

## [Using the Object Rest Spread](#)

Let's finish this! Down in `catch`, remove `Object.assign()`, remove the second argument and add `...newState`.



```

150 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 72
73     .catch(error => {
74         error.response.json().then(errorsData => {
... lines 75 - 77
78         this.setState({
79             ...newState,
... line 80
81         });
82     })
83 })
... lines 84 - 150

```

And one more time: scroll down to `handleDeleteRepLog()`. We don't need this weird code anymore! Just return a new object with `...repLog` then `isDeleting: true`.

```

150 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 92
93     handleDeleteRepLog(id) {
94         this.setState((prevState) => {
95             return {
96                 repLogs: prevState.repLogs.map(repLog => {
... lines 97 - 100
101                 return {...repLog, isDeleting: true};
102             })
103         };
104     });
... lines 105 - 117
118 }
... lines 119 - 150

```

I /love that. And even better, when we refresh, it's not broken! We rock!

# Chapter 39: Passing Data from your Server to React

Look inside RepLogCreator. The items in the drop-down are hardcoded. But, in reality, we can't just put whatever we want here: there is a specific set of valid options stored in our backend code.

We *already* know this is true because the last option is totally fake! When we send that to the server, it hits us with a validation error.

So, here is the question: instead of hardcoding these options, should we load them dynamically from the server?

The answer is... maybe? If these options won't ever change or change often, it's *really* not that big of a deal. The advantage is... simplicity!

But, if they *will* change often, or if having an invalid one on accident would cause a *hugely* critical or embarrassing bug, then yea, you *should* load them dynamically... so that you can sleep soundly at night.

## Refactoring Data to the Top Level Component

Whenever your JavaScript app needs server data, there are two options. But, they both start the same way: by moving our itemOptions up into RepLogApp, which is the only component that's even *aware* a server exists!

Copy itemOptions and then open RepLogApp. On top, initialize a new itemOptions state set to that array.

```
157 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 7
8   constructor(props) {
  ... lines 9 - 10
11   this.state = {
  ... lines 12 - 18
19     itemOptions: [
20       {id: 'cat', text: 'Cat'},
21       {id: 'fat_cat', text: 'Big Fat Cat'},
22       {id: 'laptop', text: 'My Laptop'},
23       {id: 'coffee_cup', text: 'Coffee Cup'},
24       {id: 'invalid_item', text: 'Dark Matter'}
25     ]
26   };
  ... lines 27 - 32
33 }
  ... lines 34 - 157
```

Because all state is automatically passed as props to RepLogs, go there and add the new prop type: itemOptions as an array that is required.

```
115 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 99
100 RepLogs.propTypes = {
  ... lines 101 - 112
113   itemOptions: PropTypes.array.isRequired
114 };

```

Above, destructure that, then, below, pass it down to RepLogCreator as itemOptions={itemOptions}.

```

115 lines | assets/js/RepLog/RepLogs.js
... lines 1 - 17
18 export default function RepLogs(props) {
19   const {
... lines 20 - 31
32     itemOptions
33   } = props;
... lines 34 - 39
40   return (
... lines 41 - 88
89     <RepLogCreator
... lines 90 - 91
92       itemOptions={itemOptions}
93     />
... lines 94 - 96
97   );
98 }
... lines 99 - 115

```

Copy the prop type, then do the same in RepLogCreator: define the prop type at the bottom, then go to the top of the function to destructure out itemOptions.

```

101 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 3
4 export default class RepLogCreator extends Component {
... lines 5 - 45
46   render() {
... line 47
48     const { validationErrorMessage, itemOptions } = this.props;
... lines 49 - 92
93   }
94 }
... line 95
96 RepLogCreator.propTypes = {
... lines 97 - 98
99   itemOptions: PropTypes.array.isRequired
100 };

```

Below, use the local itemOptions variable for the map function.

```

101 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 45
46   render() {
... lines 47 - 49
50     return (
... lines 51 - 68
69       {itemOptions.map(option => {
... lines 70 - 91
92     });
93   }
... lines 94 - 101

```

When we refresh... cool! The options aren't dynamic yet, but they *are* stored as state. If you change a value... yep! It shows up.

## [Two Ways to Load Server Data](#)

Now that the data lives in our top-level component, let's talk about the *two* ways we can load this dynamically from the server. Actually, we already know the first way - we did it with `repLogs`! We could set `itemOptions` to an empty array, then make an AJAX call from inside `componentDidMount()`. Of course, we would *also* need to create an API endpoint, but that's no big deal.

Or, you could use the second option: render a global variable inside Twig and read it in JavaScript. The advantage is that *this* data is available immediately: you can populate your app with some initial data, without waiting for the AJAX call.

## Passing the Options as Props

Copy the options again and go into the *entry* file: `rep_log_react.js`. This will *not* be the final home for these options - but it will get us one step closer. Create a new `itemOptions` variable and paste! *Now*, pass these as a new prop:

`itemOptions={itemOptions}`.

```
22 lines | assets/js/rep_log_react.js
... lines 1 - 6
7   const itemOptions = [
8     {id: 'cat', text: 'Cat'},
9     {id: 'fat_cat', text: 'Big Fat Cat'},
10    {id: 'laptop', text: 'My Laptop'},
11    {id: 'coffee_cup', text: 'Coffee Cup'},
12    {id: 'invalid_item', text: 'Dark Matter'}
13  ];
... line 14
15  render(
16    <RepLogApp
... line 17
18      itemOptions={itemOptions}
19    />,
... line 20
21  );
```

Thanks to this, `RepLogApp` will now *receive* a new `itemOptions` prop. Remove the state *entirely*.

At the bottom, set this prop type: `itemOptions` is an array, and you *could* make it required - I'll talk more about that in a minute.

```
152 lines | assets/js/RepLog/RepLogApp.js
... lines 1 - 147
148  RepLogApp.propTypes = {
... line 149
150    itemOptions: PropTypes.array,
151  };
```

Oh, and this is cool! We deleted the `itemOptions` state but *added* an `itemOptions` prop. And because we're passing *all* props & state to `RepLogs`, it is *still* receiving an `itemOptions` prop. In other words, this just works.

Side note: I *originally* set `itemOptions` to state because this is *needed* if you wanted to make an AJAX call to populate them: they would be empty at first, then *change* a moment later when the request finished. But really, `itemOptions` don't ever need to change. So once we passed them as props, we could remove the state.

But, if the item options really *did* need to be state - if this was something that changed throughout the life of our app - we could *still* use this strategy. We could use the `itemOptions` prop to set the *initial* value of the state. This literally means that you *would* still have an `itemOptions` state, and it would be initialized to `this.props.itemOptions`.

I might even call the prop `initialItemOptions` for clarity... though if you *do* have a state and prop with the same name, that's fine. If you look down in `render()`, the state would override the prop, because the `...state` comes second.

## Setting Initial Props

Anyways, down in `propTypes`, I did *not* make `itemOptions` a *required* prop. In a real application, I probably would: I don't want

the select to *ever* be empty. But sometimes, you *will* create a component where you want a prop to be truly optional. And in those cases, you need to be careful: if we didn't pass the `itemOptions` prop, our code would explode! `itemOptions` would be undefined instead of an array... which would be a problem when `RepLogCreator` calls `.map` on it.

To solve this, you can give any prop a default value. It's super easy: add `RepLogApp.defaultProps` = an object with `itemOptions` set to an empty array.

155 lines | [assets/js/RepLog/RepLogApp.js](#)

... lines 1 - 152

```
153 RepLogApp.defaultProps = {  
154   itemOptions: []  
155 };
```

Ok: we have *removed* the hardcoded `itemOptions` from our React app entirely. But... we're not done: they're still hardcoded in `rep_log_react.js`. We need to fetch this value dynamically from the server. Let's do that next!

# Chapter 40: Passing Server Data to React Props

We need to load this `itemOptions` data dynamically from the server. Copy the options and then find the template for this page: `templates/lift/index.html.twig`.

At the bottom, you'll find the script that loads our app. *Before* this, create a new *global* variable. So, use the window object: `window.REP_LOG_APP_PROPS` = an object with `itemOptions` set to our options.

```
81 lines | templates/lift/index.html.twig
... lines 1 - 63
64 {% block javascripts %}
... lines 65 - 66
67 <script>
68     window.REP_LOG_APP_PROPS = {
69         itemOptions: [
70             {id: 'cat', text: 'Cat'},
71             {id: 'fat_cat', text: 'Big Fat Cat'},
72             {id: 'laptop', text: 'My Laptop'},
73             {id: 'coffee_cup', text: 'Coffee Cup'},
74             {id: 'invalid_item', text: 'Dark Matter'}
75         ]
76     }
77 </script>
... lines 78 - 79
80 {% endblock %}
```

Now, go back to `rep_log_react.js` delete the old constant and, below, use `window.REP_LOG_APP_PROPS.itemOptions`.

```
14 lines | assets/js/rep_log_react.js
... lines 1 - 6
7 render(
8     <RepLogApp
... line 9
10     itemOptions={window.REP_LOG_APP_PROPS.itemOptions}
11     />,
... line 12
13 );
```

## When is it Ok to Use window?

This *will* work... but... now I have a question. Why couldn't we just copy this code and use it in `RepLogCreator` instead of passing the prop down all the levels? You *could*. But, as a best practice, I don't want *any* of my React components to use variables on the window object. The only place where I want you to feel safe using the global window object is inside of your entry point: it should grab all of the stuff you need, and pass it *into* your React app.

Like everything, don't live and die by this rule. But, the window object is a global variable. And, just like in PHP, while global variables are easy to use, they make your code harder to debug and understand. Use them in your entry, but that's it.

## Spreading all of the Props

Back in the template, I built the `REP_LOG_APP_PROPS` variable so that we could, in theory, set *other* props on it. For example, add `withHeart: true`.

```

82 lines | templates/lift/index.html.twig
... lines 1 - 66
67     <script>
68         window.REP_LOG_APP_PROPS = {
... lines 69 - 75
76         withHeart: true
77     }
78 </script>
... lines 79 - 82

```

In the entry file, to read this, we could of course use `window.REP_LOG_APP_PROPS.withHeart`. Or... we can be way cooler! Use spread attributes: `...window.REP_LOG_APP_PROPS`.

```

14 lines | assets/js/rep_log_react.js
... lines 1 - 6
7  render(
8    <RepLogApp
... line 9
10    {...window.REP_LOG_APP_PROPS}
11  />,
... line 12
13 );

```

Suddenly all of the keys on that object will be passed as props! And this is cool: set `shouldShowHeart` to `false`. Hmm: we're now passing `withHeart=false`... but thanks to the spread prop, we're passing that prop *again* as `true`.

```

14 lines | assets/js/rep_log_react.js
... lines 1 - 4
5  const shouldShowHeart = false;
... lines 6 - 14

```

When you do this, the *last* prop always wins. Yep, we *do* see the heart.

This is a *cool* way to render a component with initial data that comes from the server.

## Dumping JavaScript in Twig

Well, the data isn't *quite* dynamic yet. Let's finally finish that. Open the form: `src/Form/Type/RepLogType.php`. The choices options is the data that we want to send to React. Copy `RepLog::getThingsYouCanLiftChoices()`.

Then, go into the controller that renders this page - `LiftController` and find `indexAction()`. First, let's `dump()` that function to see what it looks like.

```

56 lines | src/Controller/LiftController.php
... lines 1 - 18
19  public function indexAction(Request $request, RepLogRepository $relogRepo, UserRepository $userRepo)
20  {
... lines 21 - 22
23      dump(RepLog::getThingsYouCanLiftChoices());die;
... lines 24 - 27
28  }
... lines 29 - 56

```

Move over and refresh! Interesting! It's an array... but it doesn't quite look right. Let's compare this to the structure we want. Ok, each item has an id like `cat` or `fat_cat`. That is the *value* on the array. We also need a text key. My app is using the translator component. The *keys* on the dumped array need to be run through the translator to be turned into the English text.

Actually, the details aren't important. The point is this: our app *does* have the data we need... but we need to "tweak" it a little bit to match what our React app is expecting.

To do that, go back to the controller. To save us some tedious work, I'll paste in some code. This code uses the `$translator`. To get that, add a new controller argument: `TranslatorInterface $translator`.

```
66 lines | src/Controller/LiftController.php
... lines 1 - 19
20     public function indexAction(Request $request, RepLogRepository $replLogRepo, UserRepository $userRepo, TranslatorInterface $translator)
21     {
... lines 22 - 23
24         $repLogAppProps = [
25             'itemOptions' => [],
26         ];
27         foreach (RepLog::getThingsYouCanLiftChoices() as $label => $id) {
28             $repLogAppProps['itemOptions'][] = [
29                 'id' => $id,
30                 'text' => $translator->trans($label),
31             ];
32         }
... lines 33 - 37
38     }
... lines 39 - 66
```

Cool! This code builds the structure we need: it has an `itemOptions` key, we loop over each, and create the `id` and `text` keys. Now when we refresh, Yep! The dumped code looks *exactly* like our `REP_LOG_APP_PROPS` JavaScript structure! Heck, we can add `withHeart => true`... because I like the heart.

```
67 lines | src/Controller/LiftController.php
... lines 1 - 19
20     public function indexAction(Request $request, RepLogRepository $replLogRepo, UserRepository $userRepo, TranslatorInterface $translator)
21     {
... lines 22 - 23
24         $repLogAppProps = [
25             'withHeart' => true,
... line 26
27         ];
... lines 28 - 38
39     }
... lines 40 - 67
```

Remove the `die` and pass this into twig as a new `repLogAppProps` variable.

```
67 lines | src/Controller/LiftController.php
... lines 1 - 34
35     return $this->render('lift/index.html.twig', array(
... line 36
37         'repLogAppProps' => $repLogAppProps,
38     ));
... lines 39 - 67
```

Ready for the last piece? Delete the old JavaScript object and replace it with: `{{ repLogAppProps|json_encode|raw }}`.



```

73 lines | templates/lift/index.html.twig
... lines 1 - 63
64 {% block javascripts %}
... lines 65 - 66
67 <script>
68     window.REP_LOG_APP_PROPS = {{ repLogAppProps|json_encode|raw }};
69 </script>
... lines 70 - 71
72 {% endblock %}

```

That will print that array as JSON... which of course, is the same as JavaScript.

Ah, do you love it? It's now *very* easy to pass dynamic values or initial state into your app. Try it: refresh!

## Removing the Old Code!

And... woh! Our app is now basically working! Yea, we're going to look at a few more things, but I think it's time to *delete* our old code and put this app into the right spot. In other words, it's time to celebrate!

Start by deleting this entire old Components directory: all of that code was used by the old app. Delete the old entry file - `rep_log.js` and inside of the template, we can remove a *ton* of old markup. The new lift-stuff-app div *now* lives *right* next to the leaderboard.

```

36 lines | templates/lift/index.html.twig
... lines 1 - 2
3 {% block body %}
4     <div class="row">
5         <div id="lift-stuff-app"></div>
6
7         <div class="col-md-5">
8             <div class="leaderboard">
... lines 9 - 16
17         </div>
18     </div>
19 </div>
20 {% endblock %}
... lines 21 - 36

```

Oh, and delete `_form.html.twig` too - more old markup. At the bottom, remove the original script tag.

And in `webpack.config.js`, delete the old entry. Wow! Webpack is angry because of the missing entry file. Stop and restart it:

```

$ yarn run encore dev-server

```

It builds! Go back and refresh! It's alive! And it works! Except... it's jumpy when it loads. The leaderboard starts on the left, then moves over once our app renders.

This is caused by a *mistake* I made. Look inside `RepLogs`. This is our main presentation component: it gives us all the markup. And, it has a `col-md-7` class on it. Now, it's not wrong to put grid classes like this inside React. But, this top-level grid class *is* a bit weird: if we tried to use this component in a different place in our site, it would *always* have that `col-md-7`. It makes more sense to *remove* that class and, instead, in `index.html.twig`, add the class there. Now, our React app will just fit inside this.

36 lines | [templates/lift/index.html.twig](#)

... lines 1 - 2

3 {% block body %}

4 <div class="row">

5 <div id="lift-stuff-app" class="col-md-7"></div>

... lines 6 - 18

19 </div>

20 {% endblock %}

... lines 21 - 36

And when you reload the page, yes: no annoying jumping!

Next: we *know* that React can be used to create re-usable components... but we haven't really done this yet. Time to change that!

# Chapter 41: Reusable Components

React's entire architecture is centered around making components that are reusable. This is *especially* easy to see with the dumb, presentational components: all they do is receive props and... render! It would be *very* easy to render those components with *different* props in different places.

But, in reality, a lot of components aren't really meant to be reused: RepLogCreator, RepLogList and RepLogs... yea, it's *pretty* unlikely we'll use those on other parts of our site... except maybe for RepLogCreator, which could be used to *edit* rep logs.

But, there *are* a lot of nice use-cases for building truly, re-usable components, basically, *tools*. For example, it's pretty simple, but, suppose we had a lot of Bootstrap submit buttons and we want to turn the submit button into its own React component. We can *totally* do that, and it's pretty awesome.

In the assets/js directory, create a new directory called components/ and inside, a new file called Button.js. I'm not putting this in RepLog because this could be used on other parts of the site.

This will be a dumb, presentation component. So, copy the import lines from the top of RepLogCreator, and then say export default function Button with the normal props argument. Inside, return the markup for a button with className="btn", because every button at *least* has that class.

```
16 lines | assets/js/Components/Button.js
1  import React from 'react';
2  import PropTypes from 'prop-types';
3
4  export default function Button(props) {
5    return (
6      <button
7        className="btn"
8      >
9    );
10 }
11
... lines 12 - 16
```

## Spreading the Props

Go back to RepLogCreator and scroll down. Ok, this button has type="submit". We *could* add that to our Button component, but not *all* buttons need this. But, no worries: we can *allow* this to be passed in as a prop! In fact, we might need to pass a *bunch* of different attributes as props.

To allow that, use the attribute spread operator ...props. It's simple: any prop passed to this component will be rendered as an attribute. And for the text, hmm: how about a prop called text: props.text. Close the button tag. At the bottom, add Button.propTypes = and define text as a string that's required.

```

16 lines | assets/js/Components/Button.js
... lines 1 - 3
4   export default function Button(props) {
5     return (
6       <button
... line 7
8         {...props}
9       >{props.text}</button>
10    );
11  }
... line 12
13  Button.propTypes = {
14    text: PropTypes.string.isRequired
15  };

```

Perfect!

Back in RepLogCreator, head up top and bring this in: import Button from ./Components/Button.

```

100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 2
3   import Button from './Components/Button';
... lines 4 - 100

```

Then *all* the way down at the bottom, use `<Button type="submit" />` and also the text prop. Copy the original text and paste it here.

```

100 lines | assets/js/RepLog/RepLogCreator.js
... lines 1 - 46
47  render() {
... lines 48 - 50
51    return (
... lines 52 - 88
89      <Button type="submit" text="I Lifted it!" />
... line 90
91    );
92  }
... lines 93 - 100

```

We *are* going to temporarily lose the btn-primary class. That *is* a problem, and we'll fix it soon. Delete the old button. This should work! Move over and refresh! There it is! The button has the class, type="submit" and the text. Hmm, but it also has a text= attribute... which makes perfect sense: we added that as a prop! Of course, we don't *actually* want that attribute, so we'll need to fix that.

## Using props.children

But first, we have a *bigger* problem! What if I wanted to add a Font Awesome icon to the text? Normally we would add a `<span className="">` and then the classes. But... this doesn't look right: I'm putting HTML inside of this string. And, actually, this wouldn't even *work*, because React escapes HTML tags in strings.

New idea: what if we could remove this text prop and treat the Button like a *true* HTML element by putting the text *inside*. That looks *awesome*. This is not only possible, this is *idea*! By doing it this way, we can include text, HTML elements or even other React components! Woh!

If you pass something in the body of a Component, that is known as its *children*, and you can access it automatically with `props.children`. It's that simple.

12 lines | [assets/js/Components/Button.js](#)

... lines 1 - 4

```
5     return (  
6         <button  
... lines 7 - 8  
9         >{props.children}</button>  
10    );  
... lines 11 - 12
```

Oh, and ESLint is angry because we're missing props validation for children. I'm going to ignore that because the children prop is a special case and, we don't really care of its text, a component or something else. But, you could add it with the PropTypes "any" type.

Remove the propTypes for now. Let's try it! Move over and refresh! Looking good. To prove that using children is awesome, add a new `<span>` with `className="fa fa-plus-circle"`.

102 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 88

```
89     <Button type="submit">  
90         I Lifted it <span className="fa fa-plus-circle"></span>  
91     </Button>  
... lines 92 - 102
```

Go check that out. Nice!

## Merging Prop Values in your Component

Ok, we're still missing the btn-primary class. This is a bit trickier. We can't just pass a className prop here, because, in the Button, we're *already* passing a className prop. So, let's just be smarter! Enter JavaScript, add ticks, then print btn, then `${props.className}`.

19 lines | [assets/js/Components/Button.js](#)

... lines 1 - 4

```
5     return (  
6         <button  
7             className={`btn ${props.className}`}  
... lines 8 - 9  
10    );  
... lines 11 - 19
```

That should do it! We're not passing this prop yet, but try it: refresh. Oh man, undefined! Of course! Let's go clean things up.

First, add `Button.propTypes` to advertise that we accept a className prop that's a string. We *could* make this required... *or* we can allow it to be optional, but fix that undefined problem. To do that, set `Button.defaultProps` to an object with className set to empty quotes.

19 lines | [assets/js/Components/Button.js](#)

... lines 1 - 11

```
12 Button.propTypes = {  
13     className: PropTypes.string  
14 };  
15  
16 Button.defaultProps = {  
17     className: ""  
18 };
```

Problem solved! Try it again. Wait! What? Now the class attribute is empty? How is that even possible? To see why, go back to `RepLogCreator` and pass a className prop here: `btn-primary`.

102 lines | [assets/js/RepLog/RepLogCreator.js](#)

... lines 1 - 88

```
89      <Button type="submit" className="btn-primary">
```

... lines 90 - 102

Go refresh again. Huh, now it has *that* class... but not the btn class. Here's the deal: sure, we have this className prop here. But, thanks to the ...props, the className prop we're passing *in* overrides the first one! We *could* move the ...props first, but, in general, we *do* want to allow whoever uses this component to override its props.

So, hmm: we basically want to print *all* of the props here... *except* for className. We *can* do that, and it's cool! Up top, let's destructure: const {} = props, then get out className. Use that below.

*Then* - this is the cool part - destructure a second variable, ...otherProps. Use *that* below in the button.

21 lines | [assets/js/Components/Button.js](#)

... lines 1 - 4

```
5    const { className, ...otherProps } = props;
```

... line 6

```
7    return (
```

```
8      <button
```

```
9        className={`btn ${className}`}
```

... lines 10 - 11

```
12    );
```

... lines 13 - 21

Yep, the ...otherProps will be *all* of the props, *except* for any that we *specifically* destructured before it. It's an *awesome* little trick.

Ok, try it out: move over, refresh and... we got it! It looks perfect! I hope this tiny component gets you excited about the possibilities of reusing code with React.

# Chapter 42: CSRF Protection Part 1

We've *gotta* talk about one more thing: security. Specifically, CSRF attacks.

## CSRF Attack?

Imagine if a malicious person built an HTML form on a totally different site, but set its `action=""` attribute to a URL on *our* site. Then, what if some user, like me, who is logged into our site, was tricked into submitting that form? Well, the form would submit, I would of course be authenticated, and the request would be successful! That's a problem! The malicious user was basically able to make a request to our site logged in as me! They could have done anything!

The other possible attack vector is if a malicious user runs JavaScript on their site that makes an AJAX call to our site. The result is exactly the same.

## Do APIs Need Protection?

So, how can we protect against this in an API? The answer... you might not need to. If you follow two rules, then CSRF attacks are not possible.

### Tip

Update: A more secure option is now available: to use SameSite cookies, which are now supported by most browsers and can be enabled in Symfony: <https://symfony.com/blog/new-in-symfony-4-2-samesite-cookie-configuration>. If you need to support older browsers, using CSRF tokens is best.

First, disallow AJAX requests from all domains except for your domain. Actually, this is just how the Internet works: you can't make AJAX requests across domains. *If you do* need to allow other domains to make AJAX requests to your domain, you do that by setting CORS headers. If you're in this situation, just make sure to only allow specific domains you trust, not everyone. This first rule prevents bad AJAX calls.

For the second rule, look at our API: `src/Controller/RepLogController`. Find `newRepLogAction()`. Notice that the body of the request is JSON. This is the second rule for CSRF protection: *only* allow data to be sent to your server as JSON. This protects us from, for example, bad forms that submit to our site. Forms cannot submit their data as JSON.

If you follow these two rules - which you probably do - then you do not need to worry about CSRF. But, to be *fully* sure, we *are* going to add one more layer: we're going to force all requests to our API to have a Content-Type header set to `application/json`. By requiring that, there is *no* way for a bad request to be made to our site, unless we're allowing it with our CORS headers.

Oh, and important side note: CSRF attacks only affect you if you allow session-based authentication like we're doing, or HTTP basic authentication. If you require an API token, you're also good!

## Creating the Event Subscriber

We're going to require the Content-Type header by creating an event subscriber, so that we don't need to add this code in every controller. First, to speed things up, install MakerBundle:

```
$ composer require maker --dev
```

When that finishes, run:

```
$ php bin/console make:subscriber
```

Call it `ApiCsrfValidationSubscriber`. And, listen to the `kernel.request` event. Done! This made one change: it created a new class in `src/EventSubscriber`.

22 lines | [src/EventSubscriber/ApiCsrfValidationSubscriber.php](#)

... lines 1 - 7

```
8 class ApiCsrfValidationSubscriber implements EventSubscriberInterface
9 {
10     public function onKernelRequest(GetResponseEvent $event)
11     {
12         // ...
13     }
14
15     public static function getSubscribedEvents()
16     {
17         return [
18             'kernel.request' => 'onKernelRequest',
19         ];
20     }
21 }
```

Awesome! Because we're listening to `kernel.request`, the `onKernelRequest()` method will be called on every request, *before* the controller. At the top of the method, first say if `!$event->isMasterRequest()`, then return. That's an internal detail to make sure we only run this code for a real request.

31 lines | [src/EventSubscriber/ApiCsrfValidationSubscriber.php](#)

... lines 1 - 9

```
10     public function onKernelRequest(GetResponseEvent $event)
11     {
12         if (!$event->isMasterRequest()) {
13             return;
14         }
15
16         ... lines 15 - 21
17
18     }
19
20     ... lines 23 - 31
```

Next, we do not need to require the Content-Type header for safe HTTP methods, like GET or HEAD, because, unless we do something awful in our code, these requests don't *change* anything on the server. Add `$request = $event->getRequest()`. Then, if `$request->isMethodSafe(false)`, just return again.

31 lines | [src/EventSubscriber/ApiCsrfValidationSubscriber.php](#)

... lines 1 - 15

```
16     $request = $event->getRequest();
17
18     // no validation needed on safe methods
19     if ($request->isMethodSafe(false)) {
20         return;
21     }
22
23     ... lines 22 - 31
```

The false part isn't important: that's a flag for a backwards-compatibility layer.

Perfect! Next, we need to determine whether or not this request is to our *api*. We'll do that with a cool annotation trick. Then, we'll make sure the Content-Type header is set to `application/json`.



# Chapter 43: CSRF Protection Part 2

The *only* tricky thing is that we *only* want to require the Content-Type header when the user is requesting an API endpoint. In our application, this means all the endpoints inside of RepLogController. So, we could see if the URL starts with /reps... but that could get ugly later if the API grows to a lot of *other* URLs.

If your app is *entirely* an API, that's easy! Or if all the URLs start with /api, that's also easy to check.

But, in our app, let's use a different trick... which is gonna be kinda fun.

Above the controller class, add @Route() with defaults={} and a new flag that I'm inventing: `_is_api` set to true.

```
101 lines | src/Controller/RepLogController.php
... lines 1 - 13
14  /**
... line 15
16  * @Route(defaults={"_is_api": true})
17  */
18  class RepLogController extends BaseController
... lines 19 - 101
```

When you put an @Route annotation above the controller class, it means its config will be applied to all of the routes below it. Now, inside of the subscriber, we can read this config. To see how, add `dump($request->attributes->all())` then die.

```
32 lines | src/EventSubscriber/ApiCsrfValidationSubscriber.php
... lines 1 - 9
10  public function onKernelRequest(GetResponseEvent $event)
11  {
... lines 12 - 16
17      dump($request->attributes->all());die;
... lines 18 - 22
23  }
... lines 24 - 32
```

If you refresh the main page... no `_is_api` here. But now go to /reps. There it is! Any defaults flags that we set are available in `$request->attributes`.

## Creating a Custom ApiRoute Annotation

The only problem is that this syntax is... oof... gross. Let's make it easier. In the Api directory, create a new PHP class called ApiRoute. Make this extend the normal Route annotation class.

Yep, we're creating a brand new, customized route annotation. Add @Annotation above the class.

```
12 lines | src/Api/ApiRoute.php
... lines 1 - 6
7  /**
8  * @Annotation
9  */
10  class ApiRoute extends Route
11  {
12  }
```

If we did nothing else, we could at least go into our controller and use it: `@ApiRoute()`.

```

102 lines | src/Controller/RepLogController.php
... lines 1 - 14
15  /**
... line 16
17  * @ApiResponse(defaults={"_is_api": true})
18  */
19  class RepLogController extends BaseController
... lines 20 - 102

```

Try it! Nothing changes. But *now*, in ApiResponse, go to the Code -> Generate menu - or Command+N on a Mac - and override the getDefaults() method. Return a merge of \_is\_api set to true and parent::getDefaults().

```

20 lines | src/Api/ApiResponse.php
... lines 1 - 11
12  public function getDefaults()
13  {
14      return array_merge(
15          ['_is_api' => true],
16          parent::getDefaults()
17      );
18  }
... lines 19 - 20

```

Nice, right? Back in the controller, remove the ugly defaults stuff. Oh, and if you want to mark just *one* route as an API route, you can also use this new annotation above just one method.

Ok, go back and refresh! Got it!

### [Validating the Content-Type Header](#)

Back in the subscriber, remove the dump. Then, if !\$request->attributes->get('\_is\_api') return. And now that we know we were only operating on API requests, check the header: if \$request->headers->get('Content-Type') does not equal application/json, we have a problem! Create a new 400 response: \$response = new JsonResponse().

The data we send back doesn't matter - I'll add a message that says what went wrong. But, give this a 415 status code: this means "Unsupported Media Type". Finish this with \$event->setResponse(\$response). This will completely stop the request: this response will be returned *without* even calling your controller.

46 lines | [src/EventSubscriber/ApiCsrfValidationSubscriber.php](#)

```
... lines 1 - 10
11     public function onKernelRequest(GetResponseEvent $event)
12     {
... lines 13 - 23
24         if (!$request->attributes->get('_is_api')) {
25             return;
26         }
27
28         if ($request->headers->get('Content-Type') !== 'application/json') {
29             $response = new JsonResponse([
30                 'message' => 'Invalid Content-Type'
31             ], 415);
32
33             $event->setResponse($response);
34
35             return;
36         }
37     }
... lines 38 - 46
```

Ok, let's try this! Find the `rep_log_api.js` file and look down at `createRepLog`. We *are* setting this Content-Type header. So, this should work! Move over, go back to `/lift` and refresh. I'll open my network tools. And.. yea! It totally works! But try to delete a rep log... failure! With a 415 status code.

### Always Sending the Content-Type Header

This is because the DELETE endpoint does *not* set this header. And... hmm, it's kinda weird... because, for the DELETE endpoint, the body of the request is empty. There's some debate, but, because of this, some people would argue that this request should not need *any* Content-Type header... because we're not *really* sending any JSON!

But, by requiring this header to *always* be set, we give our application a bit more security: it removes the possibility that's somebody could create a CSRF attack on that endpoint... or some future endpoint that we don't send any data to.

In other words, we are *always* going to set this header. Remove it from `createRepLog` and go up to `fetchJson()` so we can set this here. The only tricky thing is that it's *possible* that someone who calls this will pass a custom header, and we don't want to override that.

Add `let headers =` and set this to the Content-Type header. Then, if `options && options.headers` - so, if the user passes a custom header, merge them together: `headers = , ...options.headers` then `...headers`. Then, delete that property and, below, pass `headers` to `headers`.

53 lines | [assets/js/api/rep\\_log\\_api.js](#)

```
1  function fetchJson(url, options) {
2      let headers = {'Content-Type': 'application/json'};
3      if (options && options.headers) {
4          headers = {...options.headers, ...headers};
5          delete options.headers;
6      }
7
8      return fetch(url, Object.assign({
... line 9
10          headers: headers,
11      }, options))
... lines 12 - 17
18  }
... lines 19 - 53
```

Try it! Move over - looks like the page already refreshed. And... yes! We can delete again!

And we are protected from CSRF! That's because, first, we do *not* allow other domains to make AJAX calls to our site and, second, all of our API endpoints require a JSON body - which we *explicitly* required by looking for the Content-Type header.

Oh my gosh.... we're done! That's it, that's everything! If you've made it all the way through, you rock! You have the tools to create the craziest frontend you can think of! And yes, there *are* more things in React that we could cover, like the React router or Redux, which adds a more complex architecture on top of React, but helps solve the problem of passing around so many props.

But, these are extras - go get some *real-world* success with React and report back! We'd love to know what you're building.

Alright people, seeya next time.

