JavaScript for PHP Geeks

With <3 from KnpUniversity

Chapter 1: Lift Stuff! The js- Prefix

Guys, get ready to pump up... on your JavaScript skills! No, no, I'm not talking about the basics. Look, I get it: you know how to write JavaScript, you're a ninja and a rock star all at once with jQuery. That's awesome! In fact, it's exactly where I want to start. Because in this tutorial, we're going to flex our muscles and start asking questions about *how* things - that we've used for years - actually work.

And this will make us more dangerous right away. But, but but! It's also going to lead us to our real goal: building a foundation so we can learn about *ridiculously* cool things in future tutorials, like module loaders and front-end frameworks like ReactJS. Yep, in a few short courses, we're going to take a traditional HTML website and transform it into a modern, hipster, JavaScript-driven front-end. So buckle up.

The Project: Pump Up!

As always, please, please, please, do the heavy-lifting and code along with me. By the way, in 30 seconds, I promise you'll understand why I'm making all these amazing weight-lifting puns. I know, you just can't... weight.

Anyways, download the course code from any page and unzip it to find a `start/` directory. That will have the same code that you see here. Follow the details in the `README.md` file to get your project set up.

The last step will be to open a terminal, move into your project and do 50 pushups. I mean, run:

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

to start the built-in PHP web server. Now, this *is* a Symfony project but we're not going to talk a lot about Symfony: we'll focus on JavaScript. Pull up the site by going to `http://localhost:8000`.

Welcome... to Lift Stuff: an application for programmers, like us, who spend all of their time on a computer. With Lift Stuff, they can stay in shape and record the things that they lift while working.

Let me show you: login as `ron_furgandy`, password `pumpup`. This is the only important page on the site. On the left, we have a history of the things that we've lifted, like our cat. We can lift many different things, like a fat cat, our laptop, or our coffee cup. Let's get in shape and lift our coffee cup 10 times. I lifted it! Our progress is saved, and we're even moving up the super-retro leaderboard on the right! I'm coming for you Meowly Cyrus!

Setting up the Delete Link

But, from a JavaScript standpoint, this is all *incredibly* boring, I mean traditional! Our first job - in case I fall over my keyboard while eating a donut and mess up - is to add a delete icon to each row. When we click that, it should send an AJAX request to delete that from the database, remove the row entirely from the page, and update the total at the bottom.

Right now, this entire page is rendered on the server, and the template lives at `app/Resources/views/lift/index.html.twig`:

60 lines | app/Resources/views/lift/index.html.twig

```
{% extends 'base.html.twig' %}

{% block body %}
    <div class="row">
        <div class="col-md-7">
            <h2>
                Lift History
                <a href="#list-stuff-form" class="btn btn-md btn-success pull-right">
                    <span class="fa fa-plus"></span> Add
                </a>
            </h2>

            <table class="table table-striped">
                <thead>
                <tr>
                    <th>What</th>
                    <th>How many times?</th>
                    <th>Weight</th>
                    <th> </th>
                </tr>
                </thead>
                <tbody>
                {% for repLog in repLogs %}
                    <tr>
                        <td>{{ repLog.itemLabel|trans }}</td>
                        <td>{{ repLog.reps }}</td>
                        <td>{{ repLog.totalWeightLifted }}</td>
                        <td>
                             
                        </td>
                    </tr>
                {% else %}
                    <tr>
```

```
                <td colspan="4">Get liftin'!</td>
            </tr>
        {% endfor %}
        </tbody>
        <tfoot>
            <tr>
                <td> </td>
                <th>Total</th>
                <th>{{ totalWeight }}</th>
                <td> </td>
            </tr>
        </tfoot>
    </table>

    {{ include('lift/_form.html.twig') }}

  </div>
  <div class="col-md-5">
    <div class="leaderboard">
        <h2 class="text-center"><img class="dumbbell" src="{{ asset('assets/images/dum

        {{ include('lift/_leaderboard.html.twig') }}
    </div>
  </div>
</div>
{% endblock %}
```

Inside, we're looping over something I call a `repLog` to build the table:

60 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr>
                        <td>{{ repLog.itemLabel|trans }}</td>
                        <td>{{ repLog.reps }}</td>
                        <td>{{ repLog.totalWeightLifted }}</td>
                        <td>
                             
                        </td>
                    </tr>
... lines 32 - 35
                {% endfor %}
... lines 37 - 45
            </table>
... lines 47 - 49
        </div>
... lines 51 - 57
    </div>
{% endblock %}
```

Each `repLog` represents one item we've lifted, and it's the only important table in the database. It has an `id`, the *number* of reps that we lifted and the total weight:

200 lines | src/AppBundle/Entity/RepLog.php

```
... lines 1 - 8
/**
 * RepLog
 *
 * @ORM\Table(name="rep_log")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\RepLogRepository")
 */
class RepLog
{
... lines 17 - 27
    /**
     * @var integer
     *
     * @Serializer\Groups({"Default"})
     * @ORM\Column(name="id", type="integer")
```

```
33      * @ORM\Id
34      * @ORM\GeneratedValue(strategy="AUTO")
35      */
36     private $id;
37
38     /**
39      * @var integer
40      *
41      * @Serializer\Groups({"Default"})
42      * @ORM\Column(name="reps", type="integer")
43      * @Assert\NotBlank(message="How many times did you lift this?")
44      * @Assert\GreaterThan(value=0, message="You can certainly life more than just 0!")
45      */
46     private $reps;
47
48     /**
49      * @var string
50      *
51      * @Serializer\Groups({"Default"})
52      * @ORM\Column(name="item", type="string", length=50)
53      * @Assert\NotBlank(message="What did you lift?")
54      */
55     private $item;
56
57     /**
58      * @var float
59      *
60      * @Serializer\Groups({"Default"})
61      * @ORM\Column(name="totalWeightLifted", type="float")
62      */
63     private $totalWeightLifted;
↕   ... lines 64 - 198
199 }
```

Adding the Delete link and js- class

To add the delete link, inside the last `<td>` add a new anchor tag. Set the `href` to `#`, since we plan to let JavaScript do the work. And then, give it a class: `js-delete-rep-log`:

72 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr>
... lines 25 - 27
                        <td>
                            <a href="#" class="js-delete-rep-log">
... line 30
                            </a>
                        </td>
                    </tr>
... lines 34 - 37
                {% endfor %}
... lines 39 - 47
            </table>
... lines 49 - 51
        </div>
... lines 53 - 59
    </div>
{% endblock %}
... lines 62 - 72
```

Inside, add our cute little delete icon:

72 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr>
... lines 25 - 27
                        <td>
                            <a href="#" class="js-delete-rep-log">
                                <span class="fa fa-trash"></span>
                            </a>
                        </td>
                    </tr>
... lines 34 - 37
                {% endfor %}
... lines 39 - 47
            </table>
... lines 49 - 51
        </div>
... lines 53 - 59
    </div>
{% endblock %}
... lines 62 - 72
```

Adorable! Ok, first! Why did we add this `js-delete-rep-log` class? Well, there are only ever two reasons to add a class: to style that element, or because you want to find it in JavaScript.

Our goal is the second, and by prefixing the class with `js-`, it makes that *crystal* clear. This is a fairly popular standard: when you add a class for JavaScript, give it a `js-` prefix so that future you doesn't need to wonder which classes are for styling and which are for JavaScript. Future you will... thank you.

Copy that class and head to the bottom of the template. Add a block `javascripts`, `endblock` and call the `parent()` function:

72 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
    {{ parent() }}
... lines 65 - 70
{% endblock %}
```

This is Symfony's way of adding JavaScript to a page. Inside, add a `<script>` tag and

then, use jQuery to find all `.js-delete-rep-log` elements, and then `.on('click')`, call this function. For now, just `console.log('todo delete!')`:

72 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
    {{ parent() }}

    <script>
        $('.js-delete-rep-log').on('click', function() {
            console.log('todo delete!');
        });
    </script>
{% endblock %}
```

Resolving External JS in PHPStorm

But hmm, PhpStorm says that `$` is an unresolved function or method. Come on! I *do* have jQuery on the page. Open the base layout file - `base.html.twig` - and scroll to the bottom:

97 lines | app/Resources/views/base.html.twig

```
<!DOCTYPE html>
<html lang="en">
... lines 3 - 19
<body>
... lines 21 - 90
{% block javascripts %}
    <script src="https://code.jquery.com/jquery-3.1.1.min.js" integrity="sha256-hVVnYaiADRTO
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js" integrity=
{% endblock %}
</body>
</html>
```

Both jQuery and Bootstrap should be coming in from a CDN. Oh, but this note says that there is no locally stored library for the http link. Aha! Tell PhpStorm to download and learn all about the library by pressing `Option` + `Enter` on a Mac - or `Alt` + `Enter` on Linux or Windows - and choosing "Download Library". Do the same thing for Bootstrap.

Et voilà! The error is gone, and we'll start getting at least *some* auto-completion.

Using .on() versus .click()

Oh, and I want you to notice one other thing: we're using `.on('click')` instead of the `.click()` function. Why? Well, they both do the same thing. But, there are an *infinite* number of events you could listen to on any element: click, change, keyup, mouseover

or even custom, invented events. By using `.on()`, we have one consistent way to add a listener to *any* event.

It's a small start, but already when we refresh, open the console, and click delete, it works! Now, let's follow the rabbit hole deeper.

Chapter 2: (document).ready() & Ordering

When we use this `javascripts` block thing:

72 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
    {{ parent() }}

    <script>
        $('.js-delete-rep-log').on('click', function() {
            console.log('todo delete!');
        });
    </script>
{% endblock %}

```

It adds our new JavaScript code right *after* the main script tags in the base layout:

97 lines | app/Resources/views/base.html.twig

```
<!DOCTYPE html>
<html lang="en">
... lines 3 - 19
<body>
... lines 21 - 90
{% block javascripts %}
    <script src="https://code.jquery.com/jquery-3.1.1.min.js" integrity="sha256-hVVnYaiADRTO
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js" integrity=
{% endblock %}
</body>
</html>
```

View the HTML source and scroll to the bottom see that in action. Yep, jQuery and *then* our stuff.

Our JavaScript lives at the bottom of the page for a reason: performance. Unless you add an `async` attribute, when your browser sees a `script` tag, it stops, waits while that file is downloaded, executes it, and *then* continues.

But not everyone agrees that putting JS in the footer is the best thing since Chuck Norris. After all, if your page is *heavily* dependent on JS, your user might see a blank page for a second before your JavaScript has the chance to execute and put cool stuff

there, like a photo of Chuck Norris.

So, there might be some performance differences between putting JavaScript in the header versus the footer. But, our code should work equally well in either place, right? If I move the block `javascripts` up into my header, this should *probably* still work?

98 lines | app/Resources/views/base.html.twig

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
   ... lines 4 - 16
17     {% block javascripts %}
18         <script src="https://code.jquery.com/jquery-3.1.1.min.js" integrity="sha256-hVVnYaiADRT
19         <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js" integri
20     {% endblock %}
   ... lines 21 - 22
23 </head>
   ... lines 24 - 96
97 </html>
```

We still have 3 script tags, in the same order, just in a different spot.

Well... let's find out! Refresh! Then click delete. Ah we broke it! What happened?!

Running JavaScript Before the DOM

This may or may not be obvious to you, but it's worth mentioning: our browser executes JavaScript as soon as it sees it... which might be before some or all of the page has actually loaded. Our code is looking for all elements with the `js-delete-rep-log` class. Well, at this point, *none* of the HTML body has loaded yet, so it finds exactly zero elements.

This is the reason why you probably already always use the famous `$(document).ready()` block. Move our code inside of it, and refresh again:

74 lines | app/Resources/views/lift/index.html.twig

```
   ... lines 1 - 62
63 {% block javascripts %}
   ... lines 64 - 65
66     <script>
67         $(document).ready(function() {
68             $('.js-delete-rep-log').on('click', function () {
69                 console.log('todo delete!');
70             });
71         });
72     </script>
73 {% endblock %}
```

Yes!

Very simply, jQuery calls your `$(document).ready()` function once the DOM has fully loaded. But it's nothing fancy: it's approximately equal to putting your JavaScript code at the absolute bottom of the page. It's nice because it makes our code portable: it will work no matter *where* it lives.

We could even take the `script` tag, delete it from the block, and put it *right* in the middle of the page:

74 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 47
            </table>

            <script>
                $(document).ready(function() {
                    $('.js-delete-rep-log').on('click', function () {
                        console.log('todo delete!');
                    });
                });
            </script>

            {{ include('lift/_form.html.twig') }}
        </div>
... lines 60 - 66
    </div>
{% endblock %}

{% block javascripts %}
    {{ parent() }}

{% endblock %}
```

Now in the HTML, the external `script` tags are still on top, but our `JavaScript` lives right, smack in the middle of the page. And when we refresh, it still works super well.

Thinking out JavaScript Ordering

Of course, the *only* problem is if someone comes along and decides:

> Hey, you know what? We should really put our JavaScript in the footer! Chuck

> Norris told me it's better for performance.

98 lines | app/Resources/views/base.html.twig

```
1  <!DOCTYPE html>
2  <html lang="en">
   ... lines 3 - 19
20 <body>
   ... lines 21 - 90
91 {% block javascripts %}
92     <script src="https://code.jquery.com/jquery-3.1.1.min.js" integrity="sha256-hVVnYaiADRTO
93     <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js" integrity=
94 {% endblock %}
95
96 </body>
97 </html>
```

Now, we have a different problem. In the source, jQuery once again lives at the absolute bottom. But when we refresh the page, error! Our browser immediately tells us that `$` is not defined.

This comes from *our* code, which still lives in the middle of the page. And yea, it makes sense: as our browser loads the page, it sees the `$`, but has *not* yet downloaded `jQuery`: that script tag lives further down.

So there are *two* things we need to worry about. First, any JavaScript that I depend on needs to be included on the page before me. And actually, this will *stop* being true when we talk about module loaders in a future tutorial.

Second, before I try to select any elements with jQuery, I better make sure the DOM has loaded, which we can always guarantee with a `$(document).ready()` block.

Let's put our JavaScript back into the block so that it's always included *after* jQuery, whether that's in the header of footer:

73 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
    {{ parent() }}

    <script>
        $(document).ready(function() {
            $('.js-delete-rep-log').on('click', function () {
                console.log('todo delete!');
            });
        });
    </script>
{% endblock %}
```

Go back, refresh, and life is good again.

Next, let's talk about bubbles! I mean, event bubbling!

Chapter 3: All about Event Bubbling

I'm feeling so good about our first click listener, let's add another! When I click *anywhere* on a row, I also want to log a message.

Back in the template, give the entire table a `js` class so we can select it. How about `js-rep-log-table`:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 47
            </table>
... lines 49 - 50
        </div>
... lines 52 - 58
    </div>
{% endblock %}
... lines 61 - 77
```

Down below, find that and look inside for the `tbody tr` elements. Then, `.on('click')` add a function that prints some fascinating text: `console.log('row clicked')`:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
    {{ parent() }}

    <script>
        $(document).ready(function() {
... lines 67 - 70
            $('.js-rep-log-table tbody tr').on('click', function() {
                console.log('row clicked!');
            });
        });
    </script>
{% endblock %}
```

Beautiful! Refresh and click the row. No surprises: we see "row clicked". But check this

out: click the delete link. Hot diggity - *two* log messages! Of course it would do this! I clicked the delete link, but the delete link is inside of the row. Both things got clicked!

All about Event Bubbling

Welcome to *event bubbling*, an important concept in JavaScript that's *just* boring enough that you've probably avoided reading articles about it in the past. Let's make it awesome.

Here it goes: when we click, we cause a `click` event. Now technically, when I click the delete icon, the element that I'm *actually* clicking is the *span* that holds the icon. Cool! So, your browser goes to that `span` element and says:

> Top of the morning! I'd like to trigger a *click* event on you!

Then, if there are any listener functions attached on `click`, those are called. Next, your browser goes up one level to the anchor and says:

> Ahoy Matey! I'd like to trigger a *click* event on you!

And the same thing happens again: if there are any `click` listener functions attached to *that* element, those are executed. This includes *our* listener function. From here, it just keeps going: *bubbling* all the way up the tree: to the `td`, the `tr`, `tbody`, `table`, and eventually, to the `<body>` tag itself.

And *that* is why we see "todo delete" first: the event bubbling process notifies the link element and *then* bubles up and notifies the `tr`.

Prefixing $variables with $

Cool! Let's play with this! First, let's clean up our code a bit and make a minor performance improvement. Add `var $table = $('.js-rep-log-table')`. Then below, instead of searching the entire page for these delete links, use `$table.find()` to *only* look inside that table:

79 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
    {{ parent() }}

    <script>
        $(document).ready(function() {
            var $table = $('.js-rep-log-table');

            $table.find('.js-delete-rep-log').on('click', function () {
                console.log('todo delete!');
            });
... lines 72 - 75
        });
    </script>
{% endblock %}
```

Do the same below: `$table.find()` and look for the `tbody tr` elements in that:

```
79 lines | app/Resources/views/lift/index.html.twig
... lines 1 - 61
{% block javascripts %}
    {{ parent() }}

    <script>
        $(document).ready(function() {
            var $table = $('.js-rep-log-table');
... lines 68 - 72
            $table.find('tbody tr').on('click', function() {
                console.log('row clicked!');
            });
        });
    </script>
{% endblock %}
```

If you refresh now, it still works great. But some of you might be wondering about my variable name: `$table`? For PHP developers, that looks weird... because, ya know, `$` means something important in PHP. But in JavaScript, `$` is *not* a special character. In fact, it's *so* not special that - if you want - you can even start a variable name with it. Madness! So the `$` in `$table` isn't doing anything special, but it *is* a fairly common convention to denote a variable that is a `jQuery` object.

It's nice because when I see `$table`, I think:

> Oh! This starts with a `$`! Good show! I bet it's a jQuery object, and I can call find() or any other fancy jQuery method on it. Jolly good!

Now that we understand event bubbling, let's mess with it! Yes, we can actually *stop* the bubbling process... which is probably *not* something you want to do... but you might already be doing it accidentally.

Chapter 4: The Event Argument & stopPropagation

Back to our mission: when I click a delete link, it works... but I *hate* that it puts that annoying `#` in my URL and scrolls me up to the top of the page. You guys have probably seen and fixed that a million times. The easiest way is by finding your listener function and - at the bottom - returning `false`:

81 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62 {% block javascripts %}
... lines 63 - 64
65     <script>
66         $(document).ready(function() {
67             var $table = $('.js-rep-log-table');
68
69             $table.find('.js-delete-rep-log').on('click', function () {
70                 console.log('todo delete!');
71
72                 return false;
73             });
... lines 74 - 77
78         });
79     </script>
80 {% endblock %}
```

Go back, remove that pound sign, refresh, and click! Haha! Get outta here pound sign!

But woh, something else changed: we're also *not* getting the "row clicked" text anymore. If I click *just* the row, I get it, but if I click the delete icon, it only triggers the event on *that* element. What the heck just happened?

The Event (e) Listener Argument

Back up a step. Whenever a listener function is called, your browser passes it an *event* argument, commonly just named `e`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62  {% block javascripts %}
... lines 63 - 64
65      <script>
66          $(document).ready(function() {
... lines 67 - 68
69              $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 73
74              });
... lines 75 - 78
79          });
80      </script>
81  {% endblock %}
```

This `e` variable is *packed* with information and some functions. The most important is `e.preventDefault()`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62  {% block javascripts %}
... lines 63 - 64
65      <script>
66          $(document).ready(function() {
... lines 67 - 68
69              $table.find('.js-delete-rep-log').on('click', function (e) {
70                  e.preventDefault();
... lines 71 - 73
74              });
... lines 75 - 78
79          });
80      </script>
81  {% endblock %}
```

Another is `e.stopPropagation()`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();
                e.stopPropagation();
... lines 72 - 73
            });
... lines 75 - 78
        });
    </script>
{% endblock %}
```

It turns out that when you return `false` from a listener function, it is equivalent to calling `e.preventDefault()` *and* `e.stopPropagation()`. To prove it, remove the `return false` and refresh:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();
                e.stopPropagation();

                console.log('todo delete!');
            });
... lines 75 - 78
        });
    </script>
{% endblock %}
```

Yep, same behavior: no `#` sign, but still no "row clicked" when we click the delete icon.

e.preventDefault() versus e.stopPropagation()

The `e.preventDefault()` says: don't do the default, browser behavior for this event. Normally, when you "click" a "link", your browser navigates to its `href`... which is a `#`. So cool, `e.preventDefault()` stops that! But `e.stopPropagation()` tells your browser to *not*

bubble this event any further up the DOM tree. And that's probably *not* what you want. Do you really want your event listener to be *so* bold that it decides to prevent *all* other listeners from firing? I've literally *never* had a use-case for this.

So get rid of that pesky `e.stopPropagation()` and refresh again:

81 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62 {% block javascripts %}
... lines 63 - 64
65     <script>
66         $(document).ready(function() {
... lines 67 - 68
69             $table.find('.js-delete-rep-log').on('click', function (e) {
70                 e.preventDefault();
71
72                 console.log('todo delete!');
73             });
... lines 74 - 77
78         });
79     </script>
80 {% endblock %}
```

And things are back to normal!

You should use `e.preventDefault()` in *most* cases, but not always. Sometimes, like with a `keyup` event, if you call `preventDefault()`, that'll prevent whatever the user just typed from actually going into the text box.

Now, what else can this magical event argument help us with?

Chapter 5: The DOM Element Object

New goal! Eventually, when we click the trash icon, it will make an AJAX call. But before that, let's just see if we can turn the icon red. In our JavaScript code, we need to figure out exactly *which* `js-delete-rep-log` element was clicked.

How? I bet you've done it before... a *bunch* of times... by using the `this` variable. But don't! Wait on that - we'll talk about the infamous `this` variable later.

Using e.target

Because there's another way to find out *which* element was clicked... a better way, and it involves our magical `e` event argument. Just say `$(e.target)`. `target` is a property on the event object that points to the *actual* element that was clicked. Then, `.addClass('text-danger')`:

81 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62 {% block javascripts %}
... lines 63 - 64
65     <script>
66         $(document).ready(function() {
... lines 67 - 68
69             $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
72                 $(e.target).addClass('text-danger');
73             });
... lines 74 - 77
78         });
79     </script>
80 {% endblock %}
```

Cool? Go back, refresh. Eureka!

So what *is* this `e.target` thing exactly? I mean, is it a string? Or an object? And what else can we do with it?

Let's go digging! Add `console.log(e.target)`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
                $(e.target).addClass('text-danger');
                console.log(e.target);
            });
... lines 75 - 78
        });
    </script>
{% endblock %}
```

And then, refresh! Ok, click on some delete links. Huh... it just prints out the HTML itself. So, it's a string?

Actually, no... our browser is kinda lying to us: `e.target` is a DOM Element *object*. Google for that and find the W3Schools page all about it. You see, every element on the page is represented by a JavaScript object, a DOM Element object. My debugger is printing it like a string, but that's just for convenience... or inconvenience in this case. Nope, it's actually an object, with properties and methods that we can call. The W3Schools page shows all of this.

Pro Tip: Using console.dir()

And there's another way you can see the methods and properties on this object. Go back and change your `console.log()` to `console.dir()`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
                $(e.target).addClass('text-danger');
                console.dir(e.target);
            });
... lines 75 - 78
        });
    </script>
{% endblock %}
```

Now refresh. Click a link and check this out! It still gives you some information about what the element is, but now you can expand it to find a *huge* list of its properties and methods. Nice! One of the properties is called `className`, which we will use in a second.

If you're not familiar with `console.dir()`, it's bananas cool. Sometimes, `console.log()` gives you a string representation of something. But `console.dir()` tries to give you a tree of what that thing actually is. It's like programmer X-Ray vision!

DOM Element versus jQuery Object

So, question: how is a DOM Element object, like `e.target`, different than a jQuery object, like `$(e.target)` or something we selected, like `$table`? I mean, don't both represent an element on that page? And don't both allow us to interact with that element? Are they the same?

Not exactly. Whenever you have a jQuery object like `$table`, or `$(e.target)`, that actually represents an *array* of elements, even though there may only be *one* element. Let me show you: use `console.log()` to print out `e.target`, and also, `$(e.target)[0] === e.target`:

85 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
                $(e.target).addClass('text-danger');
                console.log(
                    e.target,
                    $(e.target)[0] === e.target
                );
            });
... lines 78 - 81
        });
    </script>
{% endblock %}
```

Go back, refresh, and click one of the links. It prints true! The jQuery object *is* an object, but it holds an *array* of DOM elements. And you can actually access the underlying DOM element objects by using the indexes, 0, 1, 2, 3 and so on. The jQuery object is just a fancy wrapper around them.

Try this example: search for all `.fa-trash` elements, find the third DOM element, which is index 2, and see if it's the same as the element that was just clicked: `e.target`:

86 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62  {% block javascripts %}
... lines 63 - 64
65      <script>
66          $(document).ready(function() {
... lines 67 - 68
69              $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
72                  $(e.target).addClass('text-danger');
73                  console.log(
74                      e.target,
75                      $(e.target)[0] === e.target,
76                      $('.fa-trash')[5] === e.target
77                  );
78              });
... lines 79 - 82
83          });
84      </script>
85  {% endblock %}
```

In theory, this should return true *only* when we click on the third trash icon.

So refresh and try it! Click the icons: false, false and then true! This is all an elaborate way of explaining that - under everything - we have these cool DOM Element objects. jQuery? That's just a fancy wrapper object that holds an array of these guys.

Of course, that fancy wrapper allows us to add a class by simply calling... `addClass()`:

86 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
62  {% block javascripts %}
... lines 63 - 64
65      <script>
66          $(document).ready(function() {
... lines 67 - 68
69              $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
72                  $(e.target).addClass('text-danger');
... lines 73 - 77
78              });
... lines 79 - 82
83          });
84      </script>
85  {% endblock %}
```

But now, we know that if we *wanted* to, we could do this directly on the DOM Element object. Try it: `e.target.className = e.target.className + ' text-danger'`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 70 - 71
                //$(e.target).addClass('text-danger');
                e.target.className = e.target.className+' text-danger';
            });
... lines 75 - 78
        });
    </script>
{% endblock %}
```

Try that out! Refresh. It works too!

It's not as elegant as using jQuery... and jQuery also helps handle browser incompatibilities, but feel empowered! Go tell a co-worker that you just learned how the Internet works!

Then come back, remove that new code and go back to using jQuery:

81 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 61
{% block javascripts %}
... lines 63 - 64
    <script>
        $(document).ready(function() {
... lines 67 - 68
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(e.target).addClass('text-danger');
            });
... lines 74 - 77
        });
    </script>
{% endblock %}
```

Chapter 6: The Magical this Variable & currentTarget

Turning the icon red is jolly good and all, but since we'll soon make an AJAX call, it would be *way* jollier if we could turn that icon into a spinning loader icon. But, there's a problem.

After the trash icon, type "Delete":

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr>
... lines 25 - 27
                        <td>
                            <a href="#" class="js-delete-rep-log">
                                <span class="fa fa-trash"></span>
                                Delete
                            </a>
                        </td>
                    </tr>
... lines 35 - 38
                {% endfor %}
... lines 40 - 48
            </table>
... lines 50 - 51
        </div>
... lines 53 - 59
    </div>
{% endblock %}
... lines 62 - 83
```

Now we have a trash icon with the word delete next to it. Back in our JavaScript, once again, `console.log()` the actual element that was clicked: `e.target`:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
... lines 64 - 65
    <script>
        $(document).ready(function() {
... lines 68 - 69
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(e.target).addClass('text-danger');
                console.log(e.target);
            });
... lines 76 - 79
        });
    </script>
{% endblock %}
```

e.target is Fooling Us!

Now, behold the madness! If I click the trash icon, `e.target` is a span. But if I click the delete text, it's actually the anchor! Woh!

True to what I said, `e.target` will be the *exact* one element that originally received the event, so click in this case. And that's a problem for us! Why? Well, I want to be able to find the `fa` span element and change it to a spinning icon. Doing that is going to be annoying, because if we click on the trash icon, `e.target` *is* that element. But if we click on the word delete, then we need to look inside of `e.target` to find the span.

Hello e.currentTarget

It would be WAY more hipster if we could retrieve the element that the listener was *attached* to. In other words, which `js-delete-rep-log` was clicked? That would make it *super* easy to look for the `fa` span inside of it and make the changes we need.

No problem! Change `e.target` to `e.currentTarget` and high-five yourself:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
... lines 64 - 65
    <script>
        $(document).ready(function() {
... lines 68 - 69
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(e.target).addClass('text-danger');
                console.log(e.currentTarget);
            });
... lines 76 - 79
        });
    </script>
{% endblock %}
```

Yep, this ends up being *much* more useful than `e.target`. Now when we refresh and click the trash icon, it's the anchor tag. Click the delete icon, it's *still* the anchor tag. No matter which element we *actually* click, `e.currentTarget` returns the original element that we attached the listener to.

Enter: this (versus currentTarget)

In fact, try this: `console.log(e.currentTarget === this)`:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
... lines 64 - 65
    <script>
        $(document).ready(function() {
... lines 68 - 69
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(e.target).addClass('text-danger');
                console.log(e.currentTarget === this);
            });
... lines 76 - 79
        });
    </script>
{% endblock %}
```

Refresh! And click anywhere on the delete link. It's *always* `true`.

There's a good chance that you've been using the `this` variable for years inside of your

listener functions to find the element that was clicked. And now we know the true and dramatic story behind it! `this` is equivalent to `e.currentTarget`, the DOM Element that we originally attached our listener to.

Ultimately that means that we can say, `$(this).addClass('text-danger')`:

82 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
63 {% block javascripts %}
... lines 64 - 65
66     <script>
67         $(document).ready(function() {
... lines 68 - 69
70             $table.find('.js-delete-rep-log').on('click', function (e) {
71                 e.preventDefault();
72
73                 $(this).addClass('text-danger');
74             });
... lines 75 - 78
79         });
80     </script>
81 {% endblock %}
```

That will always add the `text-danger` link to the anchor tag.

And finally, we can *easily* change our icon to a spinner! Just use `$(this).find('.fa')` to find the icon inside of the anchor. Then, `.removeClass('fa-trash')`, `.addClass('fa-spinner')` and `.addClass('fa-spin')`:

86 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 62
{% block javascripts %}
... lines 64 - 65
    <script>
        $(document).ready(function() {
... lines 68 - 69
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(this).addClass('text-danger');
                $(this).find('.fa')
                    .removeClass('fa-trash')
                    .addClass('fa-spinner')
                    .addClass('fa-spin');
            });
... lines 79 - 82
        });
    </script>
{% endblock %}
```

Refresh! Show me a spinner! There it is! It doesn't matter if we click the "Delete" text or the trash icon itself.

So, use the `this` variable, it's your friend. But realize what's going on: `this` is just a shortcut to `e.currentTarget`. That fact is going to become *critically* important in just a little while.

Now that we've learned this, remove the "delete" text... it's kinda ugly:

85 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
 {% block body %}
     <div class="row">
         <div class="col-md-7">
... lines 6 - 12
             <table class="table table-striped js-rep-log-table">
... lines 14 - 22
                 {% for repLog in repLogs %}
                     <tr>
... lines 25 - 27
                         <td>
                             <a href="#" class="js-delete-rep-log">
                                 <span class="fa fa-trash"></span>
                             </a>
                         </td>
                     </tr>
... lines 34 - 37
                 {% endfor %}
... lines 39 - 47
             </table>
... lines 49 - 50
         </div>
... lines 52 - 58
     </div>
 {% endblock %}
... lines 61 - 85
```

Chapter 7: A Great Place to Hide Things! The data- Attributes

Time to *finally* hook up the AJAX and delete one of these rows! Woohoo!

As an early birthday gift, I already took care of the server-side work for us. If you want to check it out, it's inside of the `src/AppBundle/Controller` directory: `RepLogController`:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 2
namespace AppBundle\Controller;
... lines 4 - 13
class RepLogController extends BaseController
{
... lines 16 - 129
}
```

I have a bunch of different RESTful API endpoints and one is called, `deleteRepLogAction()`:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 5
use AppBundle\Entity\RepLog;
... line 7
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... line 10
use Symfony\Component\HttpFoundation\Response;
... lines 12 - 13
class RepLogController extends BaseController
{
... lines 16 - 46
    /**
     * @Route("/reps/{id}", name="rep_log_delete")
     * @Method("DELETE")
     */
    public function deleteRepLogAction(RepLog $repLog)
    {
        $this->denyAccessUnlessGranted('IS_AUTHENTICATED_REMEMBERED');
        $em = $this->getDoctrine()->getManager();
        $em->remove($repLog);
        $em->flush();

        return new Response(null, 204);
    }
... lines 60 - 129
}
```

As long as we make a `DELETE` request to `/reps/ID-of-the-rep`, it'll delete it and return a blank response. Happy birthday!

Back in `index.html.twig`, inside of our listener function, how can we figure out the DELETE URL for *this* row? Or, even more basic, what's the ID of *this* specific RepLog? I have no idea! Yay!

We know that *this* link is being clicked, but it doesn't give us any information about the RepLog itself, like its ID or delete URL.

Adding a data-url Attribute

This is a *really* common problem, and the solution is to *somehow* attach extra metadata to our DOM about the RepLog, so we can read it in JavaScript. And guess what! There's an official, standard, proper way to do this! It's via a *data* attribute. Yep, according to those silly "rules" of the web, you're not really supposed to invent new attributes for your elements. Well, unless the attribute starts with `data-`, followed by lowercase letters. That's *totally* allowed!

Go Deeper!

> You can actually read the "data attributes" spec here: http://bit.ly/dry-spec-about-data-attributes

So, add an attribute called `data-url` and set it equal to the DELETE URL for *this* RepLog. The Symfony way of generating this is with `path()`, the name of the route - `rep_log_delete` - and the id: `repLog.id`:

98 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr>
... lines 25 - 27
                        <td>
                            <a href="#"
                              class="js-delete-rep-log"
                              data-url="{{ path('rep_log_delete', {id: repLog.id}) }}"
                            >
                              <span class="fa fa-trash"></span>
                            </a>
                        </td>
                    </tr>
... lines 37 - 40
                {% endfor %}
... lines 42 - 50
            </table>
... lines 52 - 53
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 98
```

Reading data- Attributes

Sweet! To read that in JavaScript, simply say `var deleteUrl = $(this)`, which we know is the link, `.data('url')`:

98 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
                e.preventDefault();

                $(this).addClass('text-danger');
                $(this).find('.fa')
                    .removeClass('fa-trash')
                    .addClass('fa-spinner')
                    .addClass('fa-spin');

                var deleteUrl = $(this).data('url');
... lines 82 - 89
            });
... lines 91 - 94
        });
    </script>
{% endblock %}
```

That's a little bit of jQuery magic: `.data()` is a shortcut to read a data attribute.

Tip

`.data()` is a wrapper around core JS functionality: the `data-*` attributes are also accessible directly on the DOM Element Object:

```
var deleteUrl = $(this)[0].dataset.url;
```

Finally, the AJAX call is really simple! I'll use `$.ajax`, set `url` to `deleteUrl`, `method` to `DELETE`, and `ice_cream` to `yes please!`. I mean, `success`, set to a function:

98 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
 {% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 80
                var deleteUrl = $(this).data('url');
... line 82
                $.ajax({
                    url: deleteUrl,
                    method: 'DELETE',
                    success: function() {
... line 87
                    }
                });
            });
... lines 91 - 94
        });
    </script>
 {% endblock %}
```

Hmm, so after this finishes, we probably want the *entire* row to disappear. Above the AJAX call, find the row with `$row = $(this).closest('tr')`:

98 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 80
                var deleteUrl = $(this).data('url');
                var $row = $(this).closest('tr');
                $.ajax({
                    url: deleteUrl,
                    method: 'DELETE',
                    success: function() {
... line 87
                    }
                });
            });
... lines 91 - 94
        });
    </script>
{% endblock %}
```

In other words, start with the link, and go up the DOM tree until you find a `tr` element. Oh, and reminder, this is `$row` because this is a jQuery object! Inside `success`, say `$row.fadeOut()` for just a *little* bit of fancy:

98 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 80
                var deleteUrl = $(this).data('url');
                var $row = $(this).closest('tr');
                $.ajax({
                    url: deleteUrl,
                    method: 'DELETE',
                    success: function() {
                        $row.fadeOut();
                    }
                });
            });
... lines 91 - 94
        });
    </script>
{% endblock %}
```

Ok, try that out! Refresh, delete my coffee cup and life is good. And if I refresh, it's truly gone. Oh, but dang, if I delete my cup of coffee record, the total weight at the bottom does *not* change. I need to refresh the page to do that. LAME! I'll re-add my coffee cup. Now, let's fix that!

Adding data-weight Metadata

If we somehow knew what the weight was for *this* specific row, we could read the *total* weight and just subtract it when it's deleted. So how can we figure out the weight for this row? Well, we could just read the HTML of the third column... but that's kinda shady. Instead, why not use another `data-` attribute?

On the `<tr>` element, add a `data-weight` attribute set to `repLog.totalWeightLifted`:

101 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 22
                {% for repLog in repLogs %}
                    <tr data-weight="{{ repLog.totalWeightLifted }}">
... lines 25 - 35
                    </tr>
... lines 37 - 40
                {% endfor %}
... lines 42 - 50
            </table>
... lines 52 - 53
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 101
```

Also, so that we know *which* `th` to update, add a class: `js-total-weight`:

101 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 42
                <tfoot>
                    <tr>
... lines 45 - 46
                        <th class="js-total-weight">{{ totalWeight }}</th>
... line 48
                    </tr>
                </tfoot>
            </table>
... lines 52 - 53
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 101
```

Let's hook this up! *Before* the AJAX call - that's important, we'll find out why soon - find the total weight container by saying `$table.find('.js-total-weight')`:

101 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 81
                var $row = $(this).closest('tr');
                var $totalWeightContainer = $table.find('.js-total-weight');
... lines 84 - 92
            });
... lines 94 - 97
        });
    </script>
{% endblock %}
```

Next add `var newWeight` set to `$totalWeightContainer.html() - $row.data('weight')`:

101 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 81
                var $row = $(this).closest('tr');
                var $totalWeightContainer = $table.find('.js-total-weight');
                var newWeight = $totalWeightContainer.html() - $row.data('weight');
... lines 85 - 92
            });
... lines 94 - 97
        });
    </script>
{% endblock %}
```

Use that inside `success`: `$totalWeightContainer.html(newWeight)`:

101 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        $(document).ready(function() {
... lines 70 - 71
            $table.find('.js-delete-rep-log').on('click', function (e) {
... lines 73 - 81
                var $row = $(this).closest('tr');
                var $totalWeightContainer = $table.find('.js-total-weight');
                var newWeight = $totalWeightContainer.html() - $row.data('weight');
                $.ajax({
... lines 86 - 87
                    success: function() {
                        $row.fadeOut();
                        $totalWeightContainer.html(newWeight);
                    }
                });
            });
... lines 94 - 97
        });
    </script>
{% endblock %}
```

Let's give this fanciness a try. Go back refresh. 459? Hit delete, it's gone. 454.

Now, how about we get into trouble with some JavaScript objects!

Chapter 8: Organizing with Objects!

Ok, this all looks pretty good... except that our code is just a bunch of functions and callback functions! Come on people, if this were PHP code, we would be using classes and objects. Let's hold our JavaScript to that same standard: let's use objects.

Creating an Object

How do you create an object? There are a few ways, but for now, it's as simple as var `RepLogApp = {}`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Yep, that's an object. Yea, I know, it's just an associative array but an associative array *is* an object in JavaScript. And its keys become the properties and methods on the object. See, JavaScript doesn't have *classes* like PHP, only objects. Well, that's not entirely true, but we'll save that for a future tutorial.

Adding a Method

Anyways, let's give our object a new method: an `initialize` key set to a `function()`. We'll call this when the page loads, and its job will be to attach all the event handlers for all the events that we need on our table. Give it a `$wrapper` argument:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 80
            },
... lines 82 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Setting a Property

Before we do anything else, set that `$wrapper` argument onto a property: `this.$wrapper = $wrapper`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
                this.$wrapper = $wrapper;
... lines 72 - 80
            },
... lines 82 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Yep, we just dynamically added a new property. This is the second time we've seen the `this` variable in JavaScript. And this time, it's more familiar: it refers to *this* object.

Next, copy our first listener registration code, but change `$table` to `this.$wrapper`. And instead of using a big ugly anonymous function, let's make this event call a new method on our object: `this.handleRepLogDelete`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
                this.$wrapper = $wrapper;

                this.$wrapper.find('.js-delete-rep-log').on(
                    'click',
                    this.handleRepLogDelete
                );
... lines 77 - 80
            },
... lines 82 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

We'll add that in a moment.

Repeat this for the other event listener: copy the registration line, change `$table` to `this.$wrapper`, and then on click, call `this.handleRowClick`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
                this.$wrapper = $wrapper;
... lines 72 - 76
                this.$wrapper.find('tbody tr').on(
                    'click',
                    this.handleRowClick
                );
            },
... lines 82 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

I already like it!

After `initialize`, create these methods! Add a key called, `handleRepLogDelete` set to a new function:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            handleRepLogDelete: function(e) {
... lines 84 - 103
            },
... lines 105 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Then go copy all of our original handler code, delete it, and put it here:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            handleRepLogDelete: function(e) {
                e.preventDefault();

                $(this).addClass('text-danger');
                $(this).find('.fa')
                    .removeClass('fa-trash')
                    .addClass('fa-spinner')
                    .addClass('fa-spin');

                var deleteUrl = $(this).data('url');
                var $row = $(this).closest('tr');
                var $totalWeightContainer = $table.find('.js-total-weight');
                var newWeight = $totalWeightContainer.html() - $row.data('weight');
                $.ajax({
                    url: deleteUrl,
                    method: 'DELETE',
                    success: function() {
                        $row.fadeOut();
                        $totalWeightContainer.html(newWeight);
                    }
                });
            },
... lines 105 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Make sure you have the, `e` argument exactly like before.

Do the same thing for our other method: `handleRowClick` set to a `function() {}`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 105
106             handleRowClick: function() {
... line 107
108             }
109         };
... lines 110 - 114
115     </script>
116 {% endblock %}
```

I'm not using the, `e` argument, so I don't *need* to add it. Copy the `console.log()` line, delete it, and put it here:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 105
106             handleRowClick: function() {
107                 console.log('row clicked!');
108             }
109         };
... lines 110 - 114
115     </script>
116 {% endblock %}
```

Don't Call your Handler Function: Pass It

There's one *teenie* detail I want you to notice: when we specify the event callback, `this.handleRepLogDelete` - we're *not* executing it:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 72
                this.$wrapper.find('.js-delete-rep-log').on(
... line 74
                    this.handleRepLogDelete
                );
                this.$wrapper.find('tbody tr').on(
... line 78
                    this.handleRowClick
                );
            },
... lines 82 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

I mean, there are no `()` on the end of it. Nope, we're simply passing the function as a reference to the `on()` function. If you forget and add `()`, things will get crazy.

Initializing (not Instantiating) the Object

Back in the `(document).ready()`, our job is really simple: find the `$table` and then pass it to `RepLogApp.initialize()`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 108
        };

        $(document).ready(function() {
            var $table = $('.js-rep-log-table');
            RepLogApp.initialize($table);
        });
    </script>
{% endblock %}
```

The cool thing about this approach is that now we have an entire object who's job is to work inside of `this.$wrapper`.

Ok, let's try this! Go back and refresh! Hit delete! Ah, it fails!

> Variable $table is not defined.

The problem is inside of `handleRepLogDelete`. Ah, cool, this makes total sense. Before, we had a `$table` variable defined above the function. That's gone, but no problem! Just use `this.$wrapper`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            handleRepLogDelete: function(e) {
... lines 84 - 93
                var $totalWeightContainer = this.$wrapper.find('.js-total-weight');
... lines 95 - 103
            },
... lines 105 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

You can already see how handy an object can be.

Ok, go back and refresh again. Open up the console, click delete and... whoa! That doesn't work either! The errors is on the exact same line. What's going on here? It says:

> Cannot read property 'find' of undefined

How can `this.$wrapper` be undefined? Let's find out.

Chapter 9: "Static" Objects & the this Variable

We just found out that, *somehow*, `this.$wrapper` is *not* our jQuery object, it's undefined!

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 82
83              handleRepLogDelete: function(e) {
... lines 84 - 93
94                  var $totalWeightContainer = this.$wrapper.find('.js-total-weight');
... lines 95 - 103
104             },
... lines 105 - 108
109         };
... lines 110 - 114
115     </script>
116 {% endblock %}
```

Rude! How is that even possible! The answer! Because JavaScript is weird, *especially* when it comes to the crazy `this` variable!

When this is not this

Here's the deal: whenever you are in a callback function, like the `success` callback of an AJAX call, the callback of an event listener, or even when passing a callback to the `setTimeout()` function, the `this` variable in your callback *changes* to be something else. And we already knew that! We know that `this` in our event handler is actually a reference to the DOM Element object that was clicked. So the `this` variable in `handleRepLogDelete` is *not* our `RepLogApp` object, even though we're *inside* of that object. Creepy!

We're going to talk a lot more about this situation... in a moment.

Referencing your Object "Statically"

Fortunately, for now, the fix is easy. If you think about it, the `RepLogApp` object is very similar to a class in PHP that has *only* static properties and methods. I mean, could we create multiple `RepLogApp` objects? Nope! There can only ever be one. And because of that, each property - like `$wrapper` - acts like a static property: you set and access it,

but it's attached to our "static", single object: `RepLogApp`, not to an individual *instance* of `RepLogApp`.

If this is hard to wrap your head around, don't worry! Coming from PHP, objects in JavaScript are weird... and they'll get stranger before we're done. But, most things you can do in PHP you can also do in JavaScript... it just looks different. The stuff inside the object may not have some special `static` keyword on them, but this is what static properties and methods look like in JavaScript.

And like static properties and methods in PHP, you can reference them by their class name. Well, in JavaScript, that mean, by their object name - `RepLogApp`:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            handleRepLogDelete: function(e) {
... lines 84 - 93
                var $totalWeightContainer = RepLogApp.$wrapper.find('.js-total-weight');
... lines 95 - 103
            },
... lines 105 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

Ok, go back and refresh now. Hit delete. It *actually* works! Sorry, I shouldn't sound so surprised!

Refactoring to More Methods!

Since we're running out of items, let's add a few more!

Now that we have a fancy object, we can use it to get even *more* organized, by breaking big functions into smaller ones.

For example, we could create a new function called, `updateTotalWeightLifted`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            updateTotalWeightLifted: function() {
... lines 84 - 89
            },
... lines 91 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

Instead of figuring out the total weight lifted here and doing the update down in the success callback:

117 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            handleRepLogDelete: function(e) {
... lines 84 - 93
                var $totalWeightContainer = RepLogApp.$wrapper.find('.js-total-weight');
                var newWeight = $totalWeightContainer.html() - $row.data('weight');
                $.ajax({
... lines 97 - 98
                    success: function() {
... line 100
                        $totalWeightContainer.html(newWeight);
                    }
                });
            },
... lines 105 - 108
        };
... lines 110 - 114
    </script>
{% endblock %}
```

We'll just call this method and have *it* do all that heavy lifting.

Add `var totalWeight = 0`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 82
83              updateTotalWeightLifted: function() {
84                  var totalWeight = 0;
... lines 85 - 89
90              },
... lines 91 - 117
118         };
... lines 119 - 123
124     </script>
125 {% endblock %}
```

Then I'll say, `this.$wrapper`, which I can do because we're *not* in a callback function: `this` is our object. Then, `.find` to look for all `tbody tr` elements, and `.each()` to loop over them:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 82
83              updateTotalWeightLifted: function() {
84                  var totalWeight = 0;
85                  this.$wrapper.find('tbody tr').each(function() {
... line 86
87                  });
... lines 88 - 89
90              },
... lines 91 - 117
118         };
... lines 119 - 123
124     </script>
125 {% endblock %}
```

But stop! Notice that when you use `.each()`, you pass it a callback function! So guess what? Inside, `this` is no longer our `RepLogApp` object, it's something different. In this case, `this` is the individual `tr` DOM Element object that we're looping over in this moment.

Inside, add up all the total weights with `totalWeight += $(this).data()` and read the `data-weight` attribute:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            updateTotalWeightLifted: function() {
                var totalWeight = 0;
                this.$wrapper.find('tbody tr').each(function() {
                    totalWeight += $(this).data('weight');
                });
... lines 88 - 89
            },
... lines 91 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

Finally use `this.$wrapper.find()` to look for our `js-total-weight` element and set its HTML to `totalWeight`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 82
            updateTotalWeightLifted: function() {
                var totalWeight = 0;
                this.$wrapper.find('tbody tr').each(function() {
                    totalWeight += $(this).data('weight');
                });

                this.$wrapper.find('.js-total-weight').html(totalWeight);
            },
... lines 91 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

Cool!

Down in `handleRepLogDelete`, we don't need any of this logic anymore, nor this logic. We just need to call our new function. The only gotcha is that the `fadeOut()` function doesn't actually remove the row from the DOM, so our new weight-totaling function would *still* count its weight.

Fix it by telling `fadeOut()` to use `normal` speed, pass it a function to be called when it finishes fading, and then say `$row.remove()` to fully remove it from the DOM:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 91
            handleRepLogDelete: function(e) {
... lines 93 - 100
                var deleteUrl = $(this).data('url');
                var $row = $(this).closest('tr');
                $.ajax({
... lines 104 - 105
                    success: function() {
                        $row.fadeOut('normal', function() {
                            $row.remove();
... line 109
                        });
                    }
                });
            },
... lines 114 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

Now we can call `updateTotalWeightLifted`.

But check this out: we're actually inside of *another* callback function, which is inside of a callback function, inside of our entire function which is itself a callback! So, `this` is *definitely* not our `RepLogApp` object.

No worries, play it safe and use `RepLogApp.updateTotalWeightLifted()` instead:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 91
            handleRepLogDelete: function(e) {
... lines 93 - 100
                var deleteUrl = $(this).data('url');
                var $row = $(this).closest('tr');
                $.ajax({
... lines 104 - 105
                    success: function() {
                        $row.fadeOut('normal', function() {
                            $row.remove();
                            RepLogApp.updateTotalWeightLifted();
                        });
                    }
                });
            },
... lines 114 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

That's the equivalent in PHP of calling a static method by using its *class* name.

Ok, try it out! Refresh the page. We're at 765. Now delete a row... 657! Nice! Let's finally figure out what's *really* going on with the `this` variable... *and* how to make it act better!

Chapter 10: Getting to the bottom of the this Variable

In PHP, when we call a function like `updateTotalWeightLifted()`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 91
92              handleRepLogDelete: function(e) {
... lines 93 - 102
103                 $.ajax({
... lines 104 - 105
106                     success: function() {
107                         $row.fadeOut('normal', function() {
... line 108
109                             RepLogApp.updateTotalWeightLifted();
110                         });
111                     }
112                 });
113             },
... lines 114 - 117
118         };
... lines 119 - 123
124     </script>
125 {% endblock %}
```

We expect the `this` variable inside of that function to be whatever object we're inside of right now. In that case, it is. But in so many other cases, `this` is something different! Like inside `handleRowClick` and `handleRepLogDelete`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
70              initialize: function($wrapper) {
... lines 71 - 72
73                  this.$wrapper.find('.js-delete-rep-log').on(
... line 74
75                      this.handleRepLogDelete
76                  );
77                  this.$wrapper.find('tbody tr').on(
... line 78
79                      this.handleRowClick
80                  );
81              },
... lines 82 - 117
118         };
... lines 119 - 123
124     </script>
125 {% endblock %}
```

What's going on? And more importantly, how can we fix it? When I'm inside a method in an object, I want `this` to act normal: I want it to point to my *object*.

How do I Know what this Is?

Here's the deal: when you call a function in JavaScript, you can *choose* to change what `this` is inside of that function when you call it. That means you could have one function and 10 different people could call your function and decide to set `this` to 10 different things.

Now, in reality, it's not that bad. But we *do* need to remember one rule of thumb: whenever you have a callback function - meaning someone else is calling a function after something happens - `this` will have changed. We've already seen this a lot: in the `click` functions, inside of `.each()`, inside of `success` and even inside of `$row.fadeOut()`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 72
                this.$wrapper.find('.js-delete-rep-log').on(
... line 74
                    this.handleRepLogDelete
                );
                this.$wrapper.find('tbody tr').on(
... line 78
                    this.handleRowClick
                );
            },

            updateTotalWeightLifted: function() {
... line 84
                this.$wrapper.find('tbody tr').each(function() {
                    totalWeight += $(this).data('weight');
                });
... lines 88 - 89
            },

            handleRepLogDelete: function(e) {
... lines 93 - 102
                $.ajax({
... lines 104 - 105
                    success: function() {
                        $row.fadeOut('normal', function() {
... line 108
                            RepLogApp.updateTotalWeightLifted();
                        });
                    }
                });
... lines 113 - 117
        };
... lines 119 - 123
    </script>
{% endblock %}
```

So what *is* `this` inside of these functions? It depends on the situation, so you need to read the docs for the `success` function, the `fadeOut()` function or the `.each()` function to be sure. For `fadeOut()`, `this` ends up being the DOM Element that just finished fading

out. So, we can actually call `$(this).remove()`:

126 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 91
92              handleRepLogDelete: function(e) {
... lines 93 - 102
103                 $.ajax({
... lines 104 - 105
106                     success: function() {
107                         $row.fadeOut('normal', function() {
108                             $(this).remove();
... line 109
110                         });
111                     }
112                 });
113             },
... lines 114 - 117
118         };
... lines 119 - 123
124     </script>
125 {% endblock %}
```

That's the same as before.

Being a Magician with this!

Let's have a little fun with the weirdness of `this`. Create a new function - just for debugging - called `whatIsThis` with a single argument, a `greeting`. Inside, just `console.log()` `this` and our `greeting`:

132 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 84
            whatIsThis: function(greeting) {
                console.log(this, greeting);
            },
... lines 88 - 123
        };
... lines 125 - 129
    </script>
{% endblock %}
```

Next, at the bottom of `initialize`, add `this.whatIsThis()` and pass it `hello`:

132 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 81
                this.whatIsThis('hello');
            },
... lines 84 - 123
        };
... lines 125 - 129
    </script>
{% endblock %}
```

Simple enough! And since we're calling this function directly - not as a callback - I would expect `this` to *actually* be what we expect: our `RepLogApp` object. Let's find out. Refresh! Expand the logged object. Yea, it's `RepLogApp`! Cool!

But now, let's get tricky! Create a new variable called `newThis` and set it to an object with important stuff like `cat` set to `meow` and `dog` set to `woof`:

133 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
70              initialize: function($wrapper) {
... lines 71 - 81
82                  var newThis = {cat: 'meow', dog: 'woof'};
... line 83
84              },
... lines 85 - 124
125         };
... lines 126 - 130
131     </script>
132 {% endblock %}
```

To force `newThis` to be `this` inside our function, call the function *indirectly* with `this.whatIsThis.call()` and pass it `newThis` and the greeting, `hello`:

133 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
70              initialize: function($wrapper) {
... lines 71 - 81
82                  var newThis = {cat: 'meow', dog: 'woof'};
83                  this.whatIsThis.call(newThis, 'hello');
84              },
... lines 85 - 124
125         };
... lines 126 - 130
131     </script>
132 {% endblock %}
```

Oh, and quick note: `this.whatIsThis` is, obviously, a function. But in JavaScript, functions are actually *objects* themselves! And there are a number of different methods that you can call on them, like `.call()`. The first argument to `call()` is the variable that should be used for `this`, followed by any arguments that should be passed to the function itself.

Refresh now and check this out! `this` is now our thoughtful cat, meow, dog, woof object. That is what is happening behind the scenes with your callback functions.

Now that we understand the magic behind `this`, how can we fix it? How can we guarantee that `this` is always our `RepLogApp` object when we're inside of it?

Chapter 11: Fixing "this" with bind()

So how can we fix this? If we're going to be fancy and use objects in JavaScript, I don't want to have to worry about whether or not `this` is *actually* `this` in each function! That's no way to live! Nope, I want to know *confidently* that inside of my `whatIsThis` function, `this` is my `RepLogApp` object... not a random array of pets and their noises.

More importantly, I want that same guarantee down in each callback function: I want to be absolutely sure that `this` is *this* object, exactly how we'd expect our methods to work.

And yes! This is possible: we can take back control! Create a new variable: `var boundWhatIsThis = this.whatIsThis.bind(this)`:

134 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 81
                var newThis = {cat: 'meow', dog: 'woof'};
                var boundWhatIsThis = this.whatIsThis.bind(this);
... line 84
            },
... lines 86 - 125
        };
... lines 127 - 131
    </script>
{% endblock %}
```

Just like `call()`, `bind()` is a method you can call on functions. You pass it what you want `this` to be - in this case our `RepLogApp` object - and it returns a *new* function that, when called, will *always* have `this` set to whatever you passed to `bind()`. Now, when we say `boundWhatIsThis.call()` and *try* to pass it an alternative `this` object, that will be ignored:

134 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 81
                var newThis = {cat: 'meow', dog: 'woof'};
                var boundWhatIsThis = this.whatIsThis.bind(this);
                boundWhatIsThis.call(newThis, 'hello');
            },
... lines 86 - 125
        };
... lines 127 - 131
    </script>
{% endblock %}
```

Try it out: refresh! Yes! Now `this` is `this` again!

Binding all of our Listener Functions

Delete that debug code. Now that we have a way to *guarantee* the value of `this`, all we need to do is repeat the trick on any listener functions. In practice, that means that whenever you register an event handling function, you should call `.bind(this)`. Add it to both event listeners:

127 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
            initialize: function($wrapper) {
... lines 71 - 72
                this.$wrapper.find('.js-delete-rep-log').on(
... line 74
                    this.handleRepLogDelete.bind(this)
                );
                this.$wrapper.find('tbody tr').on(
... line 78
                    this.handleRowClick.bind(this)
                );
            },
... lines 82 - 118
        };
... lines 120 - 124
    </script>
{% endblock %}
```

Replacing this in Event Listeners

But wait! That's going to totally mess up our function: we're *relying* on `this`: expecting it to be the DOM Element object that was clicked! Dang! But no problem, because we already learned that `this` is equal to `e.currentTarget`. Fix the problem by adding `var $link = $(e.currentTarget)`:

127 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 90
91              handleRepLogDelete: function(e) {
92                  e.preventDefault();
93
94                  var $link = $(e.currentTarget);
... lines 95 - 113
114             },
... lines 115 - 118
119         };
... lines 120 - 124
125     </script>
126 {% endblock %}
```

Now just change the `$(this)` to `$link`:

127 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65  {% block javascripts %}
... lines 66 - 67
68      <script>
69          var RepLogApp = {
... lines 70 - 90
91              handleRepLogDelete: function(e) {
... lines 92 - 93
94                  var $link = $(e.currentTarget);
95
96                  $link.addClass('text-danger');
97                  $link.find('.fa')
... lines 98 - 101
102                 var deleteUrl = $link.data('url');
103                 var $row = $link.closest('tr');
... lines 104 - 113
114             },
... lines 115 - 118
119         };
... lines 120 - 124
125     </script>
126 {% endblock %}
```

And life is good!

Try it out! Refresh, click, and winning!

Finally, we can fix something that's been bothering me. Instead of saying `RepLogApp`, I want to use `this`. We talked earlier about how `RepLogApp` is kind of like a static object, and just like in PHP, when something is static, you can reference it by its object name, or really, class name in PHP.

Always Referencing this, instead of RepLogApp

But that's not going to be true forever: in a few minutes, we're going to learn how to design objects that you can *instantiate*, meaning we could have many `RepLogApp` objects. For example, we could have *five* tables on our page and instantiate five separate `RepLogApp` objects, one for each table. Once we do that, we won't be able to simply reference our object with `RepLogApp` anymore, because we might have five of them. But if we always reference our object internally with `this`, it'll be *future* proof: working now, and also after we make things fancier.

Of course, the problem is that inside of the callback, `this` won't be our `RepLogApp` object anymore. How could we fix this? There are two options. First, we could `bind` our success function to `this`. Then, now that `this` is our `RepLogApp` object inside of `success`, we could also bind our `fadeOut` callback to `this`. *Finally*, that would let us call `this.updateTotalWeightLifted()`.

But wow, that's a lot of work, and it'll be a bit ugly! Instead, there's a simpler way. First, realize that whenever you have an anonymous function, you *could* refactor it into an individual method on your object. If we did that, then I would recommend binding that function so that `this` is the `RepLogApp` object inside.

But if that feels like overkill and you want to keep using anonymous functions, then simply go above the callback and add `var self = this`:

128 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 90
            handleRepLogDelete: function(e) {
... lines 92 - 103
                var self = this;
                $.ajax({
... lines 106 - 113
                });
            },
... lines 116 - 119
        };
... lines 121 - 125
    </script>
{% endblock %}
```

The variable `self` is *not* important in any way - I just made that up. So, it doesn't change inside of callback functions, which means we can say `self.updateTotalWeightLifted()`:

128 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 67
    <script>
        var RepLogApp = {
... lines 70 - 90
            handleRepLogDelete: function(e) {
... lines 92 - 103
                var self = this;
                $.ajax({
... lines 106 - 107
                    success: function() {
                        $row.fadeOut('normal', function() {
... line 110
                            self.updateTotalWeightLifted();
                        });
                    }
                });
            },
... lines 116 - 119
        };
... lines 121 - 125
    </script>
{% endblock %}
```

Try that! Ah, it works *great*.

So there are two important takeaways:

1. Use `bind()` to make sure that `this` is always `this` inside any methods in your object.
2. Make sure to reference your object with `this`, instead of your object's name. This isn't an absolute rule, but unless you know what you're doing, this will give you more flexibility in the long-run.

Chapter 12: Immediately Invoked Function Expression!

Our code is growing up! And to keep going, it's really time to move our `RepLogApp` into its own external JavaScript file. For now, let's keep this real simple: inside the `web/` directory - which is the public document root for the project - and in `assets/`, I'll create a new `js/` directory. Then, create a new file: `RepLogApp.js`. Copy *all* of our `RepLogApp` object and paste it here:

53 lines | web/assets/js/RepLogApp.js

```
var RepLogApp = {
    initialize: function ($wrapper) {
        this.$wrapper = $wrapper;

        this.$wrapper.find('.js-delete-rep-log').on(
            'click',
            this.handleRepLogDelete.bind(this)
        );
        this.$wrapper.find('tbody tr').on(
            'click',
            this.handleRowClick.bind(this)
        );
    },
    updateTotalWeightLifted: function () {
        var totalWeight = 0;
        this.$wrapper.find('tbody tr').each(function () {
            totalWeight += $(this).data('weight');
        });

        this.$wrapper.find('.js-total-weight').html(totalWeight);
    },

    handleRepLogDelete: function (e) {
        e.preventDefault();

        var $link = $(e.currentTarget);

        $link.addClass('text-danger');
        $link.find('.fa')
            .removeClass('fa-trash')
            .addClass('fa-spinner')
```

```
31            .addClass('fa-spinner')
32            .addClass('fa-spin');
33
34        var deleteUrl = $link.data('url');
35        var $row = $link.closest('tr');
36        var self = this;
37        $.ajax({
38            url: deleteUrl,
39            method: 'DELETE',
40            success: function () {
41                $row.fadeOut('normal', function () {
42                    $(this).remove();
43                    self.updateTotalWeightLifted();
44                });
45            }
46        });
47    },
48
49    handleRowClick: function () {
50        console.log('row clicked!');
51    }
52 };
```

Add a good old-fashioned `script` tag to bring this in:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65 {% block javascripts %}
66     {{ parent() }}
67
68     <script src="{{ asset('assets/js/RepLogApp.js') }}"></script>
69
70     <script>
71         $(document).ready(function() {
72             var $table = $('.js-rep-log-table');
73             RepLogApp.initialize($table);
74         });
75     </script>
76 {% endblock %}
```

If you don't normally use Symfony, ignore the `asset()` function: it doesn't do anything special.

To make sure we didn't mess anything up, refresh! Let's add a few items to our list. Then, delete one. It works!

Private Functions in JavaScript

One of the advantages of having objects in PHP is the possibility of having *private* functions and properties. But, that doesn't exist in JavaScript: everything is publicly accessible! That means that anyone could call any of these functions, even if we don't intend for them to be used outside of the object.

That's not the end of the world, but it *is* a bummer! Fortunately, by being clever, we *can* create private functions and variables. You just need to think differently than you would in PHP.

Creating a Faux-Private Method

First, create a function at the bottom of this object called `_calculateTotalWeight`:

59 lines | web/assets/js/RepLogApp.js

```
1  var RepLogApp = {
   ... lines 2 - 49
50     _calculateTotalWeight: function() {
   ... lines 51 - 56
57     }
58 };
```

Its job will be to handle the total weight calculation logic that's currently inside `updateTotalWeightLifted`:

59 lines | web/assets/js/RepLogApp.js

```
1  var RepLogApp = {
   ... lines 2 - 49
50     _calculateTotalWeight: function() {
51         var totalWeight = 0;
52         this.$wrapper.find('tbody tr').each(function () {
53             totalWeight += $(this).data('weight');
54         });
55
56         return totalWeight;
57     }
58 };
```

We're making this change *purely* for organization: my intention is that we will *only* use this method from inside of this object. In other words, ideally, `calculateTotalWeight` would be private!

But since *everything* is public in JavaScript, a common standard is to prefix methods that should be treated as private with an underscore. It's a nice convention, but it doesn't enforce anything. Anybody could still call this from outside of the object.

Back in `updateTotalWeightLifted`, call it: `this._calculateTotalWeight()`:

59 lines | web/assets/js/RepLogApp.js

```
var RepLogApp = {
... lines 2 - 13
    updateTotalWeightLifted: function () {
        this.$wrapper.find('.js-total-weight').html(
            this._calculateTotalWeight()
        );
    },
... lines 19 - 57
};
```

Creating a Private Object

So how could we make this *truly* private? Well, you *can't* make methods or properties in an object private. BUT, you can make *variables* private, by taking advantage of variable *scope*. What I mean is, if I have access to the `RepLogApp` object, then I can call any methods on it. But if I *didn't* have access to this, or some other object, then of course I *wouldn't* be able to call any methods on it. I know that sounds weird, so let's do it!

At the bottom of this file, create another object called: `var Helper = {}`:

69 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 54
var Helper = {
... lines 56 - 67
};
```

Commonly, we'll organize our code so that each file has just one object, like in PHP. But eventually, this variable *won't* be public - it's just a helper meant to be used only inside of this file.

I'll even add some documentation: this is private, not meant to be called from outside!

69 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 51
/**
 * A "private" object
 */
var Helper = {
... lines 56 - 67
};
```

Just like before, give this an initialize, function with a `$wrapper` argument. And then say: `this.$wrapper = $wrapper`:

69 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 54
55 var Helper = {
56     initialize: function ($wrapper) {
57         this.$wrapper = $wrapper;
58     },
... lines 59 - 67
68 };
```

Move the `calculateTotalWeight()` function into *this* object, but take off the underscore:

69 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 54
55 var Helper = {
... lines 56 - 59
60     calculateTotalWeight: function() {
61         var totalWeight = 0;
62         this.$wrapper.find('tbody tr').each(function () {
63             totalWeight += $(this).data('weight');
64         });
65
66         return totalWeight;
67     }
68 };
```

Technically, if you have access to the `Helper` variable, then you're allowed to call `calculateTotalWeight`. Again, that whole `_` thing is just a convention.

Back in our original object, let's set this up: call `Helper.initialize()` and pass it `$wrapper`:

69 lines | web/assets/js/RepLogApp.js

```
1  var RepLogApp = {
2      initialize: function ($wrapper) {
3          this.$wrapper = $wrapper;
4          Helper.initialize(this.$wrapper);
... lines 5 - 13
14     },
... lines 15 - 49
50 };
... lines 51 - 69
```

Down below, call this: `Helper.calculateTotalWeight()`:

69 lines | web/assets/js/RepLogApp.js

```
var RepLogApp = {
    initialize: function ($wrapper) {
        this.$wrapper = $wrapper;
        Helper.initialize(this.$wrapper);
... lines 5 - 13
    },
    updateTotalWeightLifted: function () {
        this.$wrapper.find('.js-total-weight').html(
            Helper.calculateTotalWeight()
        );
    },
... lines 20 - 49
};
... lines 51 - 69
```

Double-check that everything still works: refresh! It does!

But, this `Helper` object is *still* public. What I mean is, we *still* have access to it *outside* of this file. If we try to `console.log(Helper)` from our template, it works just fine:

78 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 69
    <script>
        console.log(Helper);
... lines 72 - 75
    </script>
{% endblock %}
```

What I *really* want is the ability for me to choose *which* variables I want to make available to the outside world - like `RepLogApp` - and which I *don't*, like `Helper`.

The Self-Executing Function

The way you do that is with - dun dun dun - an *immediately invoked function expression*. Also known by its friends as a *self-executing function*. Basically, that means we'll wrap all of our code inside a function... that calls itself. It's weird, but check it out: `(function() {`, then indent everything. At the bottom, add the `})` and then `()`:

71 lines | web/assets/js/RepLogApp.js

```
1  (function() {
2      var RepLogApp = {
↕  ... lines 3 - 50
51     };
↕  ... lines 52 - 55
56     var Helper = {
↕  ... lines 57 - 68
69     };
70 })();
```

What?

There are two things to check out. First, all we're doing is creating a function: it starts on top, and ends at the bottom with the `}`. But by adding the `()`, we are immediately executing that function. We're creating a function and then calling it!

Why on earth would we do this? Because! Variable scope in JavaScript is function based. When you create a variable with `var`, it's only accessible from inside of the function where you created it. If you have functions inside of that function, they have access to it too, but ultimately, that function is its home.

Before, when we weren't inside of *any* function, our two variables effectively became global: we could access them from *anywhere*. But now that we're inside of a function, the `RepLogApp` and `Helper` variables are *only* accessible from inside of this self-executing function.

This means that when we refresh, we get `Helper` is not defined. We just made the `Helper` variable private!

Unfortunately... we also made our `RepLogApp` variable private, which means the code in our template will *not* work. We still need to somehow make `RepLogApp` available publicly, but not `Helper`. How? By taking advantage of the magical `window` object.

Chapter 13: The window Object & Global Variables

Now that we're using this fancy self-executing function, we don't have access to `RepLogApp` anymore:

71 lines | web/assets/js/RepLogApp.js

```
1  (function() {
2      var RepLogApp = {
... lines 3 - 50
51     };
... lines 52 - 69
70 })();
```

How can we fix that? Very simple. Instead of `var RepLogApp`, say `window.RepLogApp`:

71 lines | web/assets/js/RepLogApp.js

```
1  (function() {
2      window.RepLogApp = {
... lines 3 - 50
51     };
... lines 52 - 69
70 })();
```

Back in the template, I'll delete the `console.log()` for Helper:

78 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65 {% block javascripts %}
... lines 66 - 69
70     <script>
71         console.log(Helper);
... lines 72 - 75
76     </script>
77 {% endblock %}
```

And then go back and refresh. It works! No error in the console, and delete does its job!

What is this window?

So what the heck just happened? Here's the deal: when you're executing JavaScript in a browser - which for you is probably always - you always have access to a global `window` variable. In fact, it's even more important than that. This `window` variable

holds all of the global variables. What I mean is: if you set a key on the `window` object, like `RepLogApp`, this becomes a global variable. That means you can reference `RepLogApp` from *anywhere* else, and this is *actually* referencing `window.RepLogApp`. More on that in a second.

Passing Yourself Global Variables

Inside of our self-executing function, we - of course - also have access to any global variables, like window or the `$` jQuery variable. But, instead of relying on these global variables, you'll often see people *pass* those variables *into* the function. It's a little weird, so let's see it.

Right now, inside of our self-executing function, we're using two global variables: `window` and `$`, for `$.ajax`, for example:

71 lines | web/assets/js/RepLogApp.js

```
1  (function() {
2      window.RepLogApp = {
... lines 3 - 21
22         handleRepLogDelete: function (e) {
... lines 23 - 35
36             $.ajax({
... lines 37 - 44
45             });
46         },
... lines 47 - 50
51     };
... lines 52 - 69
70 })();
```

At the bottom of the file, between the parentheses, reference the global `window` and `jQuery` variables and pass them as *arguments* to our function. On top, add those arguments: `window` and `$`:

71 lines | web/assets/js/RepLogApp.js

```
1  (function(window, $) {
2      window.RepLogApp = {
... lines 3 - 69
70 })(window, jQuery);
```

Now, when we reference `window` and `$` in our code, we're no longer referencing the global objects directly, we're referencing those arguments.

Why the heck would you do this? There are two reasons, and neither are *huge*. First, you can alias global variables. At the bottom, we reference the `jQuery` global variable, which is even better than referencing `$` because sometimes people setup `jQuery` in no conflict mode, where it does *not* create a `$` variable. But then above, we alias this to `$`, meaning it's safe inside for us to use that shortcut. You probably don't have this

problem, but you'll see stuff like this in third-party libraries.

Second, when you pass in a global variable as an argument, it protects you from making a really silly mistake in your code, like accidentally setting `$ = null`. If you do that now, it'll set `$` to `null` only inside this function. But before, you would have overwritten that variable *globally*. It's yet another way that self-executing blocks help to sandbox us.

Fun with window

Ok, back to this mysterious `window` variable. Inside `index.html.twig`, `console.log()` `window`:

78 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65 {% block javascripts %}
... lines 66 - 69
70     <script>
71         console.log(window);
... lines 72 - 75
76     </script>
77 {% endblock %}
```

This is pretty cool, because it will show us *all* global variables that are available.

And Boom! This is a *huge* object, and includes the `$` variable, `jQuery`, and eventually, `RepLogApp`.

But notice what's *not* here. As expected, there is no `Helper`.

Forget var? It goes Global!

Now, go back into `RepLogApp`, find `Helper`, and remove the `var`:

71 lines | web/assets/js/RepLogApp.js

```
1 (function(window, $) {
... lines 2 - 55
56     Helper = {
... lines 57 - 68
69     };
70 })(window, jQuery);
```

You've probably been taught to *never* do this. And that's right! But you may not realize exactly what happens if you do.

Refresh again and open the `window` variable. Check this out! It's a little hard to find, but all of a sudden, there *is* a global `Helper` variable! So if you forget to say `var` - which you shouldn't - it makes that variable a global object, which means it's set on `window`.

There's one other curious thing about `window`: if you're in a global context where there is no `this` variable... then `this` is actually equal to `window`:

78 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
65 {% block javascripts %}
... lines 66 - 69
70     <script>
71         console.log(window === this);
... lines 72 - 75
76     </script>
77 {% endblock %}
```

If you refresh, this expression returns true. Oh JavaScript!

Be Better: use strict

Back in `RepLogApp`, forgetting `var` is actually a mistake, but JavaScript is friendly, and it allows us to make mistakes. In real life, friendly and forgiving people are great friends! In programming, friendly and forgiving languages mean more bugs!

To tell JavaScript to *stop* being such a pushover, at the top of the `RepLogApp.js` file, inside quotes, say `'use strict'`:

73 lines | web/assets/js/RepLogApp.js

```
1  'use strict';
2
3  (function(window, $) {
... lines 4 - 57
58     Helper = {
... lines 59 - 70
71     };
72 })(window, jQuery);
```

Tip

Even better! Put `'use strict'` inside the self-executing function. Adding `'use strict'` applies to the function its inside of and any functions inside of that (just like creating a variable with `var`). If you add it outside of a function (like we did), it affects the entire file. In this case, both locations are effectively identical. But, if you use a tool that concatenates your JS files into a single file, it's safer to place `'use strict'` inside the self-executing function, to ensure it doesn't affect those other concatenated files!

I know, weird. This is a special JavaScript directive that tells your browser to activate a more strict parsing mode. Now, certain things that *were* allowed before, will cause legit errors. And sure enough, when we refresh, we get:

> Uncaught reference: Helper is not defined

Sweeeet! Even PhpStorm isn't fooled anymore, it's reporting an:

> Unresolved variable or type Helper

Re-add `var`, and life is good!

73 lines | web/assets/js/RepLogApp.js

```
1  'use strict';
2
3  (function(window, $) {
   ... lines 4 - 57
58     var Helper = {
   ... lines 59 - 70
71     };
72 })(window, jQuery);
```

Chapter 14: Instantiatable Objects & Constructors

Ok ok, it's *finally* time to talk about the JavaScript elephant in the room: *prototypical inheritance*. This means, *real* JavaScript objects that we can *instantiate*!

But first, let's do just a *little* bit of reorganization on `Helper` - it'll make our next step easier to understand.

Instead of putting all of my functions directly inside my object immediately, I'll just say `var Helper = {}`:

73 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 54
    /**
     * A "private" object
     */
    var Helper = {};
... lines 59 - 71
})(window, jQuery);
```

Then set the `Helper.initialize` key to a function, and `Helper.calculateTotalWeight` equal to its function:

73 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 54
    /**
     * A "private" object
     */
    var Helper = {};

    Helper.initialize = function ($wrapper) {
... line 61
    };
    Helper.calculateTotalWeight = function() {
... lines 64 - 69
    };

})(window, jQuery);
```

This didn't change anything: it's just a different way of putting keys onto an object.

Everything is ~~Awesome~~ (an Object)!

Ok, in JavaScript, *everything* is an object, and this is quite different than PHP. Obviously, `Helper` is an object. But we already saw earlier that functions are *also* objects. This means when we say `this.handleRepLogDelete` - which references a function - we can call some method on it called `bind()`.

Heck, even *strings* are objects: we'll see that in a moment. The only downside with our `Helper` or `RepLogApp` objects so far is that they are effectively static.

The Goal: Non-Static Objects

Why? Because, there can only ever be *one* `Helper` object. If I had *two* areas on my page, and I wanted to calculate the total weight in each, we'd be in trouble! If we called `initialize()` a second time for the second area, it would *override* the original `$wrapper` property. It acts like a static object. And that's what we need to fix: I want to be able to *instantiate* objects... just like we do in PHP with the `new` keyword. This will let us create *two* Helper instances, each with their *own* `$wrapper` property.

Creating your Constructor

How do we do that? Instead of setting `Helper` to `{}`, set it to a function. Let's set `Helper` to what *was* our `initialize()` method:

71 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 54
    /**
     * A "private" object
     */
    var Helper = function ($wrapper) {
        this.$wrapper = $wrapper;
    };
... lines 61 - 69
})(window, jQuery);
```

Huh. So now, `Helper` is a *function*... But remember that functions are objects, so it's *totally* valid to add properties or methods to it.

Why would set our object to a function? Because now we are allowed to say `this.helper = new Helper($wrapper)`:

71 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
            this.$wrapper = $wrapper;
            this.helper = new Helper(this.$wrapper);
... lines 8 - 16
        },
... lines 18 - 52
    };
... lines 54 - 69
})(window, jQuery);
```

JavaScript *does* have the new keyword just like PHP! And you can use it once `Helper` is actually a function. This returns a new *instance* of `Helper`, which we set on a property.

In PHP, when you say `new Helper()`, PHP calls the *constructor* on your object, if you have one. The same happens here, the function *is* the constructor. At this point, we could create *multiple* Helper instances, each with their *own* `$wrapper`.

Now, instead of using `Helper` in a static kind of way, we use its instance: `this.helper`:

71 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
... lines 5 - 17
        updateTotalWeightLifted: function () {
            this.$wrapper.find('.js-total-weight').html(
                this.helper.calculateTotalWeight()
            );
        },
... lines 23 - 52
    };
... lines 54 - 69
})(window, jQuery);
```

Before we keep celebrating, let's try this. Go back, refresh, and delete one of our items! Huh, it worked... but the total didn't update. And, we have an error:

> Uncaught TypeError: this.helper.calculateTotalWeight is not a function

That's odd! Why does it think our Helper doesn't have that key? The answer is all about the prototype.

Chapter 15: The Object prototype!

In `RepLogApp`, when we try to call `this.helper.calculateTotalWeight`, for some reason, it doesn't think this is a function!

71 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3  (function(window, $) {
4      window.RepLogApp = {
... lines 5 - 17
18         updateTotalWeightLifted: function () {
19             this.$wrapper.find('.js-total-weight').html(
20                 this.helper.calculateTotalWeight()
21             );
22         },
... lines 23 - 52
53     };
... lines 54 - 69
70 })(window, jQuery);
```

But down below, we can plainly see: `calculateTotalWeight` *is* a function! What the heck is going on?

To find out, in `initialize`, let's log a few things: `console.log(this.helper)` and then `Object.keys(this.helper)`:

73 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3  (function(window, $) {
4      window.RepLogApp = {
5          initialize: function ($wrapper) {
... line 6
7              this.helper = new Helper(this.$wrapper);
8              console.log(this.helper, Object.keys(this.helper));
... lines 9 - 18
19         },
... lines 20 - 54
55     };
... lines 56 - 71
72 })(window, jQuery);
```

The `Object.keys` method is an easy way to print the properties and methods *inside* an object.

Comparing the Helper object and new Helper instance

Do the same thing for `Helper` and `Object.keys(Helper)`:

73 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3  (function(window, $) {
4      window.RepLogApp = {
5          initialize: function ($wrapper) {
... line 6
7              this.helper = new Helper(this.$wrapper);
8              console.log(this.helper, Object.keys(this.helper));
9              console.log(Helper, Object.keys(Helper));
... lines 10 - 18
19         },
... lines 20 - 54
55     };
... lines 56 - 71
72 })(window, jQuery);
```

Let's look at what the difference is between our *instance* of the `Helper` object and the `Helper` object itself.

Ok, find your browser, refresh, and check this out! There's the `helper` *instance* object, but check out the *methods* and properties on it: it has `$wrapper`. Wait, so when we create a `new Helper()`, that *instance* object *does* have the `$wrapper` property... but somehow it does *not* have a `calculateTotalWeight` method!

That's why we're getting the error. The question is why? Below, where we printed the upper-case "H" `Helper` object, it prints out as a function, but in its keys, it *does* have one called `calculateTotalWeight`. Oooh, mystery!

This can be very confusing. So follow this next part closely and all the way to the end.

At this point, the `calculateTotalWeight` function is effectively still static. The only way that we can call that method is by saying `Helper.calculateTotalWeight` - by calling the method on the original, static object. We *cannot* call this method on the instantiated instance: we can't say `this.helper.calculateTotalWeight()`. It just doesn't work!

Introducing the Prototype

To fix this, instead of adding the method via `Helper.calculateTotalWeight`, we need to say `Helper.prototype.calculateTotalWeight`:

74 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 63
    Helper.prototype.calculateTotalWeight = function() {
... lines 65 - 70
    };

})(window, jQuery);
```

That weird little trick fixes everything. To test it easily, back up in `initialize()`, let's try calling `this.helper.calculateTotalWeight()`:

74 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... line 6
            this.helper = new Helper(this.$wrapper);
            console.log(this.helper, Object.keys(this.helper));
            console.log(Helper, Object.keys(Helper));
            console.log(this.helper.calculateTotalWeight());
... lines 11 - 19
        },
... lines 21 - 55
    };
... lines 57 - 72
})(window, jQuery);
```

This did not work before, but refresh! 157.5 - it works now!

The short explanation is that when you create objects that need to be instantiated, you need to add its properties and methods to this special `prototype` key.

Once you've done that and create a `new Helper`, magically, anything on the prototype, like `calculateTotalWeight`, becomes part of that object.

But, that superficial explanation is crap! Let's find out how this really works!

Chapter 16: prototype Versus __proto__

Suddenly, after adding `calculateTotalWeight` to some strange `prototype` key, we can call this method on any new *instance* of the `Helper` object. But go back to your browser and check out the first log. Huh, our helper instance *still* only has one key: `$wrapper`. I don't see `calculateTotalWeight` here... so how the heck is that working? I mean, I don't see the method we're calling!

Hello **proto**

Check out that `__proto__` property. Every object has a magic property called `__proto__`. And if you open it, *it* holds the `calculateTotalWeight` function. Here's the deal: when you call a method or access a property on an object, JavaScript first looks for it on the object itself. But if it doesn't find it there, it looks at the `__proto__` property to see if it exists on *that* object. If it does, JavaScript uses it. If it does not exist, it actually keeps going to the next `__proto__` property *inside* of the original `__proto__` and tries to look for it there. It repeats that until it gets to the top level. What you are seeing here is the top-level `__proto__` that *every* object shares. In other words, these methods and properties exist on *every* object in JavaScript.

Boy, if you think about it, this is a lot like class inheritance, where each `__proto__` acts like a class we extend. And this last `__proto__` is like some base class that *everything* extends.

proto and prototype?

Ok, so how does this relate to the `prototype` key in our code?

74 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 63
    Helper.prototype.calculateTotalWeight = function() {
... lines 65 - 70
    };

})(window, jQuery);
```

Whenever you use the `new` keyword, anything on the `prototype` key of that object becomes the `__proto__` of the newly instantiated object.

Ok, let's play with this!

Create a new variable called `playObject` set to an object with a `lift` key set to `stuff`:

80 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... lines 6 - 11
            var playObject = {
                lift: 'stuff'
            };
... lines 15 - 25
        },
... lines 27 - 61
    };
... lines 63 - 78
})(window, jQuery);
```

Next, say `playObject.__proto__.cat = 'meow'`:

80 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... lines 6 - 11
            var playObject = {
                lift: 'stuff'
            };
            playObject.__proto__.cat = 'meow';
... lines 16 - 25
        },
... lines 27 - 61
    };
... lines 63 - 78
})(window, jQuery);
```

You shouldn't *normally* access or set the `__proto__` property directly, but for playing around now, it's great. Finally, `console.log(playObject.lift)`, which we know will work, but also `playObject.cat`:

80 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... lines 6 - 11
            var playObject = {
                lift: 'stuff'
            };
            playObject.__proto__.cat = 'meow';
            console.log(playObject.lift, playObject.cat);
... lines 17 - 25
        },
... lines 27 - 61
    };
... lines 63 - 78
})(window, jQuery);
```

Ok, try it. Refresh! Hey, `stuff` and `meow`! That's the `__proto__` property in action!

Decomposing the String, Array and DateTime Object

And hey! Remember how I said that *everything* is an object in JavaScript, including strings and arrays? Yep, that means that they *also* have an `__proto__`. This time, `console.log('foo'.__proto__)` to see what methods and properties belong to a string object. I wonder what things I can call on an array? Let's find out: `[].__proto__`. And what about a `new Date()` object? Print its `__proto__` too:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... lines 6 - 7
            console.log(
                'foo'.__proto__,
                [].__proto__,
                (new Date()).__proto__
            );
... lines 13 - 21
        },
... lines 23 - 57
    };
... lines 59 - 74
})(window, jQuery);
```

Let's see what happens! Refresh! Nice! Each is a big list of things that we can call on

each type of object. Apparently strings have an `indexOf()` method, a `match()` method, `normalize()`, `search()`, `slice()` and a lot more. The Array has its own big list. If you have a `DateTime` instance, you'll be able to call `getHours()`, `getMilliseconds()` and `getMinutes()`, to name a few.

To compare, let's Google for "JavaScript string methods". Check out the W3Schools result. This basically gives you the exact same information we just found ourselves: these are the methods you can call on a string. The *cool* part is that we now understand how this works: these are all stored on the `__proto__` of each string object.

Creating Multiple Instances

The *whole* point of this new constructor and `prototype` setup is so that we could have multiple instances of our `Helper` object. The `prototype` is just the key to take advantage of it.

To prove it all works, add `var helper2 = new Helper()` and pass it a different `$wrapper`, like the footer on our page:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... line 6
            this.helper = new Helper(this.$wrapper);
            var helper2 = new Helper($('footer'));
... lines 9 - 21
        },
... lines 23 - 57
    };
... lines 59 - 74
})(window, jQuery);
```

Since the footer doesn't have any rows that have weight on it, this should return zero. Log that: `this.helper.calculateTotalWeight()` and `helper2.calculateTotalWeight()`:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = {
        initialize: function ($wrapper) {
... line 6
            this.helper = new Helper(this.$wrapper);
            var helper2 = new Helper($('footer'));
            console.log(
                this.helper.calculateTotalWeight(),
                helper2.calculateTotalWeight()
            );
... lines 13 - 21
        },
... lines 23 - 57
    };
... lines 59 - 74
})(window, jQuery);
```

Try that! Cool! 157.5 and of course, zero.

Here's the point of all of this: you *do* want to setup your objects so that they can be instantiated. And now we know how to do this. First, set your variable to a function: this will become the constructor:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 59
    /**
     * A "private" object
     */
    var Helper = function ($wrapper) {
... line 64
    };
... lines 66 - 74
})(window, jQuery);
```

And second, add any methods or properties you need under the `prototype` key:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 65
    Helper.prototype.calculateTotalWeight = function() {
... lines 67 - 72
    };

})(window, jQuery);
```

You *can* still add keys directly to `Helper`, and these are basically the equivalent of static methods: you can only call them by using the original object name, like `Helper.foo` or `Helper.bar`.

Let's keep going: we can organize all of this a bit better. And once we have, we'll be able to make `RepLogApp` object a proper, instantiatable object... with almost no work.

Chapter 17: Extending the Prototype

From now on, we'll pretty much be adding *everything* to the `prototype` key. But, it *does* get a little bit annoying to always need to say `Helper.prototype.something =` for every method:

76 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 65
    Helper.prototype.calculateTotalWeight = function() {
... lines 67 - 72
    };

})(window, jQuery);
```

No worries! We can shorten this with a shortcut that's similar to PHP's `array_merge()` function. Use `$.extend()` and pass it `Helper.prototype` and then a second object containing all of the properties you want to merge into that object. In other words, move our `calculateTotalWeight()` function into this and update it to be `calculateTotalWeight: function`:

73 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 60
    $.extend(Helper.prototype, {
        calculateTotalWeight: function() {
            var totalWeight = 0;
            this.$wrapper.find('tbody tr').each(function () {
                totalWeight += $(this).data('weight');
            });

            return totalWeight;
        }
    });

})(window, jQuery);
```

At the bottom, we don't need the semicolon anymore. If we had more properties, we'd add them right below `calculateTotalWeight`: no need to worry about writing `prototype` every time.

There's nothing special about `$.extend`, it's just a handy `array_merge`-esque function that we happen to have handy. You may see other functions from other libraries that do the same thing.

Making RepLogApp an Instantiatable Object

With this trick, it's *super* easy to make `RepLogApp` an instantiatable object. First, set `RepLogApp` itself to the former `initialize()` function. I'll un-indent everything and finish it with a semicolon:

74 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
        this.$wrapper = $wrapper;
        this.helper = new Helper(this.$wrapper);

        this.$wrapper.find('.js-delete-rep-log').on(
            'click',
            this.handleRepLogDelete.bind(this)
        );
        this.$wrapper.find('tbody tr').on(
            'click',
            this.handleRowClick.bind(this)
        );
    };
... lines 17 - 72
})(window, jQuery);
```

Constructor done!

Next, add `$.extend()` with `window.RepLogApp.prototype` and `{`. The existing keys fit right into this perfectly! Winning! At the end, add an extra `)`:

74 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 17
    $.extend(window.RepLogApp.prototype, {
        updateTotalWeightLifted: function () {
            this.$wrapper.find('.js-total-weight').html(
                this.helper.calculateTotalWeight()
            );
        },

        handleRepLogDelete: function (e) {
            e.preventDefault();

            var $link = $(e.currentTarget);

            $link.addClass('text-danger');
            $link.find('.fa')
                .removeClass('fa-trash')
                .addClass('fa-spinner')
                .addClass('fa-spin');

            var deleteUrl = $link.data('url');
            var $row = $link.closest('tr');
            var self = this;
            $.ajax({
                url: deleteUrl,
                method: 'DELETE',
                success: function () {
                    $row.fadeOut('normal', function () {
                        $(this).remove();
                        self.updateTotalWeightLifted();
                    });
                }
            });
        },

        handleRowClick: function () {
            console.log('row clicked!');
        }
    });
... lines 55 - 72
})(window, jQuery);
```

Yes! In our template, we *won't* use `RepLogApp` like this anymore. Instead, say

`var repLogApp = new RepLogApp($table)`:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 69
    <script>
        $(document).ready(function() {
            var $table = $('.js-rep-log-table');
            var repLogApp = new RepLogApp($table);
        });
    </script>
{% endblock %}
```

We won't call any methods on that new `repLogApp` variable, but we could if we wanted to. We could also create *multiple* `RepLogApp` objects if we had multiple tables on the page, or if we loaded a table via AJAX. Our JavaScript is starting to be awesome!

Chapter 18: AJAX Form Submit: The Lazy Way

I'm feeling pretty awesome about all our new skills. So let's turn to a new goal and some new challenges. Below the RepLog table, we have a very traditional form. When we fill it out, it submits to the server: no AJAX, no fanciness.

And no fun! Let's update this to submit via AJAX. Of course, that comes with a few other challenges, like needing to dynamically add a new row to the table afterwards.

AJAXify the Form

In general, there are two ways to AJAXify this form submit. First, there's the simple, traditional, easy, and lazy way! That is, we submit the form via AJAX and the server returns HTML. For example, if we forget to select an item to lift, the AJAX would return the form HTML with the error in it so we can render it on the page. Or, if it's successful, it would probably return the new `<tr>` HTML so we can put it into the table. This is easier... because you don't need to do *all* that much in JavaScript. But, this approach is also a bit outdated.

The second approach, the more modern approach, is to actually treat your backend like an API. This means that we'll only send JSON back and forth. But this also means that we'll need to do more work in JavaScript! Like, we need to actually build the new `<tr>` HTML row by hand from the JSON data!

Obviously, *that* is where we need to get to! But we'll start with the old-school way first, and then refactor to the modern approach as we learn more and more cool stuff.

Making $wrapper Wrap Everything

In both situations, step one is the same: we need attach a listener on submit of the form. Head over to our template:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 52
            {{ include('lift/_form.html.twig') }}
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 77
```

The form itself lives in another template that's included here: `_form.html.twig` inside `app/Resources/views/lift`:

22 lines | app/Resources/views/lift/_form.html.twig

```
{{ form_start(form, {
    'attr': {
        'class': 'form-inline',
        'novalidate': 'novalidate'
    }
}) }}
    {{ form_errors(form) }}

    {{ form_row(form.item, {
        'label': 'What did you lift?',
        'label_attr': {'class': 'sr-only'}
    }) }}

    {{ form_row(form.reps, {
        'label': 'How many times?',
        'label_attr': {'class': 'sr-only'},
        'attr': {'placeholder': 'How many times?'}
    }) }}

    <button type="submit" class="btn btn-primary">I Lifted it!</button>
{{ form_end(form) }}

```

This is a Symfony form, but all this fanciness ultimately renders a good, old-fashioned `form` tag. Give the form another class: `js-new-rep-log-form`:

22 lines | app/Resources/views/lift/_form.html.twig

```
1  {{ form_start(form, {
2      'attr': {
3          'class': 'form-inline js-new-rep-log-form',
4          'novalidate': 'novalidate'
5      }
6  }) }}
↕  ... lines 7 - 20
21 {{ form_end(form) }}
```

Copy that and head into `RepLogApp` so we can attach a new listener. But wait... there *is* one problem: the `$wrapper` is actually the `<table>` element:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7">
... lines 6 - 12
            <table class="table table-striped js-rep-log-table">
... lines 14 - 50
            </table>

            {{ include('lift/_form.html.twig') }}
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 77
```

And the form does *not* live inside of the `<table>`!

When you create little JavaScript applications like `RepLogApp`, you want the wrapper to be an element that goes around *everything* you need to manipulate.

Ok, no problem: let's move the `js-rep-log-table` class from the table itself to the `div` that surrounds *everything*:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7 js-rep-log-table">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 50
            </table>
... lines 52 - 53
        </div>
... lines 55 - 61
    </div>
{% endblock %}
... lines 64 - 77
```

Down below, I don't need to change anything here, but let's rename `$table` to `$wrapper` for clarity:

77 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 64
{% block javascripts %}
... lines 66 - 69
    <script>
        $(document).ready(function() {
            var $wrapper = $('.js-rep-log-table');
            var repLogApp = new RepLogApp($wrapper);
        });
    </script>
{% endblock %}
```

The Form Submit Listener

Now adding our listener is simple: `this.$wrapper.find()` and look for `.js-new-rep-log-form`. Then, `.on('submit')`, have this call a new method: `this.handleNewFormSubmit`. And don't forget the all-important `.bind(this)`:

83 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 15
        this.$wrapper.find('.js-new-rep-log-form').on(
            'submit',
            this.handleNewFormSubmit.bind(this)
        );
    };
... lines 21 - 81
})(window, jQuery);
```

Down below, add that function - `handleNewFormSubmit` - and give it the event argument. This time, calling `e.preventDefault()` will prevent the form from *actually* submitting, which is good. For now, just `console.log('submitting')`:

83 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
            e.preventDefault();
            console.log('submitting!');
        }
    });
... lines 64 - 81
})(window, jQuery);
```

Ok, test time! Head back, refresh, and try the form. Yes! We get the log, but the form doesn't submit.

Adding AJAX

Turning this form into an AJAX call will be really easy... because we already know that this form works if we submit it in the traditional way. So let's just literally send that *exact* same request, but via AJAX.

First, get the form with `$form = $(e.currentTarget)`:

89 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
            e.preventDefault();

            var $form = $(e.currentTarget);
... lines 63 - 67
        }
    });
... lines 70 - 87
})(window, jQuery);
```

Next, add `$.ajax()`, set the `url` to `$form.attr('action')` and the `method` to `POST`. For the `data`, use `$form.serialize()`:

89 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
            e.preventDefault();

            var $form = $(e.currentTarget);
            $.ajax({
                url: $form.attr('action'),
                method: 'POST',
                data: $form.serialize()
            });
        }
    });
... lines 70 - 87
})(window, jQuery);
```

That's a really lazy way to get all the values for all the fields in the form and put them in the exact format that the server is accustomed to seeing for a form submit.

That's already enough to work! Submit that form! Yea, you can see the AJAX calls in the console and web debug toolbar. Of course, we don't see any new rows until we manually refresh the page...

So that's where the real work starts: showing the validation errors on the form on error and dynamically inserting a new row on success. Let's do it!

Chapter 19: Old-School AJAX HTML

When we use AJAX to submit this form, there are two possible responses: one if there was a form validation error and one if the submit was successful.

If we have an error response, for now, we need to return the HTML for this form, but with the validation error and styling messages included in it.

In our project, find the `LiftController` in `src/AppBundle/Controller`. The `indexAction()` method is responsible for both initially rendering the form on page load, and for handling the form submit:

80 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 9
class LiftController extends BaseController
{
    /**
     * @Route("/lift", name="lift")
     */
    public function indexAction(Request $request)
    {
        $this->denyAccessUnlessGranted('IS_AUTHENTICATED_REMEMBERED');

        $form = $this->createForm(RepLogType::class);
        $form->handleRequest($request);

        if ($form->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $repLog = $form->getData();
            $repLog->setUser($this->getUser());

            $em->persist($repLog);
            $em->flush();

            $this->addFlash('notice', 'Reps crunched!');

            return $this->redirectToRoute('lift');
        }

        $repLogs = $this->getDoctrine()->getRepository('AppBundle:RepLog')
            ->findBy(array('user' => $this->getUser()))
        ;
        $totalWeight = 0;
        foreach ($repLogs as $repLog) {
            $totalWeight += $repLog->getTotalWeightLifted();
        }

        return $this->render('lift/index.html.twig', array(
            'form' => $form->createView(),
            'repLogs' => $repLogs,
            'leaderboard' => $this->getLeaders(),
            'totalWeight' => $totalWeight,
        ));
    }
... lines 50 - 80
```

If you're not too familiar with Symfony, don't worry. But, at the bottom, add an if statement: if this is an AJAX request, then - at this point - we know we've failed form

validation:

87 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 9
10 class LiftController extends BaseController
11 {
... lines 12 - 14
15     public function indexAction(Request $request)
16     {
... lines 17 - 37
38         $totalWeight = 0;
39         foreach ($repLogs as $repLog) {
40             $totalWeight += $repLog->getTotalWeightLifted();
41         }
42
43         // render just the form for AJAX, there is a validation error
44         if ($request->isXmlHttpRequest()) {
... lines 45 - 47
48         }
... lines 49 - 55
56     }
... lines 57 - 85
86 }
```

Instead of returning the entire HTML page - which you can see it's doing right now - let's render *just* the form HTML. Do that with `return $this->render('lift/_form.html.twig')` passing that a `form` variable set to `$form->createView()`:

87 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 9
class LiftController extends BaseController
{
... lines 12 - 14
    public function indexAction(Request $request)
    {
... lines 17 - 42
        // render just the form for AJAX, there is a validation error
        if ($request->isXmlHttpRequest()) {
            return $this->render('lift/_form.html.twig', [
                'form' => $form->createView()
            ]);
        }
... lines 49 - 55
    }
... lines 57 - 85
}
```

Remember, the `_form.html.twig` template is included from index, and holds *just* the form.

And just like that! When we submit, we *now* get that HTML fragment.

Adding AJAX Success

Back in `RepLogApp`, add a `success` key to the AJAX call with a `data` argument: that will be the HTML we want to put on the page:

92 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 62
            $.ajax({
... lines 64 - 66
                success: function(data) {
... line 68
                }
            });
        }
    });
... lines 73 - 90
})(window, jQuery);
```

We need to replace *all* of this form code. I'll surround the form with a new element and give it a `js-new-rep-log-form-wrapper` class:

79 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7 js-rep-log-table">
... lines 6 - 52
            <div class="js-new-rep-log-form-wrapper">
                {{ include('lift/_form.html.twig') }}
            </div>
        </div>
... lines 57 - 63
    </div>
{% endblock %}
... lines 66 - 79
```

Back in `success`, use `$form.closest()` to find that, then replace its HTML with `data`:

92 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 62
            $.ajax({
... lines 64 - 66
                success: function(data) {
                    $form.closest('.js-new-rep-log-form-wrapper').html(data);
                }
            });
        }
    });
... lines 73 - 90
})(window, jQuery);
```

Tip

We could have also used the `replaceWith()` jQuery function instead of targeting a parent element.

Sweet! Let's enjoy our work! Refresh and submit! Nice! But if I put 5 into the box and hit enter to submit a second time... it doesn't work!? What the heck? We'll fix that in a

minute.

Handling Form Success

What about when we *don't* fail validation? In that case, we'll want to dynamically add a new row to the table. In other words, the AJAX call should once again return an HTML fragment: this time for a single `<tr>` row: this row right here.

To do that, we need to isolate it into its own template. Copy it, delete it, and create a new template: `_repRow.html.twig`. Paste the contents here:

14 lines | app/Resources/views/lift/_repRow.html.twig

```
<tr data-weight="{{ repLog.totalWeightLifted }}">
    <td>{{ repLog.itemLabel|trans }}</td>
    <td>{{ repLog.reps }}</td>
    <td>{{ repLog.totalWeightLifted }}</td>
    <td>
        <a href="#"
          class="js-delete-rep-log"
          data-url="{{ path('rep_log_delete', {id: repLog.id}) }}"
        >
           <span class="fa fa-trash"></span>
        </a>
    </td>
</tr>

```

Back in the main template, include this: `lift/_repRow.html.twig`:

67 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7 js-rep-log-table">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 22
                {% for repLog in repLogs %}
                    {{ include('lift/_repRow.html.twig') }}
                {% else %}
... lines 26 - 28
                {% endfor %}
... lines 30 - 38
            </table>
... lines 40 - 43
        </div>
... lines 45 - 51
    </div>
{% endblock %}
... lines 54 - 67
```

Now that we've done this, we can render it directly in `LiftController`. We know that the form was submitted successfully if the code inside the `$form->isValid()` block is executed. Instead of redirecting to another page, if this is AJAX, then return `$this->render('lift/_repRow.html.twig')` and pass it the one variable it needs: `repLog` set to `repLog`:

97 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 10
class LiftController extends BaseController
{
... lines 13 - 15
    public function indexAction(Request $request)
    {
... lines 18 - 22
        if ($form->isValid()) {
... lines 24 - 28
            $em->flush();

            // return a blank form after success
            if ($request->isXmlHttpRequest()) {
                return $this->render('lift/_repRow.html.twig', [
                    'repLog' => $repLog
                ]);
            }
... lines 37 - 40
        }
... lines 42 - 65
    }
... lines 67 - 95
}
```

And just by doing that, when we submit successfully, our AJAX endpoint returns the new `<tr>`.

Distinguishing Between Success and Error

But, our JavaScript code is already confused! It thought the new `<tr>` code was the error response, and replaced the form with it. Lame! Our JavaScript code needs to be able to distinguish between a successful request and one that failed with validation errors.

There's a *perfectly* standard way of doing this... and I was being lazy until now! On error, we should *not* return a 200 status code, and that's what the `render()` function gives us by default. When you return a 200 status code, it activates jQuery's `success` handler.

Instead, we should return a 400 status code, or really, anything that starts with a 4. To do that, add `$html =` and then change `render()` to `renderView()`:

97 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 10
class LiftController extends BaseController
{
... lines 13 - 15
    public function indexAction(Request $request)
    {
... lines 18 - 50
        // render just the form for AJAX, there is a validation error
        if ($request->isXmlHttpRequest()) {
            $html = $this->renderView('lift/_form.html.twig', [
                'form' => $form->createView()
            ]);
... lines 56 - 57
        }
... lines 59 - 65
    }
... lines 67 - 95
}
```

This new method simply gives us the string HTML for the page. Next, return a `new Response` manually and pass it the content - `$html` - and status code - `400`:

97 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 10
11 class LiftController extends BaseController
12 {
... lines 13 - 15
16     public function indexAction(Request $request)
17     {
... lines 18 - 50
51         // render just the form for AJAX, there is a validation error
52         if ($request->isXmlHttpRequest()) {
53             $html = $this->renderView('lift/_form.html.twig', [
54                 'form' => $form->createView()
55             ]);
56
57             return new Response($html, 400);
58         }
... lines 59 - 65
66     }
... lines 67 - 95
96 }
```

As soon as we do that, the `success` function will *not* be called on errors. Instead, the `error` function will be called. For an `error` callback, the first argument is *not* the data from the response, it's a `jqXHR` object:

97 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 63
            $.ajax({
... lines 65 - 70
                error: function(jqXHR) {
... lines 72 - 73
                }
            });
        }
    });
... lines 78 - 95
})(window, jQuery);
```

That's fine, because the response content is stored on `jqXHR.responseText`:

97 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 63
            $.ajax({
... lines 65 - 70
                error: function(jqXHR) {
                    $form.closest('.js-new-rep-log-form-wrapper')
                        .html(jqXHR.responseText);
                }
            });
        }
    });
... lines 78 - 95
})(window, jQuery);
```

Now we can use the `success` function to add the new `tr` to the table. Before the AJAX call - to avoid any problems with the `this` variable - add `$tbody = this.$wrapper.find('tbody')`:

97 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 62
            var $tbody = this.$wrapper.find('tbody');
            $.ajax({
... lines 65 - 74
            });
        }
    });
... lines 78 - 95
})(window, jQuery);
```

And in `success`, add `$tbody.append(data)`:

97 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 62
            var $tbody = this.$wrapper.find('tbody');
            $.ajax({
... lines 65 - 67
                success: function(data) {
                    $tbody.append(data);
                },
... lines 71 - 74
            });
        }
    });
... lines 78 - 95
})(window, jQuery);
```

That should do it!

Try it! Refresh the page! If we submit with errors, we get the errors! If we submit with something correct, a new row is added to the table. The only problem is that it doesn't update the *total* dynamically - that still requires a refresh.

But that's easy to fix! Above the AJAX call, add `var self = this`. And then inside `success`, call `self.updateTotalWeightLifted()`:

99 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 21
    $.extend(window.RepLogApp.prototype, {
... lines 23 - 58
        handleNewFormSubmit: function(e) {
... lines 60 - 63
            var self = this;
            $.ajax({
... lines 66 - 68
                success: function(data) {
                    $tbody.append(data);
                    self.updateTotalWeightLifted();
                },
... lines 73 - 76
            });
        }
    });
... lines 80 - 97
})(window, jQuery);
```

And now, it's all updating and working perfectly.

Except... if you try to submit the form twice in a row... it refreshes fully. It's like our JavaScript stops working after one submit. And you know what else? If you try to delete a row that was just added via JavaScript, *it* doesn't work either! Ok, let's find out why!

Chapter 20: Delegate Selectors FTW!

So dang. Each time we submit, it adds a new row to the table, but its delete button doesn't work until we refresh. What's going on here?

Well, let's think about it. In `RepLogApp`, the constructor function is called when we instantiate it. So, inside `$(document).ready()`:

67 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55 {% block javascripts %}
... lines 56 - 59
60     <script>
61         $(document).ready(function() {
62             var $wrapper = $('.js-rep-log-table');
63             var repLogApp = new RepLogApp($wrapper);
64         });
65     </script>
66 {% endblock %}
```

That means it's executed after the entire page has loaded.

Then, at *that* exact moment, our code finds all elements with a `js-delete-rep-log` class in the HTML, and attaches the listener to each DOM Element:

99 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3 (function(window, $) {
4     window.RepLogApp = function ($wrapper) {
... lines 5 - 7
8         this.$wrapper.find('.js-delete-rep-log').on(
9             'click',
10            this.handleRepLogDelete.bind(this)
11        );
... lines 12 - 19
20    };
... lines 21 - 97
98 })(window, jQuery);
```

So if we have 10 delete links on the page initially, it attaches this listener to those 10 individual DOM Elements. If we add a new `js-delete-rep-log` element later, there will be no listener attached to it. So when we click delete, nothing happens! So, what's the fix?

If you're like me, you've probably fixed this in a really crappy way before. Back then,

after dynamically adding something to my page, I would manually try to attach whatever listeners it needed. This is SUPER error prone and annoying!

Your New Best Friend: Delegate Selectors

But there's a much, much, much better way. AND, it comes with a fancy name: a delegate selector. Here's the idea, instead of attaching the listener to DOM elements that might be dynamically added to the page later, attach the listener to an element that will *always* be on the page. In our case, we know that `this.$wrapper` will always be on the page.

Here's how it looks: instead of saying `this.$wrapper.find()`, use `this.$wrapper.on()` to attach the listener to the wrapper:

102 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 7
        this.$wrapper.on(
            'click',
... line 10
            this.handleRepLogDelete.bind(this)
        );
... lines 13 - 22
    };
... lines 24 - 100
})(window, jQuery);
```

Then, add an extra second argument, which is the selector for the element that you truly want to react to:

102 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 7
        this.$wrapper.on(
            'click',
            '.js-delete-rep-log',
            this.handleRepLogDelete.bind(this)
        );
... lines 13 - 22
    };
... lines 24 - 100
})(window, jQuery);
```

That's it! This works *exactly* the same as before. It just says:

> Whenever a click event bubbles up to `$wrapper`, please check to see if any elements inside of it with a `js-delete-rep-log` were also clicked. If they were, fire this function! And have a great day!

You know what else! When it calls `handleRepLogDelete`, the `e.currentTarget` is *still* the same as before: it will be the `js-delete-rep-log` link element. So all our code still works!

Ah, this is sweet! So let's use delegate selectors *everywhere*. Get rid of the `.find()` and add the selector as the second argument:

102 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 12
        this.$wrapper.on(
            'click',
            'tbody tr',
            this.handleRowClick.bind(this)
        );
        this.$wrapper.on(
            'submit',
            '.js-new-rep-log-form',
            this.handleNewFormSubmit.bind(this)
        );
    };
... lines 24 - 100
})(window, jQuery);
```

To make sure this isn't one big elaborate lie, head back and refresh! Add a new rep log to the page... and delete it! It works! And we can also submit the form again without refreshing!

So always use delegate selectors: they just make your life easy. And since we designed our `RepLogApp` object around a `$wrapper` element, there was *no* work to get this rocking.

Chapter 21: Proper JSON API Endpoint Setup

It's time to graduate from this old-school AJAX approach where the server sends us HTML, to one where the server sends us ice cream! I mean, JSON!

First, in `LiftController::indexAction()`, let's remove the two AJAX if statements from before: we won't use them anymore:

97 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 10
11 class LiftController extends BaseController
12 {
... lines 13 - 15
16     public function indexAction(Request $request)
17     {
... lines 18 - 22
23         if ($form->isValid()) {
... lines 24 - 30
31             // return a blank form after success
32             if ($request->isXmlHttpRequest()) {
33                 return $this->render('lift/_repRow.html.twig', [
34                     'repLog' => $repLog
35                 ]);
36             }
... lines 37 - 40
41         }
... lines 42 - 50
51         // render just the form for AJAX, there is a validation error
52         if ($request->isXmlHttpRequest()) {
53             $html = $this->renderView('lift/_form.html.twig', [
54                 'form' => $form->createView()
55             ]);
56
57             return new Response($html, 400);
58         }
... lines 59 - 65
66     }
... lines 67 - 95
96 }
```

In fact, we're not going to use this endpoint at all. So, close this file.

Next, head to your browser, refresh, and view the source. Find the `<form>` element and copy the entire thing. Then back in your editor, find `_form.html.twig` and completely replace this file with that:

29 lines | app/Resources/views/lift/_form.html.twig

```
<form class="form-inline js-new-rep-log-form" novalidate>
    <div class="form-group">
        <label class="sr-only control-label required" for="rep_log_item">
            What did you lift?
        </label>
        <select id="rep_log_item"
                name="rep_log[item]"
                required="required"
                class="form-control">
            <option value="" selected="selected">What did you lift?</option>
            <option value="cat">Cat</option>
            <option value="fat_cat">Big Fat Cat</option>
            <option value="laptop">My Laptop</option>
            <option value="coffee_cup">Coffee Cup</option>
        </select></div>

    <div class="form-group">
        <label class="sr-only control-label required" for="rep_log_reps">
            How many times?
        </label>
        <input type="number" id="rep_log_reps"
               name="rep_log[reps]" required="required"
               placeholder="How many times?"
               class="form-control"/>
    </div>

    <button type="submit" class="btn btn-primary">I Lifted it!</button>
</form>
```

Setting up our HTML Form

In short, we are *not* going to use the Symfony Form component to render the form. It's not because we *can't*, but this will give us a bit more transparency on how our form looks. If you like writing HTML forms by hand, then write your code like I just did. If you *are* using Symfony and like to have *it* do the work for you, awesome, use Symfony forms.

We need to make two adjustments. First, get rid of the CSRF `_token` field. Protecting your API against CSRF attacks is a little more complicated, and a topic for another time. Second, when you use the Symfony form component, it creates `name` attributes

that are namespaced. Simplify each `name` to just `item` and `reps`:

29 lines | app/Resources/views/lift/_form.html.twig

```
1  <form class="form-inline js-new-rep-log-form" novalidate>
2      <div class="form-group">
... lines 3 - 5
6          <select id="rep_log_item"
7                  name="item"
8                  required="required"
9                  class="form-control">
... lines 10 - 14
15         </select></div>
16
17     <div class="form-group">
... lines 18 - 20
21         <input type="number" id="rep_log_reps"
22                name="reps" required="required"
23                placeholder="How many times?"
24                class="form-control"/>
25     </div>
... lines 26 - 27
28 </form>
```

We're just making our life easier.

By the way, if you *did* want to use Symfony's form component to render the form, be sure to override the `getBlockPrefix()` method in your form class and return an empty string:

```
SomeFormClass extends AbstractType
{
    public function getBlockPrefix()
    {
        return '';
    }
}
```

That will tell the form to render simple names like this.

Checking out the Endpoint

Our goal is to send this data to a true API endpoint, get back JSON in the response, and start handling that.

In `src/AppBundle/Controller`, open another file: `RepLogController`. This contains a set of API endpoints for working with RepLogs: one endpoint returns a collection, another returns *one* RepLog, another deletes a RepLog, and one - `newRepLogAction()` - can be used to

create a new RepLog:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
14 class RepLogController extends BaseController
15 {
16     /**
17      * @Route("/reps", name="rep_log_list")
18      * @Method("GET")
19      */
20     public function getRepLogsAction()
21     {
... lines 22 - 33
34     }
35
36     /**
37      * @Route("/reps/{id}", name="rep_log_get")
38      * @Method("GET")
39      */
40     public function getRepLogAction(RepLog $repLog)
41     {
... lines 42 - 44
45     }
46
47     /**
48      * @Route("/reps/{id}", name="rep_log_delete")
49      * @Method("DELETE")
50      */
51     public function deleteRepLogAction(RepLog $repLog)
52     {
... lines 53 - 58
59     }
60
61     /**
62      * @Route("/reps", name="rep_log_new")
63      * @Method("POST")
64      */
65     public function newRepLogAction(Request $request)
66     {
... lines 67 - 101
102    }
103
104    /**
105     * Turns a RepLog into a RepLogApiModel for the API.
106     *
```

```
     * This could be moved into a service if it needed to be
     * re-used elsewhere.
     *
     * @param RepLog $repLog
     * @return RepLogApiModel
     */
    private function createRepLogApiModel(RepLog $repLog)
    {
... lines 115 - 128
    }
}
```

I want you to notice a few things. First, the server expects us to send it the data as JSON:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 9
use Symfony\Component\HttpFoundation\Request;
... line 11
use Symfony\Component\HttpKernel\Exception\BadRequestHttpException;

class RepLogController extends BaseController
{
... lines 16 - 64
    public function newRepLogAction(Request $request)
    {
... line 67
        $data = json_decode($request->getContent(), true);
        if ($data === null) {
            throw new BadRequestHttpException('Invalid JSON');
        }
... lines 72 - 101
    }
... lines 103 - 129
}
```

Next, if you *are* a Symfony user, you'll notice that I'm still handling the data through Symfony's form system like normal:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 5
use AppBundle\Entity\RepLog;
use AppBundle\Form\Type\RepLogType;
... lines 8 - 9
use Symfony\Component\HttpFoundation\Request;
... lines 11 - 13
class RepLogController extends BaseController
{
... lines 16 - 64
    public function newRepLogAction(Request $request)
    {
... lines 67 - 72
        $form = $this->createForm(RepLogType::class, null, [
            'csrf_protection' => false,
        ]);
        $form->submit($data);
        if (!$form->isValid()) {
... lines 78 - 82
        }

        /** @var RepLog $repLog */
        $repLog = $form->getData();
... lines 87 - 101
    }
... lines 103 - 129
}
```

If it fails form validation, we're returning a JSON collection of those errors:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
... lines 16 - 64
    public function newRepLogAction(Request $request)
    {
... lines 67 - 76
        if (!$form->isValid()) {
            $errors = $this->getErrorsFromForm($form);

            return $this->createApiResponse([
                'errors' => $errors
            ], 400);
        }
... lines 84 - 101
    }
... lines 103 - 129
}
```

The `createApiResponse()` method uses Symfony's serializer, which is a fancy way of returning JSON:

57 lines | src/AppBundle/Controller/BaseController.php

```
... lines 1 - 8
class BaseController extends Controller
{
    /**
     * @param mixed $data Usually an object you want to serialize
     * @param int $statusCode
     * @return JsonResponse
     */
    protected function createApiResponse($data, $statusCode = 200)
    {
        $json = $this->get('serializer')
            ->serialize($data, 'json');

        return new JsonResponse($json, $statusCode, [], true);
    }
... lines 23 - 56
}
```

On success, it does the same thing: returns JSON containing the new RepLog's data:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
... lines 16 - 64
    public function newRepLogAction(Request $request)
    {
... lines 67 - 91
        $apiModel = $this->createRepLogApiModel($repLog);

        $response = $this->createApiResponse($apiModel);
... lines 95 - 101
    }
... lines 103 - 129
}
```

We'll see *exactly* what it looks like in a second.

Updating the AJAX Call

Ok! Let's update our AJAX call to go to *this* endpoint. In `RepLogApp`, down in `handleNewFormSubmit`, we need to somehow get that URL:

102 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 67
            $.ajax({
                url: $form.attr('action'),
... lines 70 - 79
            });
        }
    });
... lines 83 - 100
})(window, jQuery);
```

No problem! Find the form and add a fancy new `data-url` attribute set to `path()`, then the name of that route: `rep_log_new`:

29 lines | app/Resources/views/lift/_form.html.twig

```
<form class="form-inline js-new-rep-log-form" novalidate data-url="{{ path('rep_log_new') }}"
... lines 2 - 27
</form>
```

Bam! Now, back in `RepLogApp`, before we use that, let's clear out *all* the code that actually updates our DOM: all the stuff related to updating the form with the form errors or adding the new row. That's all a todo for later:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
            e.preventDefault();

            var $form = $(e.currentTarget);
... lines 66 - 69
            $.ajax({
... lines 71 - 81
            });
        }
    });
... lines 85 - 102
})(window, jQuery);
```

But, *do* add a `console.log('success')` and `console.log('error')` so we can see if this stuff is working!

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
            e.preventDefault();

            var $form = $(e.currentTarget);
... lines 66 - 69
            $.ajax({
... lines 71 - 73
                success: function(data) {
                    // todo
                    console.log('success!');
                },
                error: function(jqXHR) {
                    // todo
                    console.log('error :(');
                }
            });
        }
    });
... lines 85 - 102
})(window, jQuery);
```

Finally, update the `url` to `$form.data('url')`:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 69
            $.ajax({
                url: $form.data('url'),
... lines 72 - 81
            });
        }
    });
... lines 85 - 102
})(window, jQuery);
```

Next, our `data` format needs to change - I'll show you exactly how.

Chapter 22: POSTing to the API Endpoint

Before we keep going, I want to go back and look at what it *used* to look like when we submitted the form. I have *not* refreshed yet, and this AJAX call is an example of what the POST request looked like using our *old* code.

Click that AJAX call and move to the "Headers" tab. When we sent the AJAX call before, what did our request look like? At the bottom, you'll see "Form Data". But more interestingly, if you click "View Source", it shows you the raw request body that we sent. It's this weird-looking, almost query-string format, with `&` and `=` between fields.

This is the *traditional* form submit format for the web, a data format called `application/x-www-form-urlencoded`, if you want to get dorky about it. When you submit a normal HTML form, the data is sent like this. In PHP, that data is parsed into the familiar `$_POST` variable. We don't realize that it originally looked like this, because PHP gives us that nice associative array.

I wanted to show this because we are *not* going to send data in this format. Remember, our endpoint expects pure JSON. So `$form.serialize()` is not going to work anymore.

Instead, above the AJAX call, create a new `formData` variable set to an associative array, or an object:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 64
            var $form = $(e.currentTarget);
            var formData = {};
... lines 67 - 82
        }
    });
... lines 85 - 102
})(window, jQuery);
```

Next, use `$.each($form.serializeArray())`:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 65
            var formData = {};
            $.each($form.serializeArray(), function(key, fieldData) {
... line 68
            });
... lines 70 - 82
        }
    });
... lines 85 - 102
})(window, jQuery);
```

If you Google for that function - jQuery `serializeArray()` - you'll see that it finds all the fields in a form and returns a big array with keys `name` and `value` for each field.

This is not exactly what we want: we want an array where the `name` is the array key and that field's value is its value. No problem, because we can loop over this and turn it into that format. Add a function with `key` and `fieldData` arguments. Then inside, simply say, `formData[fieldData.name] = fieldData.value`:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 65
            var formData = {};
            $.each($form.serializeArray(), function(key, fieldData) {
                formData[fieldData.name] = fieldData.value
            });
... lines 70 - 82
        }
    });
... lines 85 - 102
})(window, jQuery);
```

Now that `formData` has the right format, turn it into JSON with `JSON.stringify(formData)`:

104 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 65
            var formData = {};
            $.each($form.serializeArray(), function(key, fieldData) {
                formData[fieldData.name] = fieldData.value
            });
            $.ajax({
... lines 71 - 72
                data: JSON.stringify(formData),
... lines 74 - 81
            });
        }
    });
... lines 85 - 102
})(window, jQuery);
```

Remember, we're doing this because that's what our endpoint expects: it will `json_decode()` the request body.

Ok, moment of truth. Refresh! Let's lift our laptop 10 times. Submit! Of course, nothing on the page changes, but we *do* have a successful POST request! Check out the response: `id`, `item`, `label`, `reps` and `totalWeightLifted`. Cool!

Also check out the "Headers" section again and find the request body at the bottom. It's now *pure* JSON: you can see the difference between our old request format and this new one.

Ok! It's time to get to work on our UI: we need to start processing the JSON response to add errors to our form and dynamically add a new row on success.

Chapter 23: Handling JSON Validation Errors

Our first goal is to read the JSON validation errors and add them visually to the form. A moment ago, when I filled out the form with no rep number, the endpoint sent back an error structure that looked like this: with an `errors` key and a key-value array of errors below that.

Parsing the Error JSON

To get this data, we need to parse the JSON manually with `var errorData = JSON.parse(jqXHR.responseText)`:

109 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 70
            $.ajax({
... lines 72 - 78
                error: function(jqXHR) {
                    var errorData = JSON.parse(jqXHR.responseText);
... line 81
                }
            });
        },
... lines 85 - 88
    });
... lines 90 - 107
})(window, jQuery);
```

That's the raw JSON that's sent back from the server.

To actually map the `errorData` onto our fields, let's create a new function below called `_mapErrorsToForm` with an `errorData` argument. To start, just log that:

109 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 85
        _mapErrorsToForm: function(errorData) {
            console.log(errorData);
        }
    });
... lines 90 - 107
})(window, jQuery);
```

Above, to call this, we know we can't use `this` because we're in a callback. So add the classic `var self = this;`, and *then* call `self._mapErrorsToForm(errorData.errors)`:

109 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 61
        handleNewFormSubmit: function(e) {
... lines 63 - 69
            var self = this;
            $.ajax({
... lines 72 - 78
                error: function(jqXHR) {
                    var errorData = JSON.parse(jqXHR.responseText);
                    self._mapErrorsToForm(errorData.errors);
                }
            });
        },
... lines 85 - 88
    });
... lines 90 - 107
})(window, jQuery);
```

All the important stuff is under the `errors` key, so we'll pass *just* that.

Ok, refresh that! Leave the form empty, and submit! Hey, beautiful error data!

Mapping Data into HTML

So how can we use this data to make actual HTML changes to the form? There are generally two different approaches. First, the simple way: parse the data by hand and manually use jQuery to add the necessary elements and classes. This is quick to do, but doesn't scale when things get really complex. The second way is to use a client-

side template. We'll do the simple way first, but then use a client-side template for a more complex problem later.

And actually, there's a third way: which is to use a full front-end framework like ReactJS. We'll save that for a future tutorial.

Creating a Selectors Map

In `_mapErrorsToForm`, let's look at the error data and use it to add an error `span` below that field. Obviously, we need to use jQuery to find our `.js-new-rep-log-form` form element.

But wait! Way up in our constructor, we're already referencing this selector:

109 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 17
        this.$wrapper.on(
... line 19
            '.js-new-rep-log-form',
... line 21
        );
    };

... lines 25 - 107
})(window, jQuery);
```

It's no big deal, but I would like to *not* duplicate that class name in multiple places. Instead, add an `_selectors` property to our object. Give it a `newRepForm` key that's set to its selector:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
        _selectors: {
            newRepForm: '.js-new-rep-log-form'
        },
... lines 29 - 109
    });
... lines 111 - 128
})(window, jQuery);
```

Now, reference that with `this._selectors.newRepForm`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
    window.RepLogApp = function ($wrapper) {
... lines 5 - 17
        this.$wrapper.on(
... line 19
            this._selectors.newRepForm,
... line 21
        );
    };
... lines 24 - 128
})(window, jQuery);
```

Below in our function, do the same:
`var $form = this.$wrapper.find(this._selectors.newRepForm)`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... line 91
            var $form = this.$wrapper.find(this._selectors.newRepForm);
... lines 93 - 108
        }
    });
... lines 111 - 128
})(window, jQuery);
```

Mapping the Data Manually

Now what? Simple: loop over every field see if that field's `name` is present in the `errorData`. And if it is, add an error message span element below the field. To find all the fields, use `$form.find(':input')` - that's jQuery magic to find all form elements. Then, `.each()` and pass it a callback function:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... line 91
            var $form = this.$wrapper.find(this._selectors.newRepForm);
... lines 93 - 95
            $form.find(':input').each(function() {
... lines 97 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

Inside, we know that `this` is actually the form element. So we can say `var fieldName = $(this).attr('name')`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
... lines 98 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

I'm also going to find the wrapper that's around the entire form field. What I mean is, each field is surrounded by a `.form-group` element. Since we're using Bootstrap, we also need to add a class to this. Find it with `var $wrapper = $(this).closest('.form-group')`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
                var $wrapper = $(this).closest('.form-group');
... lines 99 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

Perfect!

Then, if there is *not* any `data[fieldName]`, the field doesn't have an error. Just return:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
                var $wrapper = $(this).closest('.form-group');
                if (!errorData[fieldName]) {
                    // no error!
                    return;
                }
... lines 103 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

If there *is* an error, we need to add some HTML to the page. The easy way to do that is by creating a new jQuery element. Set `var $error` to `$()` and then the HTML you want: a span with a `js-field-error` class and a `help-block` class:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
                var $wrapper = $(this).closest('.form-group');
                if (!errorData[fieldName]) {
                    // no error!
                    return;
                }

                var $error = $('<span class="js-field-error help-block"></span>');
... lines 105 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

I left the span blank because it's cleaner to add the text on the next line: `$error.html(errorsData[fieldName])`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
                var $wrapper = $(this).closest('.form-group');
                if (!errorData[fieldName]) {
                    // no error!
                    return;
                }

                var $error = $('<span class="js-field-error help-block"></span>');
                $error.html(errorData[fieldName]);
... lines 106 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

This jQuery object is now done! But it's not on the page yet. Add it with `$wrapper.append($error)`. Also call `$wrapper.addClass('has-error')`:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
... lines 91 - 95
            $form.find(':input').each(function() {
                var fieldName = $(this).attr('name');
                var $wrapper = $(this).closest('.form-group');
                if (!errorData[fieldName]) {
                    // no error!
                    return;
                }

                var $error = $('<span class="js-field-error help-block"></span>');
                $error.html(errorData[fieldName]);
                $wrapper.append($error);
                $wrapper.addClass('has-error');
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

Yes! Let's try it! Refresh and submit! There it is!

The only problem is that, once I finally fill in the field, the error message stays! AND, I get a second error message! Man, we gotta get this thing cleaned up!

No problem: at the top, use `$form.find()` to find all the `.js-field-error` elements. And, remove those. Next, find all the `form-group` elements and remove the `has-error` class:

130 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 89
        _mapErrorsToForm: function(errorData) {
            // reset things!
            var $form = this.$wrapper.find(this._selectors.newRepForm);
            $form.find('.js-field-error').remove();
            $form.find('.form-group').removeClass('has-error');

            $form.find(':input').each(function() {
... lines 97 - 107
            });
        }
    });
... lines 111 - 128
})(window, jQuery);
```

Refresh now, and re-submit! Errors! Fill in one... beautiful!

And if we fill in *both* fields, the AJAX call is successful, but nothing updates. Time to tackle that.

Chapter 24: Clearing the Form, Prepping for a Template

Let's do the easy thing first: when we submit the form successfully, these errors need to disappear!

We already have code for that, so copy it, and isolate it into its own new method called `_removeFormErrors`:

140 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 107
        _removeFormErrors: function() {
            var $form = this.$wrapper.find(this._selectors.newRepForm);
            $form.find('.js-field-error').remove();
            $form.find('.form-group').removeClass('has-error');
        },
... lines 113 - 119
    });
... lines 121 - 138
})(window, jQuery);
```

Call that from our map function:

140 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 88
        _mapErrorsToForm: function(errorData) {
            this._removeFormErrors();
            var $form = this.$wrapper.find(this._selectors.newRepForm);
... lines 92 - 105
        },
... lines 107 - 119
    });
... lines 121 - 138
})(window, jQuery);
```

The *other* thing we should do is empty, or reset the fields after submit. Let's create another function that does that *and* removes the form's errors. Call it `_clearForm`. First call `this._removeFormErrors()`:

140 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 113
        _clearForm: function() {
            this._removeFormErrors();
... lines 116 - 118
        }
    });
... lines 121 - 138
})(window, jQuery);
```

To "reset" the form, get the DOM Element itself - there will be only one - by using `[0]` and calling `reset()` on it:

140 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 113
        _clearForm: function() {
            this._removeFormErrors();

            var $form = this.$wrapper.find(this._selectors.newRepForm);
            $form[0].reset();
        }
    });
... lines 121 - 138
})(window, jQuery);
```

I love that this `[0]` thing isn't a mystery anymore!

Call this from up in success: `self._clearForm()`:

140 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 65
        handleNewFormSubmit: function(e) {
... lines 67 - 74
            $.ajax({
... lines 76 - 78
                success: function(data) {
                    self._clearForm();
                },
... lines 82 - 85
            });
        },
... lines 88 - 119
    });
... lines 121 - 138
})(window, jQuery);
```

Ok, test this baby out! Submit it empty, then fill it out for real and submit. Boom!

Client-Side Templating??

Ok, back to the main task: on success, we need to add a new row to the table. We *could* do this the easy way: by manually parsing the JSON and building the table. But there's one big problem: I do *not* want to duplicate the row markup in Twig AND in JavaScript. Instead, we're going to use client-side templates.

Let's start off simple: at the bottom of our object, add a new function: `_addRow` that has a `repLog` argument. For now just log that: this will be the RepLog data that the AJAX call sends back:

145 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
        _addRow: function(repLog) {
            console.log(repLog);
        }
    });
... lines 126 - 143
})(window, jQuery);
```

Call this from up in the `success` callback: `self._addRow(data)`:

145 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 65
        handleNewFormSubmit: function(e) {
... lines 67 - 74
            $.ajax({
... lines 76 - 78
                success: function(data) {
                    self._clearForm();
                    self._addRow(data);
                },
... lines 83 - 86
            });
        },
... lines 89 - 124
    });
... lines 126 - 143
})(window, jQuery);
```

Let's make sure things are working so far: refresh and add a new element. Yes! The data has `id`, `itemLabel` and even a `links` key with a URL for this RepLog. We are ready to template!

In a nutshell, a client-side, or JavaScript templating engine is like having Twig, but in JavaScript. There are a lot of different JavaScript templating libraries, but they all work the same: write a template - a mixture of HTML and dynamic code - and then render it, passing in variables that are used inside. Again, it's *just* like using Twig... but in JavaScript!

One simple templating engine comes from a library called *Underscore.js*. This is basically a bunch of nice, utility functions for arrays, strings and other things. It also happens to have a templating engine.

Google for Underscore CDN so we can be lazy and include it externally. Copy the minified version and then go back and open `app/Resources/views/base.html.twig`. Add the new script tag at the bottom:

99 lines | app/Resources/views/base.html.twig

```
... lines 1 - 90
91  {% block javascripts %}
... lines 92 - 93
94      <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.8.3/underscore-min.js"><
95  {% endblock %}
... lines 96 - 99
```

Now, let's start templating!

Chapter 25: JavaScript Templating

Here's the goal: use a JavaScript template to render a new RepLog `<tr>` after we successfully submit the form. The first step is to, well, create the template - a big string with a mix of HTML and dynamic code. If you look at the Underscore.js docs, you'll see how their templates are supposed to look.

Now, we don't want to actually put our templates right inside JavaScript like they show, that would get messy fast. Instead, one great method is to add a new `script` tag with a special `type="text/template"` attribute. Give this an id, like `js-rep-log-row-template`, so we can find it later:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55  {% block javascripts %}
... lines 56 - 66
67      <script type="text/template" id="js-rep-log-row-template">
... lines 68 - 80
81      </script>
82  {% endblock %}
```

Tip

The `text/template` part doesn't do anything special at all: it's just a standard to indicate that what's inside is *not* actually JavaScript, but something else.

This is one of the few places where I use ids in my code. Inside, we basically want to duplicate the `_repRow.html.twig` template, but update it to be written for Underscore.js.

So temporarily, we are *totally* going to have duplication between our Twig, server-side template and our Underscore.js, client-side template. Copy all the `<tr>` code, then paste it into the new `script` tag.

Now, update things to use the Underscore.js templating format. So, `<%= totalWeightLifted %>`:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55 {% block javascripts %}
... lines 56 - 66
67     <script type="text/template" id="js-rep-log-row-template">
68         <tr data-weight="<%= totalWeightLifted %>">
... lines 69 - 79
80         </tr>
81     </script>
82 {% endblock %}
```

This is the print syntax, and I'm using a `totalWeightLifted` variable because eventually we're going to pass these keys to the template as variables: `totalWeightLifted`, `reps`, `id`, `itemLabel` and `links`.

Do the same thing to print out `itemLabel`. Keep going: the next line will be `reps`. And then use `totalWeightLifted` again... but make sure you use the right syntax!

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55 {% block javascripts %}
... lines 56 - 66
67     <script type="text/template" id="js-rep-log-row-template">
68         <tr data-weight="<%= totalWeightLifted %>">
69             <td><%= itemLabel %></td>
70             <td><%= reps %></td>
71             <td><%= totalWeightLifted %></td>
... lines 72 - 79
80         </tr>
81     </script>
82 {% endblock %}
```

But what about this `data-url`? We can't use the Twig `path` function anymore. But we *can* use this `links._self` key! That's supposed to be the link to where we can GET info about this RepLog, but because our API is well-built, it's also the URL to use for a DELETE request.

Great! Print out `<%= links._self %>`:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55 {% block javascripts %}
... lines 56 - 66
67     <script type="text/template" id="js-rep-log-row-template">
68         <tr data-weight="<%= totalWeightLifted %>">
69             <td><%= itemLabel %></td>
70             <td><%= reps %></td>
71             <td><%= totalWeightLifted %></td>
72             <td>
73                 <a href="#"
74                   class="js-delete-rep-log"
75                   data-url="<%= links._self %>"
76                 >
77                   <span class="fa fa-trash"></span>
78                 </a>
79             </td>
80         </tr>
81     </script>
82 {% endblock %}
```

Rendering the Template

Gosh, that's a nice template. Let's go use it! Find our `_addRow()` function. First, find the template text: `$('#js-rep-log-row-template').html()`:

151 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3 (function(window, $) {
... lines 4 - 24
25     $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
122         _addRow: function(repLog) {
123             var tplText = $('#js-rep-log-row-template').html();
... lines 124 - 129
130         }
131     });
... lines 132 - 149
150 })(window, jQuery);
```

Done! Our `script` tag trick is an easy way to store a template, but we could have also loaded it via AJAX. Winning!

Next, create a template object: `var tpl = _.template(tplText)`:

151 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
        _addRow: function(repLog) {
            var tplText = $('#js-rep-log-row-template').html();
            var tpl = _.template(tplText);
... lines 125 - 129
        }
    });
... lines 132 - 149
})(window, jQuery);
```

That doesn't render the template, it just *prepares* it. Oh, and like before, my editor doesn't know what `_` is... so I'll switch back to `base.html.twig`, press `option` + `enter` or `alt` + `enter`, and download that library. Much happier!

To finally render the template, add `var html = tpl(repLog)`, where `repLog` is an array of all of the variables that should be available in the template:

151 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
        _addRow: function(repLog) {
            var tplText = $('#js-rep-log-row-template').html();
            var tpl = _.template(tplText);

            var html = tpl(repLog);
... lines 127 - 129
        }
    });
... lines 132 - 149
})(window, jQuery);
```

Finally, celebrate by adding the new markup to the table: `this.$wrapper.find('tbody')` and then `.append($.parseHTML(html))`:

151 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
        _addRow: function(repLog) {
            var tplText = $('#js-rep-log-row-template').html();
            var tpl = _.template(tplText);

            var html = tpl(repLog);
            this.$wrapper.find('tbody').append($.parseHTML(html));
... lines 128 - 129
        }
    });
... lines 132 - 149
})(window, jQuery);
```

The `$.parseHTML()` function turns raw HTML into a jQuery object.

And since we have a new row, we also need to update the total weight. Easy! `this.updateTotalWeightLifted()`:

151 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $) {
... lines 4 - 24
    $.extend(window.RepLogApp.prototype, {
... lines 26 - 121
        _addRow: function(repLog) {
            var tplText = $('#js-rep-log-row-template').html();
            var tpl = _.template(tplText);

            var html = tpl(repLog);
            this.$wrapper.find('tbody').append($.parseHTML(html));

            this.updateTotalWeightLifted();
        }
    });
... lines 132 - 149
})(window, jQuery);
```

Deep breath. Let's give this a shot. Refresh the page. I think we should lift our coffee cup ten times to stay in shape. Bah, error! Oh, that was Ryan being lazy: our endpoint returns a `links` key, not `link`. Let's fix that:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 54
55  {% block javascripts %}
... lines 56 - 66
67      <script type="text/template" id="js-rep-log-row-template">
68          <tr data-weight="<%= totalWeightLifted %>">
... lines 69 - 71
72              <td>
73                  <a href="#"
74                    class="js-delete-rep-log"
75                    data-url="<%= links._self %>"
76                  >
... line 77
78                  </a>
79              </td>
80          </tr>
81      </script>
82  {% endblock %}
```

Ok, refresh and try it gain! This time, let's lift our coffee cup 20 times! It's alive!!!

If you watch closely, it's even updating the total weight at the bottom.

I love it! Except for the massive duplication: it's a real bummer to have the row template in two places. Let me show you one way to fix this.

Chapter 26: Full-JavaScript Rendering & FOSJsRoutingBundle

When you try to render *some* things on the server, but then also want to update them dynamically in JavaScript, you're going to run into our new problem: template duplication. There are *kind of* two ways to fix it. First, if you use Twig like I do, there is a library called twig.js for JavaScript. In theory, you can write *one* Twig template and then use it on your server, and *also* in JavaScript. I've done this before and know of other companies that do it also.

My only warning is to keep these shared templates very simple: render simple variables - like `categoryName` instead of `product.category.name` - and try to avoid using many filters, because some won't work in JavaScript. But if you keep your templates simple, it works great.

The second, and more universal way is to *stop* rendering things on your server. As soon as I decide I need a JavaScript template, the *only* true way to remove duplication is to remove the duplicated server-side template and render *everything* via JavaScript.

Inside of our object, add a new function called `loadRepLogs`:

162 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 31
        loadRepLogs: function() {
... lines 33 - 38
        },
... lines 40 - 141
    });
... lines 143 - 160
})(window, jQuery, Routing);
```

Call this from our constructor:

162 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
    window.RepLogApp = function ($wrapper) {
        this.$wrapper = $wrapper;
        this.helper = new Helper(this.$wrapper);

        this.loadRepLogs();
... lines 9 - 24
    };
... lines 26 - 160
})(window, jQuery, Routing);
```

Because here's the goal: when our object is created, I want to make an AJAX call to and endpoint that returns *all* of my current RepLogs. We'll then use that to build *all* of the rows by using our template.

I already created the endpoint: `/reps`:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
    /**
     * @Route("/reps", name="rep_log_list")
     * @Method("GET")
     */
    public function getRepLogsAction()
    {
        $repLogs = $this->getDoctrine()->getRepository('AppBundle:RepLog')
            ->findBy(array('user' => $this->getUser()))
        ;

        $models = [];
        foreach ($repLogs as $repLog) {
            $models[] = $this->createRepLogApiModel($repLog);
        }

        return $this->createApiResponse([
            'items' => $models
        ]);
    }
... lines 35 - 129
}
```

We'll look at exactly what this returns in a moment.

Getting the /reps URL

But first, the question is: how can we get this URL inside of JavaScript? I mean, we could hardcode it, but that should be your last option. Well, I can think of three ways:

1. We could add a `data-` attribute to something, like on the `$wrapper` element in `index.html.twig`.
2. We could pass the URL *into* our `RepLogApp` object via a second argument to the constructor, just like we're doing with `$wrapper`.
3. If you're in Symfony, you could cheat and use a cool library called FOSJsRoutingBundle.

Using FOSJsRoutingBundle

Google for that, and click the link on the Symfony.com documentation. This allows you to expose some of your URLs in JavaScript. Copy the composer require line, open up a new tab, paste that and hit enter:

```
$ composer require friendsofsymfony/jsrouting-bundle
```

While Jordi is wrapping our package with a bow, let's finish the install instructions. Copy the new bundle line, and add that to `app/AppKernel.php`:

57 lines | app/AppKernel.php

```
... lines 1 - 5
6  class AppKernel extends Kernel
7  {
8      public function registerBundles()
9      {
10         $bundles = [
... lines 11 - 21
22             new FOS\JsRoutingBundle\FOSJsRoutingBundle(),
... lines 23 - 24
25         ];
... lines 26 - 34
35     }
... lines 36 - 55
56 }
```

We also need to import some routes: paste this into `app/config/routing.yml`:

16 lines | app/config/routing.yml

```
... lines 1 - 13
fos_js_routing:
    resource: "@FOSJsRoutingBundle/Resources/config/routing/routing.xml"
```

Finally, we need to add two script tags to our page. Open `base.html.twig` and paste them at the bottom:

101 lines | app/Resources/views/base.html.twig

```
... lines 1 - 90
{% block javascripts %}
... lines 92 - 94
    <script src="{{ asset('bundles/fosjsrouting/js/router.js') }}"></script>
    <script src="{{ path('fos_js_routing_js', { callback: 'fos.Router.setData' }) }}"></script>
{% endblock %}
... lines 98 - 101
```

This bundle exposes a *global* variable called `Routing`. And you can use that `Routing` variable to generate links in the same way that we use the `path` function in Twig templates: just pass it the route name and parameters.

Check the install process. Ding!

Tip

If you have a JavaScript error where `Routing` is not defined, you may need to run:

```
$ php bin/console assets:install
```

Now, head to `RepLogController`. In order to make this route available to that `Routing` JavaScript variable, we need to add `options={"expose" = true}`:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
    /**
     * @Route("/reps", name="rep_log_list", options={"expose" = true})
... line 18
     */
    public function getRepLogsAction()
... lines 21 - 129
}
```

Back in `RepLogApp`, remember that this library gives us a *global* `Routing` object. And of course, inside of our self-executing function, we *do* have access to global variables. But

as a best practice, we prefer to *pass* ourselves any global variables that we end up using. So at the bottom, pass in the global `Routing` object, and then add `Routing` as an argument on top:

162 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 160
})(window, jQuery, Routing);
```

Making the AJAX Call

Back down in `loadRepLogs`, let's get to work: `$.ajax()`, and set the `url` to `Routing.generate()`, passing that the name of our route: `rep_log_list`. And on `success`, just dump that data:

Array

Ok, go check it out! Refresh! You can see the `GET` AJAX call made *immediately*. And adding a new row of course still works.

But look at the data sent back from the server: it has an `items` key with 24 entries. Inside, each has the *exact* same keys that the server sends us after creating a *new* RepLog. This is *huge*: these are all the variables we need to pass into our template!

Rendering All the Rows in JavaScript

In other words, we're ready to go! Back in `index.html.twig`, find the `<tbody>` and empty it entirely: we do *not* need to render this stuff on the server anymore:

76 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="row">
        <div class="col-md-7 js-rep-log-table">
... lines 6 - 12
            <table class="table table-striped">
... lines 14 - 21
                <tbody>
                </tbody>
... lines 24 - 31
            </table>
... lines 33 - 36
        </div>
... lines 38 - 44
    </div>
{% endblock %}
... lines 47 - 76
```

In fact, we can even delete our `_repRow.html.twig` template entirely!

Let's keep celebrating: inside of `LiftController` - which renders `index.html.twig` - we don't need to pass in the `repLogs` or `totalWeight` variables to Twig: these will be filled in via JavaScript. Delete the `totalWeight` variable from Twig:

71 lines | src/AppBundle/Controller/LiftController.php

```
... lines 1 - 10
class LiftController extends BaseController
{
... lines 13 - 15
    public function indexAction(Request $request)
    {
... lines 18 - 35
        return $this->render('lift/index.html.twig', array(
            'form' => $form->createView(),
            'leaderboard' => $this->getLeaders(),
        ));
    }
... lines 41 - 69
}
```

If you refresh the page now, we've got a totally empty table. Perfect. Back in `loadRepLogs`, use `$.each()` to loop over `data.items`. Give the function `key` and `repLog` arguments:

165 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 31
        loadRepLogs: function() {
... line 33
            $.ajax({
                url: Routing.generate('rep_log_list'),
                success: function(data) {
                    $.each(data.items, function(key, repLog) {
... line 38
                    });
                }
            });
        },
... lines 43 - 144
    });
... lines 146 - 163
})(window, jQuery, Routing);
```

Finally, above the AJAX call, add `var self = this`. And inside, say `self._addRow(repLog)`:

165 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 31
        loadRepLogs: function() {
            var self = this;
            $.ajax({
                url: Routing.generate('rep_log_list'),
                success: function(data) {
                    $.each(data.items, function(key, repLog) {
                        self._addRow(repLog);
                    });
                }
            });
        },
... lines 43 - 144
    });
... lines 146 - 163
})(window, jQuery, Routing);
```

And that should do it! Refresh the page! Slight delay... boom! All the rows load dynamically: we can delete them and add more. Mission accomplished!

Chapter 27: All About Promises!

Ok, let's talk promises: JavaScript promises. These are a *hugely* important concept in modern JavaScript, and if you haven't seen them yet, you will soon.

We all know that in JavaScript, a lot of things can happen asynchronously. For example, Ajax calls happen asynchronously and even fading out an element happens asynchronously: we call the `fadeOut()` function, but it doesn't finish until later. This is *so* common that JavaScript has created an interface to standardize how this is handled. If you understand how it works, you will have a huge advantage.

Hello Promise

Google for "JavaScript promise" and click into the Mozilla.org article. To handle asynchronous operations, JavaScript has an object called a Promise. Yep, it's literally an object in plain, normal JavaScript - there are no libraries being used. There *are* some browser compatibility issues, especially with Internet Explorer... like always... but it's easy to fix, and we'll talk about it later.

This article describes the two sides to a Promise. First, if *you* need to execute some asynchronous code and then notify someone later, then *you* will *create* a `Promise` object. That's basically what jQuery does internally when we tell it to execute an AJAX call. This isn't very common to do in *our* code, but we'll see an example later.

The second side is what *we* do all the time: this is when someone *else* is doing the asynchronous work for us, and we need to do something when it finishes. We're *already* doing stuff like this in at least 5 places in our code!

Promises Versus $.ajax

Whenever something asynchronous happen, there are two possible outcomes: either the asynchronous call finished successfully, or it failed. In Promise language, we say that the Promise was fulfilled or the Promise was rejected.

Here's the basic idea: if something happens asynchronously - like an AJAX call - that code should return a Promise object. If it does, we can call `.then()` on it, and pass it the function that should be executed when the operation finishes successfully.

Now that we know that, Google for "jQuery Ajax" to find the $.ajax() documentation. Check this out: normally when we call `$.ajax()`, we don't think about what this function *returns*. In fact, we're not assigning it to anything in our code.

But apparently, it returns something called a `jqXHR` object. Search for `jqXHR object` on this page - you'll find a header that talks about it. First, it gives a bunch of basic details about this object. Fine. But look below the code block:

> The jqXHR object implements the Promise interface, giving it all the properties, methods, and behavior of a Promise.

Woh! In other words, what we get back from `$.ajax()` is an object that has all the functionality of a `Promise`! An easy, and mostly-accurate way of thinking about this is: the `jqXHR` object is a sub-class of `Promise`.

Below, it shows you all of the different methods you can call on the `jqXHR` object. You can call `.done()`, which is an alternative to the `success` option, or `.fail()` as an alternative to the `failure` option. AND, check this out, you can call `.then()`, because `.then()` exists on the `Promise` object.

Adding Promise Handlers

In practice, this means we can call `.done()` on our `$.ajax()`. It'll receive the same `data` argument that's passed to `success`. Add a little `console.log('I am successful!')`. Let's also `console.log(data)`:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).done(function(data) {
                console.log('I am successful!');
                console.log(data);
... lines 104 - 106
            })
        },
... lines 109 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

And guess what? We can just chain *more* handlers off of this one: add another `.done()` that looks the same. Print a message - `another handler` - and also `console.log(data)` again:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).done(function(data) {
                console.log('I am successful!');
                console.log(data);
            }).done(function(data) {
                console.log('another handler!');
                console.log(data);
            })
        },
... lines 109 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

Using the Standard: *only* `.then()`

Effectively `$.ajax()` returns an object that has all the functionality of a `Promise` *plus* a few additional methods. The only methods that a *true* `Promise` has on it are `.then()` and `.catch()`, for when a promise is rejected, or fails. But jQuery's object also has `.always()`, `.fail()`, `.done()` and others that you can see inside what they call their "deferred object".

The story here is that jQuery implemented this functionality *before* the `Promise` object was a standard. You could use any of these methods, but instead, I want to focus on treating what we get back from jQuery as a *pure* `Promise` object. I want to pretend that these other methods don't exist, and only rely on `.then()` and `.catch()`:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).then(function(data) {
                console.log('I am successful!');
                console.log(data);
            }).then(function(data) {
                console.log('another handler!');
                console.log(data);
            })
        },
... lines 109 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

In other words, I'm saying:

> Don't rely on `.done()`, just use `.then()`, which is the method you would use with *any* other library that implements Promises.

Modifying the Value in .then

Ok, go back and refresh now. When we submit, both handlers are still called. But woh! Check this out: our first data prints out correctly... but the second one is undefined?

If you look back at the `Promise` documentation, this makes sense. It says:

> `.then()` appends a fulfillment handler on the `Promise` and returns a *new* `Promise` resolving to the return value of the called handler.

Ah, so when we add the second `.then()`, that's not being attached to the *original* `Promise`, that's being attached to a new `Promise` that's returned from the first `.then()`. And according to the rules, the *value* for that new `Promise` is equal to whatever we return from the first.

Ok, so let's prove that's the case: `return data`:

173 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).then(function(data) {
                console.log('I am successful!');
                console.log(data);

                return data;
            }).then(function(data) {
                console.log('another handler!');
                console.log(data);
            })
        },
... lines 111 - 152
    });
... lines 154 - 171
})(window, jQuery, Routing);
```

Back in the browser, it works! Both handlers are passed the same `data`.

But what about handling failures? Oh, that's pretty crazy.

Chapter 28: Catching a Failed Promise

What about handling failures? As you can see in the `Promise` documentation, the `.then()` function has an optional second argument: a function that will be called on failure. In other words, we can go to the end of `.then()` and add a `function`. We know that the *value* passed to jQuery failures is the `jqXHR`. Let's `console.log('failed')` and also log `jqXHR.responseText`:

176 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).then(function(data) {
... lines 102 - 105
            }, function(jqXHR) {
                console.log('failed!');
                console.log(jqXHR.responseText);
            }).then(function(data) {
... lines 110 - 111
            })
        },
... lines 114 - 155
    });
... lines 157 - 174
})(window, jQuery, Routing);
```

Ok, refresh! Keep the form blank and submit. Ok cool! It *did* call our failure handler and it *did* print the `responseText` correctly.

Standardizing around .catch

The second way - and better way - to handle rejections, is to use the `.catch()` function. Both approaches are identical, but this is easier for me to understand. Instead of passing a second argument to `.then()`, close up that function and then call `.catch()`:

176 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).then(function(data) {
... lines 102 - 105
            }).catch(function(jqXHR) {
                console.log('failed!');
                console.log(jqXHR.responseText);
            }).then(function(data) {
... lines 110 - 111
            })
        },
... lines 114 - 155
    });
... lines 157 - 174
})(window, jQuery, Routing);
```

This will do the *exact* same thing as before.

Catch Recovers from Errors

But in both cases, something very weird happens: the second `.then()` success handler is being called. Wait, what? So the first `.then()` is being skipped, which makes sense, because the AJAX call failed. But after `.catch()`, the second `.then()` is being called. Why?

Here's the deal: `catch` is named `catch` for a reason: you really need to think about it in the same way as a `try-catch` block in PHP. It will catch the failed `Promise` above and return a new `Promise` that resolves successfully. That means that any handlers attached to it - like our second `.then()` - will execute as *if* everything was fine.

We're going to talk more about this, but obviously, this is *probably* not what we want. Instead, move the `.catch()` to the end:

176 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 79
        handleNewFormSubmit: function(e) {
... lines 81 - 100
            }).then(function(data) {
... lines 102 - 105
            }).then(function(data) {
... lines 107 - 108
            }).catch(function(jqXHR) {
                console.log('failed!');
                console.log(jqXHR.responseText);
            });
        },
... lines 114 - 155
    });
... lines 157 - 174
})(window, jQuery, Routing);
```

Now, the second `.then()` will only be executed if the first `.then()` is executed. The `.catch()` will catch any failed Promises - or errors - at the bottom. More on the error catching later.

Refresh now! Cool - *only* the `catch()` handler is running.

Refactoring Away from success

Ok, with our new `Promise` powers, let's refactor our `success` and `error` callbacks to modern and elegant, promises.

To do that, just copy our code from `success` into `.then()`:

161 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            $.ajax({
... lines 88 - 90
            }).then(function(data) {
                self._clearForm();
                self._addRow(data);
... lines 94 - 96
            });
        },
... lines 99 - 140
    });
... lines 142 - 159
})(window, jQuery, Routing);
```

I'm not worried about returning anything because we're not chaining our "then"s. Remove the second `.then()` and move the `error` callback code into `.catch()`:

161 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            $.ajax({
                url: $form.data('url'),
                method: 'POST',
                data: JSON.stringify(formData)
            }).then(function(data) {
                self._clearForm();
                self._addRow(data);
            }).catch(function(jqXHR) {
                var errorData = JSON.parse(jqXHR.responseText);
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 99 - 140
    });
... lines 142 - 159
})(window, jQuery, Routing);
```

With any luck, that will work *exactly* like before. Yea! The error looks good. And adding a new one works too.

Let's find our two other `$.ajax()` spots. Do the same thing there: Move the `success` function to `.then()`, and move the other `success` also to `.then()`:

161 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 31
        loadRepLogs: function() {
... line 33
            $.ajax({
                url: Routing.generate('rep_log_list'),
            }).then(function(data) {
                $.each(data.items, function(key, repLog) {
                    self._addRow(repLog);
                });
            })
        },
... lines 42 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 62
            $.ajax({
                url: deleteUrl,
                method: 'DELETE'
            }).then(function() {
                $row.fadeOut('normal', function () {
                    $(this).remove();
                    self.updateTotalWeightLifted();
                });
            })
        },
... lines 73 - 140
    });
... lines 142 - 159
})(window, jQuery, Routing);
```

Awesome!

Why is this Awesome for me?

One of the *big* advantages of Promises over adding `success` or `error` options is that you can refactor your asynchronous code into external functions. Let's try it: create a new function called, `_saveRepLog` with a `data` argument:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
... lines 98 - 102
        },
... lines 104 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

Now, move our AJAX code here, and return it. Set the `data` key to `JSON.stringify(data)`. And for the `url`, we can replace this with `Routing.generate('rep_log_new')`:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return $.ajax({
                url: Routing.generate('rep_log_new'),
                method: 'POST',
                data: JSON.stringify(data)
            });
        },
... lines 104 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

In the controller, make sure to expose that route to JavaScript:

131 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
... lines 16 - 60
    /**
     * @Route("/reps", name="rep_log_new", options={"expose" = true})
... line 63
     */
    public function newRepLogAction(Request $request)
... lines 66 - 129
}
```

Here's the point: above, replace the AJAX call with simply `this._saveRepLog()` and pass it `formData`:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 85
            var self = this;
            this._saveRepLog(formData)
            .then(function(data) {
                self._clearForm();
                self._addRow(data);
            }).catch(function(jqXHR) {
                var errorData = JSON.parse(jqXHR.responseText);
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 96 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

Isolating asynchronous code like this wasn't possible before because, in *this* function, we couldn't add any success or failure options to the AJAX call. But now, since we know `_saveRepLog()` returns a `Promise`, and since we also know that Promises have `.then()` and `.catch()` methods, we're super dangerous. If we ever needed to save a RepLog from somewhere else in our code, we could call `_saveRepLog()` to do that... and even attach *new* handlers in that case.

Next, let's look at another mysterious behavior of `.catch()`.

Chapter 29: Promise catch: Catches Errors?

Yay! Let's complicate things!

Our AJAX call works really well, because when we make an AJAX call to create a new `RepLog`, our server returns all the data for that new `RepLog`. That means that when we call `.then()` on the AJAX promise, we have all the data we need to call `_addRow()` and get that new row inserted!

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
                self._clearForm();
                self._addRow(data);
... lines 91 - 93
            });
        },
... lines 96 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

Too easy: so let's make it harder!

Making our Endpoint Less Friendly

Pretend that we don't have full control over our API. And instead of returning the `RepLog` data from the create endpoint - which is what this line does - it returns an empty response:

132 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
class RepLogController extends BaseController
{
... lines 16 - 64
    public function newRepLogAction(Request $request)
    {
... lines 67 - 93
        //$response = $this->createApiResponse($apiModel);
        $response = new Response(null, 204);
... lines 96 - 102
    }
... lines 104 - 130
}
```

Passing `null` means *no* response content, and 204 is just a different status code used for empty responses - that part doesn't make any difference.

Now head over and fill out the form successfully. Whoa!

Yep, it blew up - that's not too surprising: we get an error that says:

> `totalWeightLifted` is not defined.

And if you look closely, that's coming from `underscore.js`. This is almost definitely an error in our template. We pass the response data - which is now empty - into `._addRow()`:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 87
            .then(function(data) {
... line 89
                self._addRow(data);
... lines 91 - 93
            });
        },
... lines 96 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

And that eventually becomes the variables for the template:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 136
        _addRow: function(repLog) {
            var tplText = $('#js-rep-log-row-template').html();
            var tpl = _.template(tplText);

            var html = tpl(repLog);
            this.$wrapper.find('tbody').append($.parseHTML(html));

            this.updateTotalWeightLifted();
        }
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

An empty response means that *no* variables are being passed. Hence, `totalWeightLifted` is not defined.

But check this out: there's a *second* error:

> JSON Exception: unexpected token

A catch Catches Errors

This is coming from `RepLogApp.js`, line 94. Woh, it's coming from inside our `.catch()` handler:

166 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                var errorData = JSON.parse(jqXHR.responseText);
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 96 - 145
    });
... lines 147 - 164
})(window, jQuery, Routing);
```

Now, as we understand it, our `catch` should only be called when our `Promise` fails, in other words, when we have an AJAX error. But in this case, the server returns a 204 status code - that is a *successful* status code. So why is our `catch` being called?

Here's the deal: in reality, `.catch()` will be called if your `Promise` is rejected, *or* if a handler above it throws an error. Since our `.then()` calls `_addRow()` and *that* throws an exception, this ultimately triggers the `.catch()`. Again, this works *a lot* like the `try-catch` block in PHP!

Tip

There are some subtle cases when throwing an exception inside asynchronous code *won't* trigger your `.catch()`. The Mozilla Promise Docs discuss this!

Conditionally Handling in catch

So this complicates things a bit. Before, we assumed that the value passed to `.catch()` would *always* be the `jqXHR` object: that's what `jQuery` passes when its Promise is rejected. But now, we're realizing that it might *not* be that, because something *else* might fail.

Let's `console.log(jqXHR)`:

167 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                console.log(jqXHR);
... lines 93 - 94
            });
        },
... lines 97 - 146
    });
... lines 148 - 165
})(window, jQuery, Routing);
```

Ok, refresh and fill out our form. There it is! Thanks to the error, it logs a "ReferenceError".

We've just found out that `.catch()` will catch anything that went wrong... and that the value passed to your handler will depend on *what* went wrong. This means that, if you want, you can code for this: `if (jqXHR instanceof ReferenceError)`, then `console.log('wow!')`:

169 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                if (jqXHR instanceof ReferenceError) {
                    console.log('wow!');
                }
... lines 95 - 96
            });
        },
... lines 99 - 148
    });
... lines 150 - 167
})(window, jQuery, Routing);
```

Let's see if that hits! Refresh, lift some laptops and, there it is!

What JavaScript *doesn't* have is the ability to do more intelligent try-catch block, where you catch only *certain* types of errors. Instead, `.catch()` handles *all* errors, but then, you can write *your* code to be a bit smarter.

Since we *really* only want to catch `jqXHR` errors, we could check to see if the `jqXHR` value is what we're expecting. One way is to check if `jqXHR.responseText === 'undefined'`. If this *is* undefined, this is not the error we intended to handle. To *not* handle it, and make that error uncaught, just `throw jqXHR`:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                if (typeof jqXHR.responseText === 'undefined') {
                    throw jqXHR;
                }
... lines 95 - 98
            });
        },
... lines 101 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

Now, if you wanted to, you could add another `.catch()` on the bottom, and inside its function, log the `e` value:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                if (typeof jqXHR.responseText === 'undefined') {
                    throw jqXHR;
                }
... lines 95 - 96
            }).catch(function(e) {
                console.log(e);
            });
        },
... lines 101 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

You see, because the first `catch` throws the error, the second one will catch it.

And when we try it now, the error prints *two* times - jQuery's Promise logs a warning each time an error is thrown inside a Promise. And then at the bottom, there's our log.

Let's remove the second `.catch()` *and* the `if` statement:

168 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                var errorData = JSON.parse(jqXHR.responseText);
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 96 - 147
    });
... lines 149 - 166
})(window, jQuery, Routing);
```

Why? Well, I'm not going to code defensively unless I'm coding against a situation that might possibly happen. In this case, it was developer error: my code just isn't written correctly for the server. Instead of trying to code around that, we just need to fix things!

We do the same thing in PHP: most of the time, we let exceptions happen... because it means we messed up!

Ok, we understand more about `.catch()`, but we still need to fix this whole situation! To do that, we'll need to create our *own* Promise.

Chapter 30: Making (and Keeping) a Promise

Ignore the error for a second and go down to the AJAX call. We know that this method returns a `Promise`, and then we call `.then()` on it:

171 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 98
            });
        },

        _saveRepLog: function(data) {
            return $.ajax({
                url: Routing.generate('rep_log_new'),
                method: 'POST',
                data: JSON.stringify(data)
            });
        },
... lines 109 - 150
    });
... lines 152 - 169
})(window, jQuery, Routing);
```

But, our handler expects that the Promise's *value* will be the `RepLog` data. But now, it's `null` because that's what the server is returning!

132 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
14 class RepLogController extends BaseController
15 {
... lines 16 - 64
65     public function newRepLogAction(Request $request)
66     {
... lines 67 - 93
94         //$response = $this->createApiResponse($apiModel);
95         $response = new Response(null, 204);
... lines 96 - 102
103     }
... lines 104 - 130
131 }
```

Somehow, I want to fix this method so that it *once again* returns a Promise whose value is the `RepLog` data.

How? Well first, we're going to read the `Location` header that's sent back in the response - which is the URL we can use to fetch that RepLog's data:

132 lines | src/AppBundle/Controller/RepLogController.php

```
... lines 1 - 13
14 class RepLogController extends BaseController
15 {
... lines 16 - 64
65     public function newRepLogAction(Request $request)
66     {
... lines 67 - 95
96         // setting the Location header... it's a best-practice
97         $response->headers->set(
98             'Location',
99             $this->generateUrl('rep_log_get', ['id' => $repLog->getId()])
100         );
... lines 101 - 102
103     }
... lines 104 - 130
131 }
```

We'll use that to make a *second* AJAX call to get the data we need.

Making the Second AJAX Call

Start simple: add another `.then()` to this, with 3 arguments: `data`, `textStatus` and `jqXHR`:

168 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return $.ajax({
... lines 99 - 101
            }).then(function(data, textStatus, jqXHR) {
... line 103
            });
        },
... lines 106 - 147
    });
... lines 149 - 166
})(window, jQuery, Routing);
```

Normally, promise handlers are only passed 1 argument, but in this case jQuery cheats and passes us 3. To fetch the `Location` header, say `console.log(jqXHR.getResponseHeader('Location'))`:

168 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return $.ajax({
... lines 99 - 101
            }).then(function(data, textStatus, jqXHR) {
                console.log(jqXHR.getResponseHeader('Location'));
            });
        },
... lines 106 - 147
    });
... lines 149 - 166
})(window, jQuery, Routing);
```

Go see if that works: we still get the errors, but hey! It prints `/reps/76`! Cool! Let's make an AJAX call to that: copy the `jqXHR` line. Then, add our favorite `$.ajax()` and set the URL to that header. Add a `.then()` to *this* `Promise` with a `data` argument:

173 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return $.ajax({
... lines 99 - 101
            }).then(function(data, textStatus, jqXHR) {
                $.ajax({
                    url: jqXHR.getResponseHeader('Location')
                }).then(function(data) {
... lines 106 - 107
                });
            });
        },
... lines 111 - 152
    });
... lines 154 - 171
})(window, jQuery, Routing);
```

Finally, *this* should be the RepLog data.

To check things, add `console.log('now we are REALLY done')` and also `console.log(data)` to make sure it looks right:

173 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return $.ajax({
... lines 99 - 101
            }).then(function(data, textStatus, jqXHR) {
                $.ajax({
                    url: jqXHR.getResponseHeader('Location')
                }).then(function(data) {
                    console.log('now we are REALLY done');
                    console.log(data);
                });
            });
        },
... lines 111 - 152
    });
... lines 154 - 171
})(window, jQuery, Routing);
```

Ok, refresh and fill out the form. Ignore the errors, because there's our message and the *correct* data!

Ok, now we can just return this somehow, right? Wait, that's not going to work... When we return the main `$.ajax()`, that `Promise` is resolved - meaning finished - the moment that the *first* AJAX call is made. You can see that because the errors from the handlers happen first, and *then* the second AJAX call finishes.

Somehow, we need to return a `Promise` that isn't resolved until that *second* AJAX call finishes.

There are *two* ways to do this - we'll do the harder way... because it's a lot more interesting - but I'll mention the other way at the end.

Could we use a Promise?

What we need to do is create our *own* `Promise` object, and take control of exactly when it's resolved and what value is passed back.

If you look at the `Promise` documentation, you'll find an example of how to do this: `new Promise()` with one argument: a function that has `resolve` and `reject` arguments. I know, it looks a little weird.

Inside of that function, you'll put your asynchronous code. And as soon as it's done, you'll call the `resolve()` function and pass it whatever *value* should be passed to the handlers. If something goes wrong, call the `reject()` function. This is effectively what jQuery is doing right now inside of its `$.ajax()` function.

Browser Compatability!? Polyfill

There's one quick gotcha: not *all* browsers support the `Promise` object. But, no worries! Google for "JavaScript Promise polyfill CDN".

A polyfill is a library that gives you functionality that's normally only available in a newer version of your language, JavaScript in this case. PHP also has polyfills: small PHP libraries that backport newer PHP functionality.

This polyfill guarantees that the `Promise` object will exist in JavaScript. If it's already supported by the browser it uses that. But if *not*, it adds it.

Copy the `es6-promise.auto.min.js` path. In the next tutorial, we'll talk *all* about what that es6 part means. Next, go into `app/Resources/views/base.html.twig` and add a `script` tag with `src=""` and this path:

102 lines | app/Resources/views/base.html.twig

```
... lines 1 - 90
91  {% block javascripts %}
... lines 92 - 96
97      <script src="https://cdnjs.cloudflare.com/ajax/libs/es6-promise/4.0.5/es6-promise.auto.min.j
98  {% endblock %}
... lines 99 - 102
```

Now our `Promise` object is guaranteed!

Creating a Promise

In `_saveRepLog`, create and return a `new Promise`, passing it the 1 argument it needs: a function with `resolve` and `reject` arguments:

177 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
3   (function(window, $, Routing) {
... lines 4 - 26
27      $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
97          _saveRepLog: function(data) {
98              return new Promise(function(resolve, reject) {
... lines 99 - 112
113             });
114         },
... lines 115 - 156
157     });
... lines 158 - 175
176 })(window, jQuery, Routing);
```

Move *all* of our AJAX code inside:

177 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return new Promise(function(resolve, reject) {
                $.ajax({
                    url: Routing.generate('rep_log_new'),
                    method: 'POST',
                    data: JSON.stringify(data)
                }).then(function(data, textStatus, jqXHR) {
                    $.ajax({
                        url: jqXHR.getResponseHeader('Location')
                    }).then(function(data) {
... lines 107 - 108
                    });
... lines 110 - 111
                });
            });
        },
... lines 115 - 156
    });
... lines 158 - 175
})(window, jQuery, Routing);
```

Now, all we need to do is call `resolve()` when our asynchronous work is *finally* resolved. This happens after the *second* AJAX call. Great! Just call `resolve()` and pass it `data`:

177 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return new Promise(function(resolve, reject) {
                $.ajax({
                    url: Routing.generate('rep_log_new'),
                    method: 'POST',
                    data: JSON.stringify(data)
                }).then(function(data, textStatus, jqXHR) {
                    $.ajax({
                        url: jqXHR.getResponseHeader('Location')
                    }).then(function(data) {
                        // we're finally done!
                        resolve(data);
                    });
... lines 110 - 111
                });
            });
        },
... lines 115 - 156
    });
... lines 158 - 175
})(window, jQuery, Routing);
```

Finally, the `RepLog` data should once again be passed to the success handlers!

Go back now and refresh. Watch the total at the bottom: lift the big fat cat 10 times and... boom! The new row was added *and* the total was updated. It worked!

This is huge! Our `_saveRepLog` function *previously* returned a `jqXHR` object, which implements the `Promise` interface. Now, we've changed that to a *real* `Promise`, and our code that calls this function didn't need to change at all. The `.then()` and `.catch()` work exactly like before. Ultimately, before *and* after this change, `_saveRepLog()` returns a promise whose value is the `RepLog` data.

Handling the Reject

Of course, we also need to call `reject`, which should happen if the original AJAX call has a validation error. If you fill out the form blank now, we can see the 400 error, but it doesn't call our `.catch()` handler.

No problem: after `.then()`, add a `.catch()` to handle the AJAX failure. Inside that, call `reject()` and pass it `jqXHR`: the value that our other `.catch()` expects:

177 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 96
        _saveRepLog: function(data) {
            return new Promise(function(resolve, reject) {
                $.ajax({
                    url: Routing.generate('rep_log_new'),
                    method: 'POST',
                    data: JSON.stringify(data)
                }).then(function(data, textStatus, jqXHR) {
... lines 104 - 109
                }).catch(function(jqXHR) {
                    reject(jqXHR);
                });
            });
        },
... lines 115 - 156
    });
... lines 158 - 175
})(window, jQuery, Routing);
```

We *could* also add a `.catch()` to the second AJAX call, but this should never fail under normal circumstances, so I think that's overkill.

Refresh again! And try the form blank. Perfect! But, we can get a *little* bit fancier.

Chapter 31: Promise Chaining

Oh, but now we can get *even* cooler! The `.catch()` handler above reads the `responseText` off of the `jqXHR` object and uses its error data:

177 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(jqXHR) {
                var errorData = JSON.parse(jqXHR.responseText);
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 96 - 156
    });
... lines 158 - 175
})(window, jQuery, Routing);
```

If we want, we could simplify the code in the handler by doing that *before* we reject our Promise.

Controlling Resolved Values

Let me show you: copy the `errorData` line and move it down into the other `.catch()`. Now, when we call `reject()`, pass it this:

178 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 95
        _saveRepLog: function(data) {
            return new Promise(function(resolve, reject) {
                $.ajax({
... lines 99 - 108
                }).catch(function(jqXHR) {
                    var errorData = JSON.parse(jqXHR.responseText);

                    reject(errorData);
                });
            });
        },
... lines 116 - 157
    });
... lines 159 - 176
})(window, jQuery, Routing);
```

As soon as we do that, any `.catch()` handlers will be passed the nice, clean `errorData`:

178 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 77
        handleNewFormSubmit: function(e) {
... lines 79 - 86
            this._saveRepLog(formData)
            .then(function(data) {
... lines 89 - 90
            }).catch(function(errorData) {
                self._mapErrorsToForm(errorData.errors);
            });
        },
... lines 95 - 157
    });
... lines 159 - 176
})(window, jQuery, Routing);
```

We no longer need to worry about parsing the JSON.

Refresh! And submit the form. Yes! Now, if we ever need to call `_saveRepLog()` from

somewhere else, attaching a `.catch()` handler will be easier: we're passed the most relevant error data.

Creating your own `Promise` objects is not that common, but it's super powerful, giving you the ability to perform *multiple* asynchronous actions and allow other functions to do something once they *all* finish.

Returning a Promise from a Handler

Now, there *was* an easier way to do this. Sometimes, inside a handler - like `.then()`, you'll want to make *another* asynchronous action:

178 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 95
        _saveRepLog: function(data) {
            return new Promise(function(resolve, reject) {
                $.ajax({
... lines 99 - 101
                }).then(function(data, textStatus, jqXHR) {
                    $.ajax({
... lines 104 - 107
                    });
... lines 109 - 112
                });
            });
        },
... lines 116 - 157
    });
... lines 159 - 176
})(window, jQuery, Routing);
```

That's exactly what's happening in `_saveRepLog()`. In this case, you can actually return a `Promise` from your handler.

Here's a simpler version of how our code could have looked to solve this same problem. Well, simpler at least in terms of the number of lines.

The first `$.ajax()` returns a `Promise`, and we immediately attach a `.then()` listener to it. From inside of *that* `.then()`, we return *another* `Promise`. When you do this, any *other* chained handlers will not be called until *that* `Promise`, meaning, the second AJAX call, has completed.

Let me say it a different way. First, because we're chaining `.then()` onto the `$.ajax()`, the return value of `_saveRepLog()` is actually whatever the `.then()` function returns. And

what is that? Both `.then()` and `.catch()` always return a *Promise* object.

And, up until now, the *value* used by the `Promise` returned by `.then()` or `.catch()` would be whatever value the function inside returned. But! *If* that function returns a *Promise*, then effectively, *that* `Promise` is what is ultimately returned by `.then()` or `.catch()`.

Tip

Technically, `.then()` should return a new `Promise` that mimics that `Promise` returned by the function inside of it. But it's easier to imagine that it directly returns the `Promise` that was returned inside of it.

That's a long way of saying that *other* chained listeners, will wait until that internal `Promise` is resolved. In our example, it means that any `.then()` handlers attached to `_saveRepLog()` will wait until the *inner* AJAX call is finished. In fact, that's the whole point of Promises: to allow us to perform multiple asynchronous actions by chaining a few `.then()` calls, instead of doing the old, ugly, nested handler functions.

Phew! Ok! Let's move on to *one* last, real-world example of using a Promise: inside an external library.

Chapter 32: SweetAlert: Killing it with Promises

For our last trick, Google for a library called SweetAlert2. Very simply, this library give us sweet alert boxes, like this. And you can customize it in a lot of ways, like having a "Yes" and "Cancel" button.

We're going to use SweetAlert so that when we click the delete icon, an alert opens so the user can confirm the delete before we actually do it.

SweetAlert: Basic Usage

To get this installed, go to the CDN. Copy the JavaScript file first. This time, instead of putting this in our base layout, we'll add the JavaScript to *just* this page: `index.html.twig`. Add the `<script src="">` and paste:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 53
54  {% block javascripts %}
55      {{ parent() }}
... line 56
57      <script src="https://cdn.jsdelivr.net/sweetalert2/6.1.0/sweetalert2.min.js"></script>
... lines 58 - 81
82  {% endblock %}
```

This also comes with a CSS file: copy that too. Back in `index.html.twig`, override a block called `stylesheets` and add the `endblock`. Call `parent()` to include the normal stylesheets, and then add the link tag with this path:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 47
48  {% block stylesheets %}
49      {{ parent() }}
50
51      <link rel="stylesheet" href="https://cdn.jsdelivr.net/sweetalert2/6.1.0/sweetalert2.min.css" /
52  {% endblock %}
... lines 53 - 83
```

Perfect!

This library exposes a global `swal()` function. Copy the timer example - it's *somewhat* similar to what we want. Then, open `RepLogApp.js`. Remember, whenever we reference a global object, we like to pass it *into* our self-executing function. You don't need to do

this, but it's super hipster. Pass `swal` at the bottom and also `swal` on top:

194 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 192
})(window, jQuery, Routing, swal);
```

If you want some auto-completion on that library, you can of course select it and hit `option` + `enter` or `alt` + `enter` to tell PhpStorm to download it.

Down in `handleRepLogDelete`, here's the plan. First, we'll open the alert. And then, when the user clicks "OK", we'll run all of the code below that actually deletes the `RepLog`. To prepare for that, isolate all of that into its own new method: `_deleteRepLog` with a `$link` argument:

194 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 48
        handleRepLogDelete: function (e) {
            e.preventDefault();

            var $link = $(e.currentTarget);
... lines 53 - 67
        },

        _deleteRepLog: function($link) {
            $link.addClass('text-danger');
            $link.find('.fa')
                .removeClass('fa-trash')
                .addClass('fa-spinner')
                .addClass('fa-spin');

            var deleteUrl = $link.data('url');
            var $row = $link.closest('tr');
            var self = this;
            $.ajax({
                url: deleteUrl,
                method: 'DELETE'
            }).then(function() {
                $row.fadeOut('normal', function () {
                    $(this).remove();
                    self.updateTotalWeightLifted();
                });
            })
        },
... lines 90 - 192
})(window, jQuery, Routing, swal);
```

This doesn't change anything: we could still just call this function directly from above. But instead, paste the SweetAlert code and update the title - "Delete this log" - and the text - "Did you not actually lift this?". And remove the timer option. Instead, add `showCancelButton: true`:

194 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
            e.preventDefault();

            var $link = $(e.currentTarget);
... lines 53 - 54
            swal({
                title: 'Delete this log?',
                text: 'What? Did you not actually lift this?',
                showCancelButton: true
            }).then(
                function () {
... line 61
                },
                function () {
... line 64
                }
            );

        },
... lines 69 - 174
    });
... lines 176 - 192
})(window, jQuery, Routing, swal);
```

With *just* that, we should be able to refresh, and... oh! Error!

> swal is not defined

Of course! I need be more careful with my ordering. Right now, we still need to manually make sure that we include the libraries in the correct order: including SweetAlert first, so that it's available to `RepLogApp`:

83 lines | app/Resources/views/lift/index.html.twig

```
... lines 1 - 53
{% block javascripts %}
... lines 55 - 56
    <script src="https://cdn.jsdelivr.net/sweetalert2/6.1.0/sweetalert2.min.js"></script>
    <script src="{{ asset('assets/js/RepLogApp.js') }}"></script>
... lines 59 - 81
{% endblock %}
```

We're going to fix this pesky problem in a future tutorial.

Ok, try it again. Things look happy! Now, click the little trash icon. Boom! We have "OK" and "Cancel".

Handling a SweetAlert Promise

When we call `swal()`, guess what it returns? A promise! A freaking Promise! We can tell because the code has a `.then` chained to it, with two arguments. The first argument is the function that's called on success, and the second is called when the Promise is rejected. But, we already knew that.

Specifically, for SweetAlert, the success, or resolved handler is called if we click "OK", and the reject handler is called if we click "Cancel". Easy! Above the `swal()` call, add `var self = this`. Then, inside the success handler, use `self._deleteRepLog($link)`:

194 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 53
            var self = this;
            swal({
... lines 56 - 58
            }).then(
                function () {
                    self._deleteRepLog($link);
                },
                function () {
... line 64
                }
            );

        },
... lines 69 - 174
    });
... lines 176 - 192
})(window, jQuery, Routing, swal);
```

Down in the reject function, we don't need to do anything. Just call `console.log('canceled')`:

194 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 53
            var self = this;
            swal({
... lines 56 - 58
            }).then(
                function () {
                    self._deleteRepLog($link);
                },
                function () {
                    console.log('canceled');
                }
            );

        },
... lines 69 - 174
    });
... lines 176 - 192
})(window, jQuery, Routing, swal);
```

Let's try it! Refresh, click the trash icon and hit "Cancel". Yea, there's the log! Now hit "OK". It deletes it! Guys, this is why understanding promises is *so* important.

And we *also* know that instead of passing two arguments to `.then()`, we could instead chain a `.catch()` onto this:

190 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 53
            var self = this;
            swal({
... lines 56 - 58
            }).then(function () {
                self._deleteRepLog($link);
            }).catch(function(arg) {
                console.log('canceled', arg);
            });
        },
... lines 65 - 170
    });
... lines 172 - 188
})(window, jQuery, Routing, swal);
```

Chapter 33: Sweet Alert: Create a Promise!

And we also *also* know that both functions are passed a value, and what that value is depends on the library. Add an `arg` to `.catch()` and log it:

190 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 54
            swal({
... lines 56 - 60
            }).catch(function(arg) {
                console.log('canceled', arg);
            });
        },
... lines 65 - 170
    });
... lines 172 - 188
})(window, jQuery, Routing, swal);
```

Ok, refresh, hit delete and hit cancel. Oh, it's a string: "cancel". Try it again, but hit escape this time to close the alert. Now it's `esc`. Interesting! If you search for "Promise" on its docs, you'll find a spot called "Handling Dismissals". Ah, it basically says:

> When an alert is dismissed by the user, the reject function is passed one of these strings, documenting the reason it was dismissed.

That's pretty cool. And more importantly, it was easy for us to understand.

Kung fu by Creating another Promise

Because we understand Promises, there's one other really cool thing we can do. Search for `preConfirm`. If you pass a `preConfirm` option, then after the user clicks "Ok", but before SweetAlert closes, it will call your function. You can do anything inside... but if what you want to do is asynchronous, like an AJAX call, then you need to return a Promise from this function. This will tell SweetAlert when your work is done so that it knows when it's ok to close the alert.

Let's try it! First, add a `showLoaderOnConfirm` option set to `true`:

198 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 54
            swal({
                title: 'Delete this log?',
                text: 'What? Did you not actually lift this?',
                showCancelButton: true,
                showLoaderOnConfirm: true,
... lines 60 - 70
            });
        },
... lines 73 - 178
    });
... lines 180 - 196
})(window, jQuery, Routing, swal);
```

That will show a little loading icon after the user clicks "OK". Next, add the `preConfirm` option set to a function. Inside, return a `new Promise` with the familiar `resolve` and `reject` arguments:

198 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 54
            swal({
... lines 56 - 59
                preConfirm: function() {
                    return new Promise(function(resolve, reject) {
... lines 62 - 64
                    });
                }
... lines 67 - 70
            });
        },
... lines 73 - 178
    });
... lines 180 - 196
})(window, jQuery, Routing, swal);
```

Just to fake it, let's pretend we need to do some work before we can actually delete the `RepLog`, and that work will take about a second. Use `setTimeout()` to fake this: pass that a function and set it to wait for one second. After the second, we'll call `resolve()`:

198 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 54
            swal({
... lines 56 - 59
                preConfirm: function() {
                    return new Promise(function(resolve, reject) {
                        setTimeout(function() {
                            resolve();
                        }, 1000);
                    });
                }
... lines 67 - 70
            });
        },
... lines 73 - 178
    });
... lines 180 - 196
})(window, jQuery, Routing, swal);
```

Try it! Refresh and click delete. After I hit ok, you should see a loading icon for one second, before the alert finally closes. Do it! There it was! Viva promises!

More realistically, sometimes - instead of doing my work after the alert closes, I like to do my work, my AJAX call, inside of `preConfirm`. After all, SweetAlert shows the user a pretty fancy loading icon while they're waiting. Let's do that here - it's *super* easy!

Move the `self._deleteRepLog()` call up into the `preConfirm` function and return it. Then get rid of the `.then()` entirely:

193 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 48
        handleRepLogDelete: function (e) {
... lines 50 - 54
            swal({
... lines 56 - 59
                preConfirm: function() {
                    return self._deleteRepLog($link);
                }
            }).catch(function(arg) {
                // canceling is cool!
            });
        },
... lines 67 - 173
    });
... lines 175 - 191
})(window, jQuery, Routing, swal);
```

This is *totally* legal, as long as the `_deleteRepLog()` function returns a Promise. In other words, as long as we *return* `$.ajax()`, SweetAlert will be happy:

193 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 67
        _deleteRepLog: function($link) {
... lines 69 - 78
            return $.ajax({
... lines 80 - 86
            })
        },
... lines 89 - 173
    });
... lines 175 - 191
})(window, jQuery, Routing, swal);
```

We can still keep the catch here, because if you hit cancel, that will still reject the promise and call `.catch()`. Head back, refresh, and click delete. You should see the loading icon for *just* a moment, while our AJAX call finishes. Hit "Ok"! Beautiful!

Cleanup My Mistakes

Oh, and by the way, if you noticed that I was still using `.done()` in a few places, that was an accident! Let's change this to `.then()`, and do the same thing in `loadRepLogs`:

193 lines | web/assets/js/RepLogApp.js

```
... lines 1 - 2
(function(window, $, Routing, swal) {
... lines 4 - 26
    $.extend(window.RepLogApp.prototype, {
... lines 28 - 31
        loadRepLogs: function() {
... line 33
            $.ajax({
... line 35
            }).then(function(data) {
... lines 37 - 39
            })
        },
... lines 42 - 67
        _deleteRepLog: function($link) {
... lines 69 - 78
            return $.ajax({
... lines 80 - 81
            }).then(function() {
... lines 83 - 86
            })
        },
... lines 89 - 173
    });
... lines 175 - 191
})(window, jQuery, Routing, swal);
```

Now we're using the *true* Promise functions, not the `.done()` function that only lives in jQuery.

Woh, we're done! I hope you guys *thoroughly* enjoyed this weird dive into some of the neglected parts of JavaScript! In the next tutorial in this series, we're going to talk about ES6, a *new* version of JavaScript, which has a lot of new features and new syntaxes that you probably haven't seen yet. But, they're *critical* to writing modern JavaScript.

All right guys, see you next time.