Symfony UX: Turbo

Chapter 1: Turbo: Drive, Frames & Streams!

Hey friends! Welcome back for part two of our Symfony UX series. The whole point of this series is to take a traditional web app - so an app with Twig templates that return HTML - and learn to do two things with it.

First, how to write truly professional JavaScript that... always works... even if some HTML is loaded via Ajax. We covered this in the first tutorial about Stimulus.

The second goal of this series is all about how we can make our app feel like a single page application. What I mean is: how we can make our site *lightning* fast by *never* having *any* full page refreshes. *That* is what Turbo gives us.

The 3 Parts of Turbo

To be more precise, Turbo is actually *three* different parts.

The first is "Turbo Drive". It's what turns clicks and form submits into Ajax calls. This is what gives you that single page app experience.

The second part is "Turbo Frames", which allows you to separate your page into small sections that can load and navigate independently.

And the third part is "Turbo Streams", which allows you to update any HTML element that's currently showing on the page... from inside your Symfony app. Crazy, right? When you use Turbo Streams along with Mercure, this can even give you the ability to make a real time chat app... while writing *zero* JavaScript.

And you're free to use all three parts... or just one or two: they operate independently.

Is Turbo New?

Now Turbo itself is... sort of brand new. If you check out its GitHub page, it's version 7.0.0-beta.5 at the time of recording. So... why is it version 7 if it's so new? Because *one* part of turbo

- Turbo Drive, the part that turns link clicks and form submits into Ajax calls - has been around for *years*. It was previously called "Turbolinks" and you can still find helpful blog posts and StackOverflow answers if you search using that term.

But the other two parts - Turbo Frames and Turbo Streams *are* brand new. These are mostly already very good, but we *will* see a few rough edges and missing features along the way. But we won't let that stop us: Turbo's event system will give us the power to do almost anything.

Before we dive in, I also need to mention that Stimulus and Turbo were both affected by a serious situation at the company Basecamp. This has left both libraries without their lead developers. Am I worried? It's not ideal... but I'm not too worried. The community is large and some big companies use this technology. And at the very least, I'm confident that Turbo - or something very similar to Turbo - will be around for a long time to come. You can't stop a great idea. We're actively integrating Turbo into SymfonyCasts right now.

Project Setup

So let's do this! To "turbocharge" your learning experience you should code along with me! Hey - the puns probably won't get any better, so, settle in. Download the course code from this page. When you unzip it, you'll have a `start/` directory with the same code that you see here. Check out the, `README.md` file for all the setup details.

I'll go through just the last few steps. Open a terminal and move into the project. I'll use the Symfony binary to start a local web server with:

```
symfony serve -d
```

Before we go check that out, let's also make sure to run Webpack. Install the Node dependencies with:

```
yarn install
```

And... when that finishes, run Webpack with:

```
yarn watch
```

As soon as this builds... perfect - spin over to your browser and head to https://127.0.0.1:8000 to see... MVP Office Supplies! Our store for selling minimally viable office products to trendy startups. This is the same project as the first tutorial, though I did make some changes, like adding a review system below each product... and upgrading some libraries.

Now that we have this running, let's install Turbo and activate Turbo Drive to instantly eliminate full page refreshes. Woh.

Chapter 2: Installing Turbo

Wouldn't it be cool if when we click on a link or even submit a form, instead of that triggering a full page reload, it made an Ajax call... then updated the page with the new HTML? Well, that's *exactly* what Turbo Drive does. And it's a *huge* step towards making our app feel like a single page application.

Turbo itself is... just a JavaScript library! It has nothing to do with Symfony. But Symfony *does* have a package that makes it easier to use. Let's go get that package installed.

Head back to your terminal, open a new tab and run:

```
composer require "symfony/ux-turbo:^1.3"
```

After this finishes... run: `git status` to see what its recipe did:

```
git status
```

Ok: it looks like it installed a new bundle... called `TurboBundle`. It also changed our `package.json` file... let's go find that. It added two new packages including turbo itself. The recipe *also* updated our `controllers.json` file, which we learned about in the Stimulus tutorial. This adds a new Stimulus controller to our app. More on what that controller *does* a bit later.

assets/controllers.json

```json
{
    "controllers": {
        "@symfony/ux-chartjs": {
            "chart": {
                "enabled": true,
                "fetch": "lazy"
            }
        },
        "@symfony/ux-turbo": {
            "turbo-core": {
                "enabled": true,
                "fetch": "eager"
            }
        }
    },
    "entrypoints": []
}
```

But... you probably noticed that we have an error from yarn:

> *“The file `@symfony/ux-turbo/package.json` could not be found. Try running `yarn install --force`.”*

That makes sense! As we learned about in the first tutorial, we need to re-install our yarn dependencies so it can copy the new `@symfony/ux-turbo` package from our `vendor/` sdirectory into the `node_modules/` directory. Let's do it:

```
yarn install --force
```

When that finishes... run `yarn watch` again and... it's happy!

```
yarn watch
```

Hello Turbo Drive

Cool! So the `@hotwired/turbo` JavaScript package is now installed. Now... what do we need to do to *activate* Turbo Drive?

The answer is... nothing! It's already working!

Head back to your browser and refresh the page. Start clicking around. Woh! It's alive! And it *feels* fast!

Open up your browser tools... and then go to network tools and watch for XHR requests - or Ajax request. Yep! Every single click is now an Ajax request. There are *zero* full page reloads!

We now have... dare I say... a single page app! Tutorial finished! Good luck!

Okay, okay... of *course* the tutorial isn't finished yet. Turbo Drive feels like black magic... and that's never a great feeling. So next, let's discover how Turbo Drive *works* behind the scenes. We'll also see how Turbo was magically activated simply by installing it and I'll introduce you to a few subtle features of Turbo Drive that are already making the experience feel extra quick.

Chapter 3: How Turbo Drive Works

This is Turbo Drive. And yes, it feels like absolute magic. So let's break down how this works.

How Was Turbo Activated? The Magic Stimulus Controller

To start... we never wrote any JavaScript that said:

> *“Hey Turbo! Please activate your Drive functionality.”*

So... how would did this automatically start working? That is thanks to the magic of the `assets/controllers.json` file. This is normally a mechanism in Symfony UX for third party libraries to add new Stimulus controllers to our app. And in this case, that's true... but it's kind of a trick.

Let's go find the file that's being referenced. It lives in `node_modules/@symfony/ux-turbo` then `src/turbo_controller.js`. If you're wondering how I knew to open this exact file... this `turbo-core` string here matches up with a special key inside of the `package.json` file of this library. So `turbo-core` points to `dist/turbo_controller.js`. So, technically the file in the `dist/` folder is loaded... but I'm opening the original in `src/` because it's a bit easier to read.

And... there's not much here! This exposes an *empty* controller. And really, the *whole* point of this file is to import `Turbo` and set it onto the `window` variable. This accomplishes two things. First, when you import `Turbo`, it automatically activates Turbo Drive across your entire site. We'll talk about how to disable it globally or selectively a bit later. And second, Turbo is set onto the `window` variable, which makes it a *global* variable. You may or may not need this. It's useful if you need to programmatically visit a link, but from outside a JavaScript file. We'll see that later.

So we now know *who* activated Turbo. But... how the heck does Turbo Drive work? It's a pretty simple idea. Turbo watches link clicks - and also form submits like this add to cart form submit - and *intercepts* them. It then performs those requests in the background via an Ajax call, which we can see here. When that finishes, it updates the HTML of the page from the HTML in the response... *all* without a full refresh. But, it *does* modify the URL, which gives us normal browser behavior, like clicking back and forward.

Snapshots & Previews

Speaking of back and forward, Turbo Drive has a feature called "snapshots". Let me refresh the page real quick. As you navigate to a new page, it stores a "snapshot" of the page you're leaving. Then, if you click back in your browser, it instantly loads that snapshot with *no* network request. It does the same if you go forward. And if you *revisit* a page that you've already been to, so, a page whose snapshot has been stored, Turbo will give you an *instant* "preview" of that page while it waits for the Ajax call for that page to finish. You can see how *super* fast the pages are that we've already gone to versus ones that we have *not* gone to yet. By the way, this snapshot cache isn't persistent: it clears when you refresh the page.

Some of this preview & snapshot stuff is kind of hard to see because things are so fast. So in your editor, open up `public/index.php` and add a `sleep()` for two seconds.

public/index.php
```php
<?php

use App\Kernel;
sleep(2);
require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

// ... lines 7 - 10
```

Now head back to your browser and refresh the page... which takes 2 seconds. Click back to the homepage. Oh! This shows off the progress bar! If an Ajax call takes longer than 500 milliseconds, the progress bar shows up, which you can customize with CSS if you want. Because our site is slow, we see it each time we click to a new page.

But now, let's click *back* to "Office Supplies", which we visited before. When I do, watch closely: the page will show up instantly. Boom! *Then* it finishes loading the Ajax call. This is the *preview* feature. When you navigate to a page you've already been to, Turbo loads the page from cache for an instant experience. But it *still* makes an Ajax call for the page. And when that finishes, it takes the new HTML and renders it onto the page. Most of the time - like right now - we don't really notice that Ajax call finishing... because the new HTML is identical to the preview.

And if we click backward and forward, as we mentioned earlier, those pages load instantly with *no* Ajax request. Let's go take out that `sleep`.

Merging of the <head> Tag

Okay... but how does this all *really* work? What is Turbo doing behind the scenes to make it all happen? Let's go a step deeper. This is important because, to get Drive working happily on your site, as the saying goes, the devil is in the details. We'll spend the first part of this tutorial talking about potential problems that Turbo Drive can cause and how to fix them.

Let me refresh the page again to clear the snapshot cache.

Okay: when we click a link, Turbo intercepts that and makes an Ajax call instead. Oh, by the way, these extra Ajax requests are for the web debug toolbar.

Anyways, the Ajax request that Turbo makes when we click returns a *full* HTML page. When Turbo gets that full HTML response, it *merges* the *new* `head` tag into the existing `head` tag and then *replaces* the `body` with the *new* `body`.

The way it merges the old and new `head` is smart. Go over to the Elements part of the debugging tools and open up the `head` tag.

When the Ajax request finishes, Turbo first finds anything in the `head` *other* than JavaScript and CSS elements and *removes* those. Then it looks in the *new* `head` for any non JavaScript and non CSS elements and *adds* those.

We can actually see this. Reload the page and look back at the `head`. I see two non-JavaScript and non-CSS tags: a `meta` tag with the `charset` and the `title` element. When I click to go to another page, these will be removed. Then, any elements from the *new* page's `head` will be added to the bottom. And... boom! The new page happens to have the same two tags, but you can see that the original ones were removed and the new ones added. I was lazy and didn't give each page a unique title, but if the next page *did* have a new title, it *would* show up.

How JavaScript & CSS is Handled

Let's talk about how JavaScript and CSS is handled. If the *new* `head` tag contains JavaScript or CSS tags - and it probably will, since we're returning full HTML pages - Turbo checks to see if these elements already exist in the *current* `head`. If they do - like the next page we click *also* has a script tag for `build/runtime.js` - then Turbo ignores it. There's no reason to add the same `script` or CSS multiple times. But if the CSS or JavaScript element does *not* exist on the current page, it *will* add it. This is actually a big reason why Turbo Drive feels so fast: each time you navigate, your browser does *not* need to re-parse all of your JavaScript and CSS like it *normally* would with a traditional full page reload.

The result of all of this is... exactly what we see as we click around! The title changes on each page - the login page has a different title - and *if* a page contained new JavaScript or CSS, that would be added automatically.

So... this is amazing! Well, yes, it *is* amazing. But to get this amazingness to work perfectly, there *is* a little bit of work that we need to do. The first bit involves making your JavaScript Turbo-friendly. Let's dive into that topic next.

Chapter 4: Turbo-Friendly JavaScript

The *biggest* gotcha with Turbo Drive is JavaScript. And that's for one simple reason: suddenly there are *no* full page refreshes! And... a lot of JavaScript is written to *expect* that behavior.

How JavaScript in head Is Parsed

Let's see how some classic JavaScript behaves with Turbo. Open `assets/app.js`: this file is loaded on every page. Let's use jQuery to run some code after the page finishes loading. You might recognize this code.

Import `$` from `jquery` - I already have that installed. Then use `$(document).ready()` and pass a function that should be called once the page is fully loaded with `console.log('page is ready')`. After this block, also `console.log('script is done')`.

```
assets/app.js
// ... lines 1 - 14

$(document).ready(() => {
    console.log('page is ready!');
})
console.log('script is done!');
```

Cool. Go refresh... and check out the console. Yep! We see both logs: script is done first, then "page is ready" shortly after. But when we click to another page, we see nothing! And that makes sense! `app.js` is *not* re-executed... and the page does *not* become "ready" again. This is a *big* difference compared to a traditional web app. But, it's also what makes Turbo so fast: re-parsing all that JavaScript over and over again on each page load takes time!

The Problem of JavaScript in body

However, if you put JavaScript into the *body* of your page, then it *does* work like normal. Open up `templates/base.html.twig` and - anywhere in the `body`, I'll go to the bottom - add a `script` tag and `console.log('body executing!')`.

```
templates/base.html.twig
// ... lines 1 - 80
        </div>

        <script>
        console.log('body executing!');
        </script>
    </body>
</html>
```

Refresh now. We see all three logs. Click to another page. Hey! The new log is there! And... this *also* makes sense. Turbo replaces the old body with the new body. And so, any script tags in the new body are parsed & executed.

But... this is *not* necessarily a good thing... for two reasons! First, re-parsing the same JavaScript on every page is wasteful and can slow down your page. That's what Turbo Drive helps us avoid.

Second, putting JavaScript into your body can... sometimes cause weird things to happen. Watch closely: I'm going to clear my console.... then click back to a page that I just visited a minute ago. Woh! There are *two* logs!

This logged once when the page *preview* was shown from cache and a second time when the fresh HTML was rendered. This... might be okay? Logging two messages doesn't hurt anything. But this might cause some big problems in other situations, like double-counting page views in an analytics system. The topic of external JavaScript is something we'll dive into a bit later.

Here's another issue. Suppose you - or some third-party JavaScript library - adds an event listener to the entire document. Go back to `base.html.twig`. Use the `document` variable. `document` basically represents the `html` tag, which unlike the `body`, is *never* replaced by Turbo. Well, technically, `document` is sort of like the *owner* of the `html` element... but that's not important here.

Anyways, add an event listener to this: `document.addEventListener()` to listen to the `click` event. On click, `console.log('document clicked')`.

```
templates/base.html.twig
// ... lines 1 - 80
        </div>

        <script>
        document.addEventListener('click', () => {
            console.log('body clicked!');
        })
        </script>
    </body>
</html>
```

We should be able to click *anywhere* to see this message. Refresh, go to the console and... click. There it is! Click again and another log! Easy peasy.

Now clear the console and click to another page. Oh, let's clear the `console` again. And... click. Ah! *Two* logs! That is *definitely* not what we want!

This happens because, each time we execute the script, it adds *another* listener to the `document`. After 10 clicks, our function would be called 10 times!

Go remove the `script` tag and the jQuery loading code.

```
templates/base.html.twig
// ... lines 1 - 76
        <div
            class="footer mb-0"
            {{ stimulus_controller('made-with-love') }}
        >
        </div>
    </body>
</html>
```

```
assets/app.js
// ... lines 1 - 7
// any CSS you import will output into a single css file (app.css in this case)
import './styles/app.css';

// start the Stimulus application
import './bootstrap';
```

Writing JavaScript that you (and Turbo) will Love

So... what *is* the best way to write JavaScript so it works nicely with Turbo Drive? Well... Stimulus of course!

We already know from the first tutorial in this series that if a new `data-controller` element appears on the page - like `data-controller="counter"`, which powers this contest area up here, its Stimulus controller will *always* work perfectly, even if that HTML is loaded via Ajax. *That* is the most *powerful* part of Stimulus and it works *brilliantly* with Turbo.

One other lesson is that you should probably remove any JavaScript that you have inside your `body` element... even though it mostly works. That's because of the potential for the bad behavior that we saw a minute ago. In a little while, we'll talk about external JavaScript - like widgets or analytics - which are often supposed to be added to your body.

But let me be clear about one thing: I do *not* want you to think about all of this like:

> *“Hey! Turbo is forcing me to write my JavaScript a certain way!”*

Nope: Turbo is forcing you to write *better* JavaScript: JavaScript that only needs to be loaded and executed *once*... and then keeps on working forever, even as new content is loaded onto the page.

So this whole JavaScript topic is definitely the biggest hurdle to using Turbo Drive. Until you have all the JavaScript on your site written properly, things won't work well. But you *can* fix the JavaScript for just *some* pages on your site and activate Turbo Drive only for those. Let's see how next and also learn how we're able to put all of our `script` tags into the `head` element *without* hurting page-load performance.

Chapter 5: The "defer" Attribute & Conditionally Activating Turbo

Inspect element and go check out the `head` tag. Notice that all of our `script` elements live up here in the `head` with a `defer` attribute. That's on purpose. And this `defer` attribute comes from our configuration: `config/packages/webpack_encore.yaml`: `script_attributes`, `defer`

config/packages/webpack_encore.yaml

```yaml
// ... lines 1 - 6
    # Set attributes that will be rendered on all script and link tags
    script_attributes:
        defer: true
    # link_attributes:
// ... lines 11 - 31
```

The defer Attribute

The reason we placed our `script` tags up in the `head` element is... well, we learned why in the last chapter! By adding them here, they won't be re-executed on every Turbo visit.

But normally, adding `script` tags to the `head` is bad for performance. When your browser sees a `script` tag, it freezes the page rendering while it downloads the file and executes it. But by adding `defer`, the file is downloaded in the background and the page continues loading *without* waiting. Once the page *finishes* loading, *then* the JavaScript is executed. If you want to learn more about the `defer` attribute, we have a blog post about it on symfony.com: https://symfony.com/blog/moving-script-inside-head-and-the-defer-attribute

Anyways, here's the big takeaway about using Turbo Drive and JavaScript: to get it to work reliably, all of your JavaScript needs to be written in Stimulus. But that does *not* mean that you need to completely rewrite it. If you have a big block of JavaScript that works on an element, you can copy that code into the `connect()` method of a Stimulus controller, which is called each time a matching `data-controller` element is found. Often, the only change you need to make is to remove any `document.ready()` code and tweak your JavaScript to target `this.element`.

And... if you can't or don't want to use Stimulus, you can also tweak your code so that it's executed on each "Turbo page load", like by wrapping that code in a Turbo event, that's fired on each visit instead of using jQuery's `document.ready()` method. We'll talk about Turbo events later.

Completely Disabling Turbo

By the way, if you *did* need to disable Turbo for a specific link... or even for an entire section of the page, you can do that with a special `data-turbo` attribute. For example, to *completely* disable Turbo drive on your entire site, head over to `base.html.twig`. Find the `body` tag and add `data-turbo="false"`.

templates/base.html.twig

```
// ... lines 1 - 11
        {% endblock %}
    </head>
    <body data-turbo="false">
        <div class="page-top">
            <header class="header px-2">
// ... lines 17 - 84
```

Now, any link clicks or form submits *inside* of this element - which is everything - will *not* use Turbo drive. Check it out: refresh the page and click around. We are back to boring full page reloads. Boo.

To reenable Turbo Drive on a link or section, you can set the same attribute to true. For example, let's activate Drive for *just* the links up in the `navbar`. Find that element... it's this `ul`, and add `data-turbo="true"`

```
templates/base.html.twig
// ... lines 1 - 27

                    <ul class="navbar-nav" data-turbo="true">
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('app_cart') }}">
                                Shopping Cart ({{ count_cart_items() }})
                            </a>
                        </li>
// ... lines 35 - 84
```

Refresh again. When we click a category, it still triggers a full page reload. But if we click to go to the cart... that loaded with Drive! You can use this strategy to activate Turbo Drive on only some parts of your site that are ready.

Let's remove both of these to *fully* get Turbo Drive again.

```
templates/base.html.twig
// ... lines 1 - 12
    </head>
    <body>
        <div class="page-top">
            <header class="header px-2">
// ... lines 17 - 27

                    <ul class="navbar-nav">
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('app_cart') }}">
                                Shopping Cart ({{ count_cart_items() }})
                            </a>
                        </li>
// ... lines 35 - 84
```

Next: we've activated Turbo Drive and gotten the no-page-reload goodness with zero changes to our Symfony code! That's... amazing! But... there is *one* tiny change that we *will* need to make to any pages that have a *form*.

Chapter 6: Form 422 Status & renderForm()

We already know that Turbo Drive also works for form submits. To prove it, head to the login page and log in as `shopper@example.com` password `buy`... using these handy cheating links that are powered by a Stimulus controller.

Submit and... yep! That loaded via Turbo! Now head to the admin area. This is a generated CRUD for creating, editing and deleting products. Click to edit a product and... make it look a bit more exciting with some exclamation points. Hit enter to submit and... that worked too! It submitted via Ajax and redirected back to the list page. There are my exclamation points!

Failing Validation... Doesn't Work?

But now, let's make a change that will fail *validation*: clear out the name field and... hit Update. Uh... nothing happened? Check out the console. Ooh.

> *"Form responses must redirect to another location."*

Okay. Part of what makes Turbo so cool is that you get the single page app experience without making *any* changes to your server code. But the *one* big exception to that rule is forms. Don't worry: the change we need is minor... it's really an *improvement* on our code. And the change is *especially* easy in Symfony 5.3.

The 422 Status Code

Let's go find the controller for this page: it's in `src/Controller/ProductAdminController.php`... and `edit` action. Here we go. In short, if the form has a validation error, we need to return a 422 status code instead of a 200 status code.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 63
    /**
     * @Route("/{id}/edit", name="product_admin_edit", methods={"GET","POST"})
     */
    public function edit(Request $request, Product $product): Response
    {
        $form = $this->createForm(ProductType::class, $product);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $this->getDoctrine()->getManager()->flush();

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

        return $this->render('product_admin/edit.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
        ], new Response(null, $form->isSubmitted() && !$form->isValid() ? 422 : 200));
    }
// ... lines 83 - 98
```

Right now, both when the page originally loads *and* when we have a validation error, we return `$this->render()`, which sets a 200 status code. Using a 422 status code when there's a validation error is actually more correct. And it tells Turbo that the form submit failed and it should re-render the page with the new HTML.

So how can we set the status code on the response that `$this->render()` creates? The easiest way is by passing the little-known third argument: a `Response` object that the render function will put the template content *into*. Say `new Response()` - get the one from `HttpFoundation` and pass `null` for the content, because that will be replaced by the template HTML. For the status code, we can't use 422 all the time because we don't want that status code when we simply navigate to this page. So use the ternary syntax: if `$form->isSubmitted()` and `$form->isValid()`, I mean if *not* `$form->isValid()`, then use 422. Else use 200.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 76

        return $this->render('product_admin/edit.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
        ], new Response(null, $form->isSubmitted() && !$form->isValid() ?
422 : 200));
    }

// ... lines 84 - 98
```

That's it! Back over at the browser, we don't even need to refresh. Hit update and... voilà! We see the validation error! Let's put the content back...remove my exclamation points, hit enter again and... it works.

Turbo Handles Redirects too

By the way, on success, in our controller, we are redirecting with a 302 status code, which is perfect! That *is* what you should do after a successful form submit.

The interesting thing is that... Turbo correctly handled this!

Check out your network tools. Let's look closely at what happened when we submitted the form. This request is the `POST` request from the submit. It returned a 302 redirect. When an Ajax request returns a redirect, your browser automatically *follows* it. What I mean is: in this case, our browser made a *second* Ajax request to the redirect URL - which is the product list page.

At this point, Turbo did something really smart: it *detected* that this 2nd Ajax request happened due to a redirect. It then used the HTML from that Ajax call to update the page like normal *and* it changed the URL in our browser to match the redirected URL. In other words, redirects work *perfectly* with Turbo Drive out of the box.

Now if you look at the Turbo documentation, they will tell you to return a 303 status code instead of 302 when redirecting after a form submit. But both work *exactly* the same. 303 is... *technically* a little bit more correct... and so more hipster... but it really doesn't matter.

Symfony 5.3's renderForm() Shortcut

Okay, back to this 422 status code fix. If you're using Symfony 5.3 - and I am - then fixing this is even easier thanks to a new `renderForm()` controller shortcut. Here's how it works: change `render()` to `renderForm()`. Then, remove the Response object.

That's it! Well, that's *almost* it. Also remove the `createView()` call on the form.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 76

        return $this->renderForm('product_admin/edit.html.twig', [
            'product' => $product,
            'form' => $form,
        ]);
    }
// ... lines 83 - 98
```

Let's break this down. The `renderForm()` method is *identical* to `$this->render()` except that it loops over all of the variables that we pass into the template. If any of them are a `Form` object, it does two things. First, it calls `createView()`, which is just a really kind thing for it to do: we don't have to call that ourselves anymore. Second, if the `Form` has been submitted and it's invalid, it changes the status code to 422.

So all we need to do now is repeat this change everywhere else in our app... which is kind of boring, but simple! Copy `renderForm()` and scroll up to the `new` action. You can actually see that we did the 422 logic in the first tutorial because we wrote some custom JavaScript that - like Turbo - needed to know if a form was simply rendering or if it had a validation error.

Change this to `renderForm()`, we don't need `createView()`... and we don't need the third argument at all. Much nicer.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 53

        return $this->renderForm('product_admin/' . $template, [
            'product' => $product,
            'form' => $form,
        ]);
    }
// ... lines 60 - 95
```

Let's clear the tabs and go to `CartController`. There are two spots inside here. I'll search for `createView()`.

```
src/Controller/CartController.php
// ... lines 1 - 29

        return $this->renderForm('cart/cart.html.twig', [
            'cart' => $cartStorage->getOrCreateCart(),
            'featuredProduct' => $featuredProduct,
            'addToCartForm' => $addToCartForm,
        ]);
    }
// ... lines 37 - 68

        return $this->renderForm('product/show.html.twig', [
            'product' => $product,
            'categories' => $categoryRepository->findAll(),
            'addToCartForm' => $addToCartForm,
        ]);
    }
// ... lines 76 - 107
```

Cool: `renderForm()`, then take off `createView()`. For the next one... it's exactly the same. I'll take a big sip of coffee... and speed through the rest of the controllers: `CheckoutController` has one spot, `ProductController` has two spots, one of which renders two forms including a conditional `reviewForm` that can be simplified, `RegistrationController` has one spot... and `ReviewAdminController` has two spots.

```
src/Controller/CheckoutController.php
// ... lines 1 - 45

        return $this->renderForm('checkout/checkout.html.twig', [
            'checkoutForm' => $checkoutForm,
            'featuredProduct' => $featuredProduct,
            'addToCartForm' => $addToCartForm,
        ]);
    }
// ... lines 53 - 71
```

src/Controller/ProductController.php

```
// ... lines 1 - 58

        return $this->renderForm('product/show.html.twig', [
            'product' => $product,
            'currentCategory' => $product->getCategory(),
            'categories' => $categoryRepository->findAll(),
            'addToCartForm' => $addToCartForm,
            'reviewForm' => $reviewForm ?: null,
        ]);
    }
// ... lines 68 - 91

        return $this->renderForm('product/reviews.html.twig', [
            'product' => $product,
            'currentCategory' => $product->getCategory(),
            'categories' => $categoryRepository->findAll(),
            'reviewForm' => $reviewForm?: null,
        ]);
    }
// ... lines 100 - 109
```

src/Controller/RegistrationController.php

```
// ... lines 1 - 47

        return $this->renderForm('registration/register.html.twig', [
            'registrationForm' => $form,
            'featuredProduct' => $productRepository->findFeatured(),
        ]);
    }
}
```

```
src/Controller/ReviewAdminController.php
// ... lines 1 - 43

        return $this->renderForm('review_admin/new.html.twig', [
            'review' => $review,
            'form' => $form,
        ]);
    }
// ... lines 50 - 63

        return $this->renderForm('review_admin/edit.html.twig', [
            'review' => $review,
            'form' => $form,
        ]);
    }
// ... lines 70 - 85
```

Phew! Good, straightforward, boring work. The only form we *didn't* need to change was the login form. That's because the login form works a bit differently than other forms on our site. On failure, it redirects and stores the error in the session. So if we put some bad info and submit... it already works fine.

Hey! With a few small changes to our code, our site now has fully-functional Ajax submitted forms! That's just... incredible.

Next, let's talk more about that snapshot functionality: the feature that instantly shows you a page from cache when hitting the back button or when navigating to a page that we've already been to. As *awesome* as that feature is - and it really makes the site feel fast - sometimes it can take a snapshot when the page is in a "state" that we don't want.

Chapter 7: Form Submits & The Preview Feature

One of the cooler features of Turbo Drive is its snapshot feature, which we know about already. When we visit a page that we've already been to, like Office Supplies or Furniture, it instantly shows the snapshot while it waits for the new Ajax call to finish in the background. And when we hit back, it instantly shows the snapshot with *no* Ajax call.

This feature, which is *great* for making your site feel *really* snappy - is, I'll admit, one of the most *problematic* when it comes to perfecting your site with Turbo Drive.

Snapshots and Form Submits

Let's see one problem. Head over to the registration page and fill out the form incorrectly: I'll use a bogus email address and hit enter. Cool. The form submitted via Ajax and we see the errors.

Now click back to the homepage. I'm going to revisit the registration page. But watch closely when I do. Woh! For *just* a moment, we saw the form *with* the values filled in and the validation errors!

Here's what happened. When we were originally on the registration page with the validation errors showing, we clicked to leave the page to go to the homepage. At *that* moment - *just* before we were navigated away, Turbo saved the snapshot for the registration page. That means the snapshot was for a page that had filled-in form fields and validation errors.

Then, when we clicked back to the registration page, that snapshot was restored with errors and all. A moment later, when the Ajax call finished, the fresh content - with an empty form - replaced the snapshot.

This is a known issue with submitted forms. And... well... maybe it's not really an issue. It's... tricky. And maybe you don't really care that this shows up for a moment before it clears. In that case, just ignore it and move on with your life! Go grab a baguette!

How to Handle Problematic Snapshots

But let's say that we *do* want to avoid this. One option is that we could disable the snapshot from being taken on this page *completely*. But when I fill out the form... and get the errors... and go to the homepage... and then hit the "back" button in my browser, it *is* nice that, thanks to the snapshot, we see the form *with* the fields still filled-in. So... you kind of want the snapshot cache to be used when hitting the back button... but not for the preview.

There are two main ways to fix the problem of a "bad snapshot". The first involves *preparing* a page before its snapshot is taken. We could clear the form errors and empty the fields so that the snapshot is clean. The code to do this would work for *any* form on your site... so it would kind of take care of everything all at once. The only downside is that clicking the back button would show an empty form. We're not going to use this solution in *this* case, but we will leverage this soon for a different preview problem.

Disabling the Preview on a Page

A second solution is to simply disable the preview feature for *this* page. And, that's one of the nice things about Turbo. Don't like something? Just disable it.

How? By adding a special `meta` tag to the `head` element. Head over to the code and open up `templates/base.html.twig`. We don't want to remove the preview functionality for *every* page. So instead of adding the `meta` tag right here, add a block so that a *child* template can add new meta elements: `{% block metas %}` `{% endblock %}`.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
        <meta charset="UTF-8">
        {% block metas %}{% endblock %}
        <title>{% block title %}MVP Office Supplies{% endblock %}</title>
        {% block stylesheets %}
            {{ encore_entry_link_tags('app') }}
        {% endblock %}

        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </head>
// ... lines 15 - 85
```

Now open up `templates/registration/register.html.twig` and override that block: `{% block metas %}`, `{% endblock %}` and inside add `<meta> name="turbo-cache-control"` with `content="no-preview"`.

templates/registration/register.html.twig

```
// ... lines 1 - 4
{% block metas %}
    <meta name="turbo-cache-control" content="no-preview">
{% endblock %}
// ... lines 8 - 36
```

The `no-preview` means: don't show a preview for this page. The other possible value is `no-cache`, which tells Turbo to not do *any* snapshotting: not even for the back button.

Let's see how this feels! Refresh the registration page, fill out the form with errors and click away from this page. Now, click back to it. Beautiful! Instead of instantly showing the preview, it stayed on the previous page until the new Ajax call finished loading, just like a *normal* navigation. You can repeat this for any pages that have a public-facing form where you care enough to avoid this problem.

Dimming the Opacity of a Preview

Speaking of the preview feature, you can also change what a preview *looks* like... in case you want to make it more obvious that a preview is being shown or give it a "loading" feel. How? Open your Elements inspector. It's quick, but watch this `html` element. Whenever you navigate and a preview is rendered, Turbo will add a `data-turbo-preview` attribute to the `html` element.

Boom! It was fast, but I saw it! Let's use that to see if we can lower the *opacity* on previews.

Head over to `assets/styles/app.css`. Target that attribute using the lesser-known attribute syntax: `[data-turbo-preview]` then `body` to apply some body styling. Set the `opacity` to .2 so it's really obvious.

assets/styles/app.css

```
// ... lines 1 - 7
[data-turbo-preview] body {
    opacity: .2;
}
// ... lines 11 - 167
```

Let's go check it! Refresh. As we click to new pages, we don't see anything. But if we click to a page that we've been to... yes! The whole page was nearly invisible while the preview was being shown. This is also kind of a fun way, while you're developing, to get a feel for when a preview is shown.

But... since this looks a bit extreme, let's go back to `app.css` and comment it out.

```css
assets/styles/app.css
// ... lines 1 - 7
/*
[data-turbo-preview] body {
    opacity: .2;
}
*/
// ... lines 13 - 169
```

Next: in addition to the form situation we just saw, there's one other common time when the preview feature will do something that... we don't want. Let's talk about what happens when something like a modal is open at the moment a snapshot is taken.

Chapter 8: The Problem of Snapshots & JavaScript Popups

Let's go log in so we can access the product admin page. I'll click the cheating links to fill in the fields and hit sign in. Now click "Admin" and then click the "New" button.

Snapshotting Pages with an Open Modal

This opens a Bootstrap 5 modal. Oh, and usually there is a dark gray backdrop... behind this... which is missing right now. Refresh... then hit this button again. *There* is the backdrop. Why was it missing the first time? It's actually a bug in Bootstrap 5.0.1 when using Turbo. But don't worry, it's already fixed and will be available in 5.0.2.

Anyways, now that I have this modal open with my backdrop, click the back button in your browser and then revisit the admin page. Woh! The modal was still open for just a moment and then closed. This is *very* similar to what happened with our submitted form. The snapshot was taken when the modal was open. And so, when the preview is rendered... it... still has a modal!

Do this flow again: click the button then hit "back" in your browser. But this time hit the "forward" button in your browser. Whoa. The modal stays open! Which I guess is okay: that *is* an accurate representation of the page's state. The only problem is that... well... the modal is completely nonfunctional. I can click the "Cancel" button until Symfony 10 comes out... and nothing will ever happen.

Snapshotting: Event Listeners are Lost

There are a few important things we need to understand. As we talked about a few minutes ago, the snapshot for a page is taken the moment you navigate *away* from that page. And so, if a modal or a dropdown or anything else is currently visible at that moment, well... it gets cached!

Also when Turbo takes a snapshot, it *clones* the `body` element using a method called `cloneNode()`. That's important because it means that any JavaScript listeners - like an "on click" listener for this cancel button - are *not* included in that clone. When we're looking at a snapshot, it's not *really* the same `body` from before: it's a clone with no JavaScript listeners attached.

That is why the modal doesn't work: it's the same HTML, but without any JavaScript listeners. This was an intentional design decision inside Turbo. Cloning the `body` element, which removes all of the listeners, helps keep Turbo fast by avoiding memory leaks.

If you write all of your JavaScript with Stimulus, this is no problem. When the snapshot is restored, a new Stimulus controller instance will be created automatically and everyone is happy. But in this case, this is Bootstrap's modal... so we can't exactly tell them to use Stimulus.

And, besides, even if this modal *was* functional, it would still show up and then disappear when we navigate back to the admin page... which isn't a huge deal, but it's not perfect.

Listening to the turbo:before-cache Event

So what's the solution? Clean up the page *before* the snapshot is taken. Head over to the Turbo documentation, click on Reference and go to Events. Turbo dispatches a *bunch* of events when it does different things, like when we visit a page or submit a form. Learning how to leverage these will be the difference between a "nice" Turbo experience and an *awesome* one. Check out this `turbo:before-cache` event:

> *“Fires before Turbo saves the current page to cache.”*

That sounds perfect! We could run code to close the modal! Copy that event name.

How do we use this? Open up `assets/app.js`. Usually when we want to add some JavaScript, we write a Stimulus controller. But for Turbo events, we actually don't need that. Instead, say `document.addEventListener()` - which is how you add an event listener in normal JavaScript - then paste the event name. Pass an arrow function with an `event` argument and, inside, `console.log(event)`.

```
assets/app.js
// ... lines 1 - 12

document.addEventListener('turbo:before-cache', (event) => {
    console.log(event);
});
```

Turbo dispatches most of its events on the `html` tag itself. And, remember, as we navigate around, the `html` element is never removed: this one `html` element sticks around forever. That's nice because it means we can attach an event listener to it just *one* time and it will always be there. And since `app.js` is only executed *once* - on initial page load - the listener won't be added over and over again as we navigate to new pages.

Oh, and like we talked about earlier, the `document` variable is kind of the "parent" of the `html` element. You can attach the event to *it* - like we're doing - or to the actual `html` element itself... which is `document.documentElement`. It doesn't matter.

Anyways, let's see this in action. Go refresh the page and open the console. Now, click to another page. There it is! The moment we navigated away from the product admin page, a snapshot was taken. If you expand the `event` object that we logged, often this `detail` key here will contain extra information that's relevant to this event. There's nothing in this case... but we *will* see this with other events later.

Let's Hide the Modal!

So here's my thinking: we're using Bootstrap 5's modal system, and it has a built-in method to hide a modal. So, in this function, *if* a modal is open, we'll call that `hide()` method and... done! The page will cache with a hidden modal and we can all take a snapshot of a group high-five.

To do that, import `{ Modal }` from `bootstrap`. Remove the `event` argument - we won't need it - and the log. Now, if `document.body` - that's an easy way to get the `body` element - `.classList.contains('modal-open')`, then we know that there *is* a modal currently open.

assets/app.js

```
// ... lines 1 - 13
import { Modal } from 'bootstrap';

document.addEventListener('turbo:before-cache', () => {
    if (document.body.classList.contains('modal-open')) {
        const modal = Modal.getInstance(document.querySelector('.modal'));
        modal.hide();
    }
});
```

I'm using a bit of Bootstrap-specific knowledge here. Click over to the product admin page and open the modal. Yup! When the modal is open, the `body` element gets a `modal-open` class. We're using that as an easy way to check if the modal is open.

Inside of the if, now that we know that the modal *is* open, we can say `const modal =` and use a nice method from Bootstrap to get that the modal instance that's connected to our element: `Modal.getInstance()` and pass it the Element that the modal is attached to. If you inspect element, it's always going to be this element here: the one with the `modal` class. We can find that with `document.querySelector('.modal')`.

If you're not very familiar with using native JavaScript without jQuery, that's fine. You *can* use jQuery instead of native JavaScript if you want to. But this is about as complicated as it gets. We're using `classList` to see if an element has a class and then using the `querySelector()` method to find an element with a certain class on it. Now that we have the Bootstrap modal instance, we can call its `hide()` method: `modal.hide()`.

That's it! Testing time! Find your browser, refresh, open the modal, hit back, then hit forward. Ummmm. It... kind of worked? The modal isn't there... but this gray backdrop *is* there?

What happened? The problem is that Bootstrap's `hide()` method is asynchronous. To say that a less-fancy way, when you hide a Bootstrap modal, it doesn't instantly hide: it fades out over time with an animation. *After* that animation finishes, it does the rest of its cleanup, like removing this backdrop. Unfortunately, the snapshot is taken immediately, before the modal has finished doing all of its cleanup.

This is one of the *trickiest* things with the preview feature: how to clean up and play nice with third-party JavaScript. So next, let's find a way to solve this both for Bootstrap's modal and also a Sweetalert modal that we have on a different page. That will give us clean preview functionality across our entire site whenever either of these are used.

Chapter 9: Cleanup Before Snapshotting (e.g. Modals)

Refresh the page, open a modal, click back, then click forward again. Say hello to a very strange-looking page. The modal did *not* completely hide itself. The problem is that hiding a modal is asynchronous: Bootstrap *waits* for the transition to finish before finally removing all of the elements, like its backdrop.

But the snapshot does *not* wait: Turbo takes the snapshot immediately, which is when the modal has only *started* to be removed and its backdrop is still visible. Worse, because the modal element is technically *removed* from the page, it's CSS transition is canceled. That's... a very low-level detail... but it means that Bootstrap's modal system is never notified that the animation finished, and so it never does its final cleanup.

Forcing Bootstrap's Modal to Close Immediately

The solution is to force both the modal and the backdrop to hide *synchronously* in this situation: to *not* use the animation that you *normally* see on close.

Telling a modal that you want it to work *without* an animation *is* something you can configure. But I don't want to remove the animation entirely: I only want to remove it when I'm hiding it right before the snapshot is taken. Unfortunately, changing whether or not you want a modal to have an animation *after* you create it is... well... not something that's really supported.

So... it's a bit ugly to get this working. I'll paste in the code.

```
assets/app.js
// ... lines 1 - 13
import { Modal } from 'bootstrap';

document.addEventListener('turbo:before-cache', () => {
    if (document.body.classList.contains('modal-open')) {
        const modalEl = document.querySelector('.modal');
        const modal = Modal.getInstance(modalEl);
        modalEl.classList.remove('fade');
        modal._backdrop._config.isAnimated = false;
        modal.hide();
        modal.dispose();
    }
});
```

This does the same thing as before: it finds the element, gets the modal instance and calls `hide()` on it. But it also does some extra stuff. Most importantly, before it hides, we remove the `fade` class from the modal. We also reach *into* this ugly internal backdrop object's config to set an `isAnimated` flag to false.

The results is that bootstrap will now know that both the modal and the backdrop should *not* use an animation: both should hide instantly.

The precise fix for this type of problem will be different each time you run into it. And usually, you'll need to dig around in the third-party code a bit to find out the best option. Figuring this out, I admit, was tricky. But ultimately don't over-think it: your goal is to basically clear out any elements that you don't want visible in the snapshot. Often, you can just find the problematic element and remove it.

The good news about what we have here is that this will fix the problem for the entire site. Let's see it. Refresh the page, open the modal, click back, click forward and... yes! It's gone. If we click to add a new product, the modal still works! You might notice that the *backdrop* is missing... but that's only due to the bug in Bootstrap 5.0.1 that I mentioned earlier. That will *not* be a problem in 5.0.2.

Dynamically Disabling Snapshot Caches

By the way, if you're having trouble figuring out how to clean up some third party code before the page is snapshotted, there is one other, less-elegant, but simpler solution. Instead of trying to *remove* the problematic element, you could disable the snapshot cache *only* when that element is open.

I won't actually try this live in the video, but let's see how this might work. Bootstrap's modal system dispatches an event both when a modal is opened and when it's hidden. We can use that to add and remove the `turbo-cache-control` meta tag that we saw earlier.

For example, check out this code:

```
// assets/app.js
const findCacheControlMeta = () => {
    return document.querySelector('meta[name="turbo-cache-control"]');
}

document.addEventListener('show.bs.modal', () => {
    if (findCacheControlMeta()) {
        // don't modify an existing one
        return;
    }

    const meta = document.createElement('meta');
    meta.name = 'turbo-cache-control';
    meta.content = 'no-cache';
    meta.dataset.removable = true;
    document.querySelector('head').appendChild(meta);
});
```

This listens to the `show.bs.modal` event, which is dispatched every time *any* modal is opened. Inside, if there is already a `turbo-cache-control` meta tag, we do nothing: we don't want to change any cache behavior. But if there is *not* one, we *add* a `turbo-cache-control` set to `no-cache`.

Thanks to this, if we leave the page when the modal is open, Turbo will see that this page should *not* be cached. Hitting back or revisiting the page will result in a normal navigation visit where *no* snapshot is used.

Notice that I added an extra `removable` key to the meta tag's `dataset`. That's useful when *removing* this meta tag when the modal closes. Check out the other half of this code:

```
// assets/app.js
const findCacheControlMeta = () => {
    return document.querySelector('meta[name="turbo-cache-control"]');
}

// ...

document.addEventListener('hidden.bs.modal', () => {
    const meta = findCacheControlMeta();
    // only remove it if we added it
    if (!meta || !meta.dataset.removable) {
        return;
    }

    meta.remove();
});
```

The `hidden.bs.modal` event is dispatched after a modal has been *fully* removed from the page. If we find a `turbo-cache-control` meta tag *and* it has the `removable` data key - which means *we* added it - we know it can now be safely removed. Thanks to this, if we navigate away from the page, Turbo will create a snapshot like normal.

This solution is, maybe less-elegant than the one I'm using... but in practice, it works really well, and could be repeated for any other problematic JavaScript elements on your site.

Next: now that we've crushed the Bootstrap modal, let's see one other example with a Sweetalert modal. I'll also show you a Webpack trick where we can import Sweetalert to help us hide the element... but without causing SweetAlert's JavaScript to be downloaded on every page.

Chapter 10: Fixing the Sweetalert Modal

So... we've crushed the Bootstrap modal problem! But we have the same issue in one other spot. Go back to the homepage, add an item to your cart and *go* to the cart. Try to remove the item. This cute little dialog is powered by a library called Sweetalert. Once again, if we click back and then forward, it pops up again, which *might* be ok... if it actually worked! But... it doesn't... because all of its event listeners are gone.

Okay: let's try using Sweetalert's close functionality to tell it to close before the page is snapshotted. To do that, `import Swal from 'sweetalert2'`.

Then, down inside of the function, if `Swal.isVisible()` - they have a nice function to check if Sweetalert is visible - then `Swal.close()`.

assets/app.js

```
// ... lines 1 - 13
import { Modal } from 'bootstrap';
import Swal from 'sweetalert2';

document.addEventListener('turbo:before-cache', () => {
    if (document.body.classList.contains('modal-open')) {
        const modalEl = document.querySelector('.modal');
        const modal = Modal.getInstance(modalEl);
        modalEl.classList.remove('fade');
        modal._backdrop._config.isAnimated = false;
        modal.hide();
        modal.dispose();
    }

    if (Swal.isVisible()) {
        Swal.close();
    }
});
```

It's that simple! Or at least... it *might* be. Let's go try this. Refresh the cart page, hit remove, go back, go forward and... it worked! Wait, I can't scroll... and nothing is clickable! Inspect element anywhere. Uh oh: a Sweetalert backdrop element is still there! It's invisible, but it's blocking the page!

This is the exact same problem we just saw with Bootstrap's modal: the close animation never finishes, so cleanup never happens. The solution is to, once again, tell Sweetalert to close... but *without* an animation.

This is easier than Bootstrap... but it still tool some digging to figure out how to do it. In this case, right before we close, we can say `Swal.getPopup()` - which gives you the `Element` associated with the dialog - `.style.animationDuration = 0ms`.

```
assets/app.js
// ... lines 1 - 13
import { Modal } from 'bootstrap';
import Swal from 'sweetalert2';

document.addEventListener('turbo:before-cache', () => {
    if (document.body.classList.contains('modal-open')) {
        const modalEl = document.querySelector('.modal');
        const modal = Modal.getInstance(modalEl);
        modalEl.classList.remove('fade');
        modal._backdrop._config.isAnimated = false;
        modal.hide();
        modal.dispose();
    }

    if (Swal.isVisible()) {
        Swal.getPopup().style.animationDuration = '0ms'
        Swal.close();
    }
});
```

How could I possibly know that *this* is the code we need? If you look internally at Sweetalert, you'll notice that it looks at its popup element and checks to see *if* the popup element has an `animationDuration` declared on its style. If it does, then it waits for the animation to finish before cleaning up. By changing the `animationDuration` to zero, Sweetalert will *now* see that it does *not* need to wait... and will clean up everything immediately.

Let's try it! Refresh, click remove, click back and click forward. Everything looks fine! When I hover over the checkout button, it is *not* being blocked by a backdrop *and* I can click "remove" again. All good!

Lazily Importing Sweetalert

One tiny problem with this approach is that both Bootstrap's modal and the `sweetalert2` library will now be downloaded on every page since we're importing them from our main `app.js` file.

You might not care... and you probably shouldn't care... at least not until you investigate optimizing your CSS and JS file sizes later.

But, this *is* interesting. Sweetalert is only used on this *one* page. So, it's kind of wasteful to force the user to download it on *every* page load... even though they will rarely need it.

Open `assets/controllers/submit-confirm_controller.js`. This is the controller that handles the Sweetalert confirmation on this page. Notice that it has `stimulusFetch: lazy` above it.

```
assets/controllers/submit-confirm_controller.js
// ... lines 1 - 4
/* stimulusFetch: 'lazy' */
export default class extends Controller {
    static values = {
        title: String,
        text: String,
        icon: String,
        confirmButtonText: String,
        submitAsync: Boolean,
    }
// ... lines 14 - 54
```

This is something that we added in our Stimulus tutorial. Thanks to this, *before* we started adding all of this new code in `app.js` - so pretend this isn't there - the `sweetalert2` JavaScript was *not* downloaded on every page. It was only downloaded when an element that uses this controller first appeared on the page... which is pretty cool! The code for this controller & its dependencies literally *waits* until its needed and then downloads itself.

But now that we're importing `sweetalert2` directly in `app.js`, it *is* being downloaded on every page. If you care enough about this, you can fix it using a very specific Webpack trick. It's a little nuts actually. I'll paste in the first half of the code, indent, then close things.

```
assets/app.js
// ... lines 1 - 25
    // internal way to see if sweetalert2 has been imported yet
    if (__webpack_modules__[require.resolveWeak('sweetalert2')]) {
        // because we know it's been imported, this will run synchronously
        import('sweetalert2').then((Swal) => {
            if (Swal.default.isVisible()) {
                Swal.default.getPopup().style.animationDuration = '0ms'
                Swal.default.close();
            }
        })
    }
// ... lines 36 - 37
```

Let's walk through this. The `__webpack_modules__` thing is an internal way - along with `require.resolveWeak` - to check to see if `sweetalert2` has *already* been downloaded and is available. But it does this *without* causing it to become packaged with `app.js`. If it *has* already been downloaded, *then* we can use this import to grab it. Because we know it's already available, this executes instantly. Then, we run our normal code down here. The only thing we need to change - due to the way that the `import()` function works - is that every `Swal` needs a `.default` to access that module's `default` export.

If this isn't making much sense to you, don't worry. This is a complex performance optimization. I thought I'd mention it for the performance and Webpack geeks out there.

Oh, and before we try this, scroll up and remove the now-unused import.

To see the result of this, go back to the homepage and then do a full page refresh. Over on the network tools, view the JS tab. It's not super obvious yet, but if you look closely at the names here... you won't see any that mention sweetalert2. It has *not* been downloaded yet.

Let me clear this and let's watch what happens when we click to the cart page. Yes! Check it out. One of the files that was downloaded - this one - has `sweetalert2` in the middle of its name! That contains Sweetalert and proves that it wasn't downloaded until it was actually needed... even though we have some code in `app.js` that takes advantage of it.

So now that we've tackled some of the most annoying problems with Turbo, which is cleaning up the snapshot, let's organize all of the new event code to make room for more turbo event listeners later. That will put us in a great position to discuss the last tricky thing with Turbo drive: handling third-party JavaScript like JavaScript widgets and analytics code.

Chapter 11: Organizing our Turbo Events Code

To get Turbo Drive to work *super* nicely, we're going to need to hook into a few events, like `turbo:before-cache`. Before we're done, we'll listen into even *more* of these to help us properly load JavaScript widgets, add transitions, and do more craziness when we talk about Turbo Frames.

Isolating the Turbo Logic

So instead of putting all that logic right here in `app.js`, let's organize a bit. There's no right or wrong way to do this, but let's create a class that holds all of the special Turbo logic. In the `assets/` directory, add a sub-directory called `turbo/` and, inside, a new file: `turbo-helper.js`. Start with `const TurboHelper = class {}` with a `constructor() {}` inside.

Now, head back to `app.js`, copy all of this code, and paste! When we did that, PhpStorm helpfully added the `import { Modal }` for us. At the bottom of this file, `export default new TurboHelper()`.

```
assets/turbo/turbo-helper.js
```

```javascript
import { Modal } from 'bootstrap';

const TurboHelper = class {
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            if (document.body.classList.contains('modal-open')) {
                const modalEl = document.querySelector('.modal');
                const modal = Modal.getInstance(modalEl);
                modalEl.classList.remove('fade');
                modal._backdrop._config.isAnimated = false;
                modal.hide();
                modal.dispose();
            }

            // internal way to see if sweetalert2 has been imported yet
            if (__webpack_modules__[require.resolveWeak('sweetalert2')]) {
                // because we know it's been imported, this will run synchronously
                import(/* webpackMode: 'weak' */'sweetalert2').then((Swal) => {
                    if (Swal.default.isVisible()) {
                        Swal.default.getPopup().style.animationDuration = '0ms'
                        Swal.default.close();
                    }
                })
            }
        });
    }
}

export default new TurboHelper();
```

This is kind of cool: it instantiates a new instance of our object and exports it. It won't really matter for us... but thanks to this, each time we import this module, we will get the same *one* instance of this object.

In `app.js`, delete all the original code and then `import './turbo/turbo-helper'`. We don't need to set that to a variable... and *just* by importing it, the object will be instantiated and the listeners will be registered. So... this should be enough to get things working!

```
assets/app.js
// ... lines 1 - 10
// start the Stimulus application
import './bootstrap';

import './turbo/turbo-helper';
```

Let's try! Refresh, click to remove an item, go back and go forward. Yep! All good.

Organizing the Class

Now that we have a class, we can organize a bit more. Copy the modal code, remove it, create a new method below called `closeModal()` and paste. Then, back up inside the `turbo:before-cache` callback, say `this.closeModal()`.

Repeat this for Sweetalert: copy all of the Sweetalert code, create a new method called `closeSweetalert()`, paste... and... then back in the callback, use it: `this.closeSweetalert()`.

assets/turbo/turbo-helper.js

```
import { Modal } from 'bootstrap';

const TurboHelper = class {
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });
    }

    closeModal() {
        if (document.body.classList.contains('modal-open')) {
            const modalEl = document.querySelector('.modal');
            const modal = Modal.getInstance(modalEl);
            modalEl.classList.remove('fade');
            modal._backdrop._config.isAnimated = false;
            modal.hide();
            modal.dispose();
        }
    }

    closeSweetalert() {
        // internal way to see if sweetalert2 has been imported yet
        if (__webpack_modules__[require.resolveWeak('sweetalert2')]) {
            // because we know it's been imported, this will run synchronously
            import(/* webpackMode: 'weak' */'sweetalert2').then((Swal) => {
                if (Swal.default.isVisible()) {
                    Swal.default.getPopup().style.animationDuration = '0ms'
                    Swal.default.close();
                }
            })
        }
    }
}

export default new TurboHelper();
```

That looks better! Let's... make sure we didn't mess anything up. Do the same dance as before: refresh, click remove, go back and go forward. All good!

Next: let's learn what types of things can go wrong when including third-party hosted JavaScript, like a JavaScript widget or analytics code. This type of JavaScript is usually supposed to be included in the *body* of the page... and often it expects full page refreshes.

Chapter 12: 3rd Party JavaScript Widgets

In a perfect world, all your JavaScript would be written in Stimulus and you would have *zero* `script` elements in your `body` tag. With that ideal setup, your JavaScript would always work - regardless of how or when new HTML was loaded - and it would only be parsed and executed one time, on initial page load.

But what about externally-hosted JavaScript? I'm talking about a third party service that you sign up for... then you're supposed to copy some JavaScript from their site, paste it onto *your* site... and suddenly you get a "feedback" button or a "share on Twitter" button... or maybe it's analytics JavaScript. These bits of JavaScript will *definitely* not be written in Stimulus and, often, funny things start to happen when you use them. Not, like "funny haha", more funny weird...

Adding an External Weather Widget

Let's see an example. Let's integrate a third-party weather widget onto our site. Head over to weatherwidget.io, which, as its name suggests, allows us to embed a handy weather widget onto our site.

Click this "get code" button. So this is pretty common: you sign up for some service and then they give you some JavaScript that you're supposed to paste onto your site.

Let's do it: copy this... then go open `templates/base.html.twig`. Head to the bottom and paste this in the footer: right before the closing `body` tag... though you could put this anywhere.

```
templates/base.html.twig
// ... lines 1 - 75
        {% block body %}{% endblock %}

        <div
            class="footer mb-0"
            {{ stimulus_controller('made-with-love') }}
        >
        </div>

        <a class="weatherwidget-io"
href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW
YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
        <script>
        !function(d,s,id){var js,fjs=d.getElementsByTagName(s)
[0];if(!d.getElementById(id))
{js=d.createElement(s);js.id=id;js.src='https://weatherwidget.io/js/widget.m
(document,'script','weatherwidget-io-js');
        </script>
    </body>
</html>
```

Cool: this gives us an `a` tag... which just says "New York weather". Then, my *guess* is that this JavaScript will execute and transform that `a` tag into the cool weather widget that you see down here.

Let's find out! Do a whole page refresh, scroll all the way down and yes! We have a weather widget! Now, navigate to another page and... it's broken! It's just the original anchor tag. Where did our cool little widget go?

What Happens When External JavaScript Executes

The JavaScript code that we pasted is pretty impossible to read. To help, select it and then go to Code -> Reformat code. There we go! It's still a little hard to read, but it's doable.

```
templates/base.html.twig
// ... lines 1 - 83
        <a class="weatherwidget-io" href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
        <script>
        !function (d, s, id) {
            var js, fjs = d.getElementsByTagName(s)[0];
            if (!d.getElementById(id)) {
                js = d.createElement(s);
                js.id = id;
                js.src = 'https://weatherwidget.io/js/widget.min.js';
                fjs.parentNode.insertBefore(js, fjs);
            }
        }(document, 'script', 'weatherwidget-io-js');
        </script>
// ... lines 96 - 98
```

This is a function that calls itself and passes in these three arguments. Basically, when this JavaScript is executed, it adds a new `script` tag to the `head` element of our page that points to this `widget.min.js` script on their site. But this function is smart: it gives the `script` tag an id set to `weatherwidget-io-js`. And before it adds the `script` tag, it checks to see if it's already on the page. If the `script` tag *does* already exist, it avoids adding it twice.

Back over at our browser, find and expand the `head` tag. Yup! There's the `script` tag with `id="weatherwidget-io-js` that points to `widget.min.js`.

So here's what's going wrong in our case. When the page first loads, like right now, this JavaScript function executes and the new `widget.min.js` script tag is added to our page. Our browser downloads that file and then, my guess is that, when that JavaScript executes, it looks for elements with a `weatherwidget-io` class on it and transforms them into the fancy weather widget.

Inspect element on this. Yup! There's the anchor tag... but now with a big `iframe` inside.

But *then*, when we navigate to another page, the entire `body` tag is replaced by a new `body` tag. The weather widget that lives inside the original `anchor` tag is now gone from the page, replaced by a new anchor tag that's just the original boring one that says "New York weather".

However, Turbo *does* see the `script` tag that's inside of the new body - the script tag that we have down at the bottom of `base.html.twig` - and it *does* re-execute these lines. But this time, since the script with id `weatherwidget-io-js` already exists up here in the `head` tag, it does not re-add it to our page. And so, no JavaScript ever runs that *re-initializes* the widget into our new anchor tag.

Add the script Element on Each Visit?

Okay, so now that we understand what's going on, shouldn't we just, you know, tweak the JavaScript so that it *always* inserts the `script` tag? Let's try it. I'll cheat and temporarily add `|| true` to the if statement so that it always executes and adds that element.

All right. Refresh. On page one, the weather widget works. Click over to the cart and... yea! The weather widget *still* works! Problem solved! And don't worry, the `script` tag isn't downloaded multiple times: your browser is smart enough to pull it from cache after it downloads it the first time.

Having Many Duplicate script Tags on your Page

But... this might not be the best solution for two reasons. Look at the `head` element of our page. Woh! We have two script tag!. And each time we navigate, we would get yet *another* one.

That... might be ok? But it seems a bit crazy: eventually a user might have 50 identical `script` elements on their page.

And actually, that's precisely how some external JavaScript works. Some external JavaScript snippets do *not* have this if statement here. And so, one of the problems is that it *does* add more and more and more script tags when you using Turbo.

The second problem is that... whether or not executing this script file over and over again is a good idea... sort of depends on what that `script` tag does! If it simply reinitialize the weather widget, cool! That sounds safe. But if it, for example, adds an event listener to the document each time it's executed, then each time we load that script tag, we're going to add a second, third, fourth, or fifth listener. Then, suddenly when you, for example, click the page, that JavaScript widget's listener will execute 5 times and... do whatever it normally does *way* more times than normal.

My point is: you need to be careful with third-party JavaScript. Let's put back the if statement the way we found it.

So in this case, re-executing the `widget.min.js` script tag after each visit is probably okay: it *does* seem to simply reinitialize the weather widget on this element. But I would *love* to do that without duplicating the `script` tag and ending up with 50 of them in my `head`. How can we do that? By removing the previous `script` tag right before the page renders. And how can we do *that*? Via a new event listener. Let's talk about that next and discuss the proper way to handle analytics code so that you don't under-count or *over* count your visits.

Chapter 13: Fixing External JS + Analytics Code

Head back to the Turbo docs, specifically to Reference and then Events. We saw this list of events earlier. Now we're going to hook into a *new* one: `turbo:before-render`.

The turbo:before-render Event

Here it is. This event triggers before Turbo renders a page, but *not* counting the initial page load. In other words, it triggers when Turbo is *specifically* responsible for rendering the page. We can use this to help our third party weather widget get working right before the page renders.

Head over to `assets/turbo/turbo-helper.js` and, up here in the `constructor`... say `document.addEventListener()` to listen to `turbo:before-render`. Pass this an arrow function and then log "before render" so we can see *exactly* when this does and doesn't execute.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:before-render', () => {
            console.log('before render!');
        });
    }
// ... lines 14 - 41
```

Cool. Let's test it!

Find your browser, refresh, and open the console. Okay. So nothing on initial page load. But then, when we click to another page, there it is! Click to another page... there's a second one. Click to the homepage, a third one. Awesome.

Now, clear out the console... and go back to a page we went to a second ago. It logs twice! This is an important detail about this event. It fired twice because *first* the preview was rendered and *then* the final page was rendered. Just keep that fact in mind.

Removing the Weather Script Tag Before Render

So here's the plan: right before the page is rendered, so inside of our new listener, we're going to find and remove this `weatherwidget-io-js` script tag. Then, with any luck, when the new page is loaded, the JavaScript from our base template will execute, it will re-add that script tag and everything will work!

Let's check it! Replace the log with `document.querySelector()` and look for `#weatherwidget-io-js`. Then say, `.remove()`. You can also code defensively to make sure the element *exists* first before trying to call `remove()`... not a bad idea.

assets/turbo/turbo-helper.js

```
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:before-render', () => {
            document.querySelector('#weatherwidget-io-js').remove();
        });
    }
// ... lines 14 - 41
```

Ok: refresh. It works and... navigate to a different page. Yea! It *still* works! If you look inside the `head` element, it accomplishes this *without* duplicating the `script` tag.

Calling the External Script Directly on Navigation

I like this solution. But if you're willing to do some digging, there *might* be an alternate solution.

Copy the `widget.min.js` URL and open it in your browser. It's minified... so pretty unreadable. Copy the source, close it, spin over to your editor and create a new file anywhere, like `pizza.js`... we're not going to actually use this. Paste the code, select it, then go back up to Code -> Reformat Code so we can at least, *kind of* read it.

It's still not super clear, but... let's see. Ah! There's a function called `__weatherwidget_init`... and it looks like *this* might be the key to re-initializing the weather widget! In other words, instead of removing and re-adding the `script` tag on each render, we might be able to just... call this function!

The turbo:render Event

Let's do some experimenting! Start by changing the event from `turbo:before-render` to `turbo:render`... that's another new event. Why are we switching to it? In order for the `__weatherwidget_init` function to work, the new `weatherwidget-io` anchor tag needs to actually live *on* the page.

But `turbo:before-render` is triggered too early: it's triggered *before* the new body is on the page. Fortunately, `turbo:render` is called *after* it's on the page. This means that, inside of the callback, we know that the *new* body *will* be on the page. And so, we can call that `__weatherwidget_init` function. Let me steal that name from the other file... and paste it here.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            __weatherwidget_init();
        });
    }
// ... lines 14 - 41
```

Testing time! Refresh! The first page works: no surprise. And when we go to a second page... yes! It still works! No matter how many pages we go to, it keeps working. I like this solution better, though, I also realize that we're sort of using an "internal" function from that widget script... and it's possible they could change their JavaScript some time in the future.

Now that we have this working, let's refactor this logic into a method for clarity. Copy the `__weatherwidget_init` function, go to the bottom of the class and create a new method, how about `initializeWeatherWidget`. Paste, then call that from up here in our listener: `this.initializeWeatherWidget()`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            this.initializeWeatherWidget();
        });
    }
// ... lines 14 - 38
    initializeWeatherWidget() {
        __weatherwidget_init();
    }
// ... lines 42 - 45
```

Solving External Widgets with Stimulus?

By the way, there *is* a third way to solve this problem, and we'll talk about it later. It's especially appropriate if you need to load an external widget - like our weather widget - but that widget might be loaded onto the page at any time, even via a custom, non-Turbo Drive Ajax call. This solution basically involves running the same code that we have here, but leveraging a Stimulus controller.

Handling Analytics Code

Before we move on, we do need to talk about *one* last type of external JavaScript: analytics code. As an example, here's what Google analytics code looks like: this is what you're supposed to paste into the `head` tag of your page.

It turns out that the key line that triggers the visit is this last one: `gtag('config')`. If we pasted all of this onto our site, guess what would happen? It would register the first visit... then the code would *never* execute again, no matter how many pages the user visited. That's not great. Fortunately, single page applications - like those written in Vue or React - have the same problem.... and you can often find docs that talk about how to integrate with *those*.

In this case, the solution would be to paste all this code - *except* for the `gtag('config')` line - into your `head` like normal. For this last line, we need to execute it on initial page load and then every Turbo "visit" after.

The turbo:load Event & Analytics

Let me open a GitHub issue that talks about this with a really nice solution. As you can see here, `henrik` is using a `turbo:load` event. That's yet *another* event that we haven't talked about yet. `turbo:load` is nice because it's executed on initial page load and *one* time for every visit: it avoids the "double dispatch" that happens with the `turbo:before-render` and `turbo:render` events when you visit a page that shows a preview. In other words, `turbo:load` is triggered *exactly* when you would want your analytics code to trigger a visit.

Inside the callback, `henrik` calls `gtag('config')` to trigger that visit. This `googleAnalyticsIDForScript` thing is just their way of referencing whatever your Google Analytics ID is. The one special thing that you need to do with this function is pass a little bit of extra data to make sure analytics knows what the actual URL is that it should use.

Next: we already know that, with Turbo Drive, we download each CSS and JavaScript file just *one* time. Then, as we navigate around, if Turbo sees a CSS or JS file in the new page's `head` tag that already exists on the *current* page, it ignores it.

But what happens if we deploy a new version of our site and the content of these files has changed? How can we force the user to download the newest version of our assets? That's an important question.... and one where the answer is refreshingly simple.

Chapter 14: Reloading When JS/CSS Changes

How does Turbo handle when a JavaScript or CSS file that's downloaded onto our page changes? When we navigate, it's smart enough to merge any new CSS or JS into our `head` element without *duplicating* anything that's already there.

But what about a CSS or JavaScript file whose contents just updated because we deployed? This is really a problem specific to production because locally, if we change a CSS or JS file in our editor, we just come back and trigger a full page reload manually. But how is this handled on production?

Well... if you do nothing, it's pretty simple: your users will continue to surf around with the old CSS and JavaScript... which is *not* something we want... especially since they will be getting the *newest* HTML from our site... which may only work with the newest CSS and JavaScript.

Activating Asset Versioning

But a slightly different thing happens if we enable *versioning* on our assets. Head to your editor and open up `webpack.config.js`. About halfway down this file... you'll find `enableVersioning()`.

This tells Encore that, if we are doing a production build, each filename should contain a hash that's unique to its contents. It's a great strategy to make sure that when you deploy updates, each file gets a new file name... which forces users - in a *non* Turbo universe - to download the latest version.

To see what happens *with* Turbo, let's activate this for dev builds also by removing the `Encore.isProduction()` argument.

webpack.config.js

```
// ... lines 1 - 44
    // enables hashed filenames (e.g. app.abc123.css)
    .enableVersioning()
// ... lines 47 - 76
```

To make this take effect, find your terminal, go to the tab that's running Encore, hit Control+C and then rerun:

```
yarn watch
```

When that finishes... move over, refresh, and navigate around. If you check out the `head` tag, we have versioned filenames! The `app.css` file is now `app.blahblah.css`, and the `app.js` file also has a hash.

Let's go modify the `app.js` file - that's over at `assets/app.js`. At the bottom, `console.log('new code')`.

```
assets/app.js
// ... lines 1 - 14

console.log('new code!');
```

Now, *without* refreshing your browser, navigate to a new page.. and look at the console. Interesting... no log! And we have *two* `app.js` script tags on the page... which is probably not what we want.

First, the new file wasn't executed because Webpack was smart enough to realize that the `app` entry script *has* already been loaded. So even though the `script` tag *was* added... and downloaded, Webpack prevented it from running: it knows that something weird is going on.

And even if it *did* load, it would probably mean that we would have things like event listeners registered twice on the page... which is *also* not what we want.

What we see in the `head` tag at least *does* make sense based on what we know about Turbo. Because the `app.js` has a new filename, it looks like a *new* script file. And so, Turbo added it to the `head`.

Refreshing the Page with data-turbo-track="reload"

So... how do we fix this mess? Well, let's think. One of the huge benefits of Turbo is that your JavaScript and CSS are downloaded and executed just *once* on initial page load... and then are reused for every navigation after. It's a big reason why Turbo is so fast. But if one of these files changes... we sort of *do* need to hit the "reset" button. In other words, this is *one* case when the page *should* do a full page reload so that our browser can download everything new.

Fortunately, there's an easy way to do this: by adding a special `data-turbo-track` attribute to every CSS and JS tag. And, it turns out, adding that attribute is super easy!

Open `config/packages/webpack_encore.yaml`. The `script_attributes` key allows us to add an attribute to *every* `script` tag that Encore outputs. Add `data-turbo-track` and set it to `reload`. We'll talk about what this does in a second. Also uncomment `link_attributes` and set the same thing here.

```yaml
config/packages/webpack_encore.yaml
// ... lines 1 - 6
    # Set attributes that will be rendered on all script and link tags
    script_attributes:
        defer: true
        'data-turbo-track': reload
    link_attributes:
        'data-turbo-track': reload
// ... lines 13 - 33
```

With this simple change, every `script` and `link` tag that Encore renders will now have that `data-turbo-track="reload"` attribute on it.

So here's how this works... it's pretty simple. When we navigate, Turbo finds all of the elements with `data-turbo-track` on the *current* page and compares their filenames to the `data-turbo-track` elements on the *new* page. If the total collection of filenames on the old page does *not* exactly match the total collection of filenames on the *new* page, Turbo will trigger a full page reload.

Watch: if we click around, we see a lot of nice, boring Turbo-powered visits. But now go back to `assets/app.js` and remove that `console.log()`.

Behind the scenes, a new `app.js` file with a *new* filename was just output. You can see it in the Encore terminal: before the filename was this, now the filename is different.

Back at our browser, let's visit a new page. Watch carefully. Yes! That was a full page reload! Turbo saw that the new page's "tracked" `script` and `link` tag filenames did *not* exactly match the old page's tracked filenames, and so, it triggered a normal, full-page-reload navigation. Problem solved!

Next: sometimes you may need to navigate to another page via custom JavaScript code. Like, maybe you have some custom JavaScript... where, after an Ajax call, you want to redirect to another URL. Could we use Turbo to do that visit instead of triggering a full page reload? Absolutely.

Chapter 15: Manual Visits with Turbo

Sometimes you need to trigger a Turbo visit programmatically... like after running some custom JavaScript, you want to send the user to another page.

Head over to your code and open `assets/controllers/counter_controller.js`. This very advanced Stimulus controller powers this high-tech "click for a chance to win" area. Each time I click the button, the counter goes up. Amazing!

Let's pretend that, after 10 clicks, the user wins and we want to redirect them to a "you won!" page. Let's first do this with normal JavaScript. Inside of the `increment()` method - which is called each time we click - say, if `this.count` equals `10`, then redirect using raw JavaScript: `window.location.href` equals `/you-won`, which is a page I already created.

```
assets/controllers/counter_controller.js
// ... lines 1 - 6
    increment() {
        this.count++;
        this.countTarget.innerText = this.count;

        if (this.count === 10) {
            window.location.href = '/you-won';
        }
    }
// ... lines 15 - 16
```

Let's make sure this works. Refresh the homepage... click a bunch of times and... eureka! We're winners! But... that worked via a full page refresh, *not* via Turbo.

Navigating with Turbo

Could we navigate *with* Turbo? Totally! Start by importing Turbo into this file. This is the most complicated part because... the syntax looks a little funny. It's `import * as Turbo from` and then the name of the library, which is `@hotwired/turbo`. The `* as Turbo` is needed due to how that library exports things.

Down in the method, instead of `window.location.href`, we can say `Turbo.visit()` and pass in the URL.

```
assets/controllers/counter_controller.js
```

```
import { Controller } from 'stimulus';
import * as Turbo from '@hotwired/turbo';

export default class extends Controller {
    count = 0;
    static targets = ['count'];

    increment() {
        this.count++;
        this.countTarget.innerText = this.count;

        if (this.count === 10) {
            Turbo.visit('/you-won');
        }
    }
}
```

Let's try it again! Go back to the homepage and do a full page refresh. Actually... it did a full page refresh automatically because of the asset tracking we created in the last chapter. Cool!

Time to click! Watch when we get to 10. Beautiful! That navigated with Turbo. We can see the Ajax call right here. And... yea! It's just that easy.

But if you want to be more hipster, you can use de-structuring to *just* import the `visit` function. It looks like this `import { visit } from '@hotwired/turbo'`. Then below, literally call `visit()` as a function.

assets/controllers/counter_controller.js

```
import { Controller } from 'stimulus';
import { visit } from '@hotwired/turbo';

export default class extends Controller {
    count = 0;
    static targets = ['count'];

    increment() {
        this.count++;
        this.countTarget.innerText = this.count;

        if (this.count === 10) {
            visit('/you-won');
        }
    }
}
```

This will work exactly the same as before.

What if Turbo isn't Available?

There's one other tricky situation that you might run into when it comes to navigating with Turbo: if you're writing JavaScript... but you are *not* in a file that's parsed by Webpack. In other words, you're somewhere where you *can't* use the `import` keyword.

This is probably not very common and, really, in a perfect world, 100% of our JavaScript *will* be written in a Webpack-parsed file.

But just in case, let's see how we can navigate with Turbo from inside some *inline* JavaScript on our page. Open up `templates/base.html.twig` and head to the bottom. Right before the closing `</body>`, add a `<script>` tag. We're going to pretend that when we click the logo... which has `id="logo-img"`... that we want to go to the cart page.

Do that by saying `document.getElementById()`, pass it, `logo-img`, `.addEventListener('click')` and pass an arrow function with an `event` argument. Inside, say `event.preventDefault()` so that it doesn't follow the link that the image is inside of. Oh... yikes! I forgot my comma. That's better.

How can we fetch the Turbo object to trigger the visit? It turns out... it's already available as a global variable! So we can immediately say: `Turbo.visit('/cart')`

```
templates/base.html.twig
// ... lines 1 - 96
        <script>
        document.getElementById('logo-img').addEventListener('click',
(event) => {
            event.preventDefault();
            Turbo.visit('/cart');
        })
        </script>
// ... lines 103 - 105
```

That's it! But... who *set* Turbo as a global object? I don't remember doing that! Starting in Turbo 7 beta 6, when you import the `@hotwired/turbo` library, it automatically sets itself as a global variable. So if you have Turbo working on your site, there *is* a `Turbo` global variable, which is done to help with this exact situation.

Anyways, if we go and do a full page refresh... then click the logo image, instead of going to the homepage like it normally would, it navigates us - via Turbo - to the cart page.

Next, we are now done with all the Turbo Drive tricky parts! Before we move onto Turbo frames, let's try doing a few fun things. The first will be to experiment with adding CSS transitions as we navigate between pages with Drive.

Chapter 16: CSS Page Transitions

What about CSS transitions between pages as we click around? This is something that a competing library called Swup does very well. But in Turbo, it's not so easy. Well, it *will* be easier once a PR is merged into Turbo.

Here's the basic problem: when you click, Turbo makes an Ajax call for the new HTML. Then, when that Ajax call finishes, it *immediately* puts the new body onto the page. To be able to have a CSS transition between visits, we need a way to *pause* that process. When the Turbo Ajax call finishes, we need to be able to tell Turbo to *not* immediately render the new page so that we can *instead* start a CSS transition - like fading out the old page. Then, once that transition finishes, we tell Turbo to *finally* finish its job of putting in the new body.

The missing piece right now, which the pull request addresses, and which *has* gotten a thumbs up from the maintainers, is that there's no ability to pause that process. If you're interested in complex CSS transitions, keep an eye on this issue.

Does this mean that we can't add *any* transitions? Actually, no! It just means we can't create super-precise animations. For example, imagine that we want to slide the old content up, wait for that transition to finish, then immediately slide the new content down. That's *not* going to work until we have more control over the process.

But if we just want to fade out the old page and fade *in* the new page, that *will* work. Why? Because if the fade out doesn't *quite* finish before the fade in starts... that's probably not a huge deal. It's a little imprecise, but it will still look good. So even though we can't add *perfect* CSS transitions yet, let's learn how to do this. It's a fascinating example of the power of Turbo events.

So here's the plan: at various times while the old page is leaving and the new page is entering, we're going to add some CSS classes that allow us to cause those to fade out and fade in.

Adding the Transition CSS

Let's actually start with the CSS. Open up `assets/styles/app.css`. Right on top inside `body`, add `transition: opacity 1000ms`.

Two things about this. First, 1000 milliseconds is *way* too long for a transition, but it'll make this easy for us to see while we're developing. Second, if you're new to CSS transitions, this line doesn't *cause* a transition. It just says that *if* the opacity of the body ever changes, I want it to change gradually over one second, instead of immediately.

Below this, add `body.turbo-loading`. Inside, set the opacity to `.2`... which is probably too low of an opacity for a nice effect... but again, it'll make it easy for us to see.

This `turbo-loading` class is *not* something that's part of Turbo: it's something that *we* are going to add to cause the transition.

```
assets/styles/app.css
// ... lines 1 - 4
body {
    font-family: 'Montserrat', sans-serif;
    transition: opacity 1000ms;
}
body.turbo-loading, body.turbo-loaded {
    opacity: .2;
}
// ... lines 12 - 173
```

Triggering the Fade Out Transition

Let's do it. Go back to `assets/turbo/turbo-helper.js` and, in the constructor, here we are, add a new event listener at the bottom. Step one is, when we click a link, we want to add the `turbo-loading` class to the `<body>`. That will cause the old body to fade out.

Do that with `document.addEventListener()` and, this time, listen to an event called `turbo:visit`. This is yet *another* event that we haven't seen before. This is triggered immediately after a visit *starts*. Inside, say `document.body` - that's an easy way to get the `body` element - then `.classList.add('turbo-loading')`. I'll add a comment that explains what this does.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            this.initializeWeatherWidget();
        });

        document.addEventListener('turbo:visit', () => {
            // fade out the old body
            document.body.classList.add('turbo-loading');
        });
// ... lines 18 - 61
```

To make it easy to see if this is working, go to `public/index.php`... and add a 1 second `sleep()` temporarily.

```
public/index.php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';
sleep(1);

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

Ok: let's go refresh the page... this will be kind of slow. Ready? Click! Nice! The page faded out. But then the new content shows up immediately. We haven't added the fade *in* effect yet.

Triggering the Fade In Transition

Let's do *that*. Head back to `turbo-helper.js`. I'm going to paste in two more listener functions. Let's walk through this: we've seen both of these events before.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 17
        document.addEventListener('turbo:before-render', (event) => {
            // when we are *about* to render, start us faded out
            event.detail.newBody.classList.add('turbo-loading');
        });
        document.addEventListener('turbo:render', () => {
            // after rendering, we first allow the turbo-loading class to set the low opacity
            // THEN, one frame later, we remove the turbo-loading class, which allows the fade in
            requestAnimationFrame(() => {
                document.body.classList.remove('turbo-loading');
            });
        });
// ... lines 29 - 61
```

`turbo:before-render` fires right *before* the new body is added to the page. This allows us to add the `turbo-loading` class to the *new* body before it's added to the page. This will set its opacity to `.2` to start: we want it to *start* faded out.

Then the `turbo:render` event is triggered right *after* that new body is added to the page. Here, we want to *remove* the `turbo-loading` class. That will set the opacity *back* to 1... and thanks to the transition, it should happen slowly over 1 second.

But we can't remove the class immediately... we can't just put this line directly here in the listener. Why not? We need the new body to be rendered for at least 1 "frame" with the lower opacity... so with the `turbo-loading` class. If we remove it immediately - by just putting the line right here - the element will actually start at *full* opacity with *no* transition... because it never got the chance to render with the *low* opacity.

This is why we have this `requestAnimationFrame()` function. This is a built-in browser function that says:

> *"Hey, once you do render the next frame, please call this function."*

This allows the element to be rendered for one frame with the low capacity... and *then* we remove the class to transition to full opacity. Pretty freaking cool.

Let's try it. Refresh, and... click. Yes! The fade out and fade in transition looks perfect! Yay! Until... we visit a page we've already been to. Woh. That was weird. It... sort of faded in and... then faded in again?

Let's find out what's going on next and use more Turbo smartness to fix it. By the end, we are going to have *perfect* fade transitions.

Chapter 17: Polished CSS Transitions

The fade-out and fade-in transition works... until you visit a page that you've already been to... then things get weird. Instead of fading out, it, sort of, fades in... then fades in again?

This happens because, back over in `turbo-helper.js`, both `turbo:before-render` and `turbo:render` happen when both a real page renders *and* when a preview renders.

```
assets/turbo/turbo-helper.js
```

```javascript
// ... lines 1 - 2
const TurboHelper = class {
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            this.initializeWeatherWidget();
        });

        document.addEventListener('turbo:visit', () => {
            // fade out the old body
            document.body.classList.add('turbo-loading');
        });
        document.addEventListener('turbo:before-render', (event) => {
            // when we are *about* to render, start us faded out
            event.detail.newBody.classList.add('turbo-loading');
        });
        document.addEventListener('turbo:render', () => {
            // after rendering, we first allow the .turbo-loaded to set the low opacity
            // THEN, 10ms later, we remove the turbo-loaded class, which allows the fade in
            setTimeout(() => {
                document.body.classList.remove('turbo-loading');
            }, 10);
        });
    }
// ... lines 30 - 61
```

That means that, when a preview is shown, it gets the same transition effect as a real page. When we click a page we've previously been to, the preview instantly shows - starting faded out - and then fades in. When the Ajax call finishes for the real page, that *also* starts faded out and then fades in.

Tricky, eh? The solution is to detect if what's rendering is a preview and then do something different. Specifically, if we *are* rendering a preview, we want to start with *full* opacity and then fade out, so that we get the same effect as a normal visit.

Detecting if a Preview is Rendering

How do we detect if what's rendering is a preview? By looking for the `data-turbo-preview` attribute on the `html` element. Watch, if we go to back to a *previous* page, watch the `html` tag. Yup! It has a `data-turbo-preview` attribute while it's showing.

Back in `turbo-helper`, start by going all the way to the bottom and creating a new method called `isPreviewRendered()`. Inside, return `document.documentElement` - that's how you get the HTML tag - `.hasAttribute('data-turbo-preview')`.

assets/turbo/turbo-helper.js

```
// ... lines 1 - 73

    isPreviewRendered() {
        return document.documentElement.hasAttribute('data-turbo-preview');
    }
}
// ... lines 79 - 81
```

We're using `hasAttribute` instead of `dataset` because we don't care what the *value* is - it would be an empty string - we just care whether or not it *exists*.

Copy that method name and head back up to our listeners. Start with `before:render`: if `this.isPreviewRendered()`... then do nothing for the moment... but in the else, do the normal logic.

Before we add the preview logic, I need to mention that this *can* be confusing. Because we're inside of `before:render`, if a preview is being rendered, then it hasn't *actually* been rendered onto the page yet. Even though that's true, the current page *will* already have the `data-turbo-preview` attribute on it, which means we *can* use our `isPreviewRendered()` function to figure out if what we're *about* to render is a preview.

Anyways, *if* this is a preview, we want to remove the `turbo-loading` class so that the preview starts at *full* opacity. Then, one frame later, we want to re-add that class to cause the preview to fade out... because, once the new Ajax call finishes, the real page will fade *in*.

Copy the code from below, paste, but *remove* the class. Then, steal the `requestAnimationFrame()` code, paste *that*... and grab the `classList.add()` from below and use that exactly.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 18
        document.addEventListener('turbo:before-render', (event) => {
            if (this.isPreviewRendered()) {
                // this is a preview that has been instantly swapped
                // remove .turbo-loading so the preview starts fully opaque
                event.detail.newBody.classList.remove('turbo-loading');
                // start fading out 1 frame later after opacity starts full
                requestAnimationFrame(() => {
                    document.body.classList.add('turbo-loading');
                });
            } else {
                // when we are *about* to render a fresh page
                // we should already be faded out, so start us faded out
                event.detail.newBody.classList.add('turbo-loading');
            }
        });
// ... lines 34 - 81
```

Perfect! So this will remove `turbo-loading` from the new body, then, one frame later, re-add it to cause the fade out.

Now, in `turbo:render`, we only want to remove the `turbo-loading` class if this is *not* a preview. So if not `this.isPreviewRendered()`, then remove that `turbo-loading` class.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 33
        document.addEventListener('turbo:render', () => {
            if (!this.isPreviewRendered()) {
                // if this is a preview, then we do nothing: stay faded out
                // after rendering the REAL page, we first allow the .turbo-loading to
                // instantly start the page at lower opacity. THEN remove the class
                // one frame later, which allows the fade in
                requestAnimationFrame(() => {
                    document.body.classList.remove('turbo-loading');
                });
            }
        });
// ... lines 45 - 81
```

Yes, I know, it's *pretty* complex. Let's take it for a test drive. Do a full page refresh. If we click to new pages... this all still looks fine. And if we click to a previous page... yes! That did it! The preview instantly shows, fades out, then the new page fades in.

Restore Visits: No Transitions

But... there's one last edge case. Click the "back" button in your browser. Hmm. It instantly goes to low opacity and then fades in. Not terrible... but a little odd. This happens because the snapshot of every page is taken right *before* the *new* page is "swapped in". Thanks to our new fade out functionality... it means that snapshots are taken when the page has the `turbo-loading` class! In other words, snapshots are taken when the page has *low* opacity! Thanks to this, when the snapshot is restored, it has low opacity. Once the class is removed by our listener code, it fades in.

For me, since clicking back and forward loads instantly, I'd prefer to not have *any* CSS transition.

How can we do that? When you click back or forward like this, even though it's pulling the page from the snapshot cache, it is *not* considered a "preview". And so the `isPreviewRendered()` returns false. *That* means that we're down in this case. Here, *if* this is a "restoration" visit - that's what it's called when you click the back or forward buttons in your browser - then we want the new page to *start* with full opacity and *not* have a transition.

Check it out: say `const isRestoration` equals `event.detail.newBody.classList.contains('turbo-loading')`.

That... probably looks a bit confusing. Because of the transition system we just built, every page snapshot will have a `turbo-loading` class. Since we know this is *not* a preview, if the body has the `turbo-loading` class, then this *must* be a snapshot that's being used for a *restoration* visit. And *if* it's a restoration visit, say `event.detail.newBody.classList.remove('turbo-loading')`. I'll add a note above explaining this. Oh, duh, sorry - this probably looks super confusing because I forgot to wrap this in an if `isRestoration`. *If* this is a restoration, remove that class and return.

assets/turbo/turbo-helper.js

```
// ... lines 1 - 18
        document.addEventListener('turbo:before-render', (event) => {
            if (this.isPreviewRendered()) {
                // this is a preview that has been instantly swapped
                // remove .turbo-loading so the preview starts fully opaque
                event.detail.newBody.classList.remove('turbo-loading');
                // start fading out 1 frame later after opacity starts full
                requestAnimationFrame(() => {
                    document.body.classList.add('turbo-loading');
                });
            } else {
                const isRestoration = event.detail.newBody.classList.contains('turbo-loading');
                if (isRestoration) {
                    // this is a restoration (back button). Remove the class
                    // so it simply starts with full opacity

                    event.detail.newBody.classList.remove('turbo-loading');

                    return;
                }

                // when we are *about* to render a fresh page
                // we should already be faded out, so start us faded out
                event.detail.newBody.classList.add('turbo-loading');
            }
        });
// ... lines 44 - 91
```

This will cause the page to start with full capacity and never change.

Phew! Okay, let's make sure this helps. Head back, refresh, click around to a new page, another new page, click to a previous page, and now hit back. Got it! Back and forth show instantly.

Yup, this *is* tricky. My hope is that CSS transitions will be easier in the future with Turbo. It *is* doable now, but you *do* need to keep track of several things.

Organizing our Logic

Before we keep going, let's isolate all of this logic - which is getting kind of big - into its own method. Copy both `document.addEventListener()` sections, remove them, go down to the bottom, and create a new method called `initializeTransitions()`. Paste all that logic there, head back up to the constructor and call it: `this.initializeTransitions()`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            this.initializeWeatherWidget();
        });

        this.initializeTransitions();
    }
// ... lines 16 - 47

    initializeTransitions() {
        document.addEventListener('turbo:visit', () => {
            // fade out the old body
            document.body.classList.add('turbo-loading');
        });

        document.addEventListener('turbo:before-render', (event) => {
            if (this.isPreviewRendered()) {
                // this is a preview that has been instantly swapped
                // remove .turbo-loading so the preview starts fully opaque
                event.detail.newBody.classList.remove('turbo-loading');
                // start fading out 1 frame later after opacity starts full
                requestAnimationFrame(() => {
                    document.body.classList.add('turbo-loading');
                });
            } else {
                const isRestoration = event.detail.newBody.classList.contains('turbo-loading');
                if (isRestoration) {
                    // this is a restoration (back button). Remove the class
                    // so it simply starts with full opacity

                    event.detail.newBody.classList.remove('turbo-loading');

                    return;
                }

                // when we are *about* to render a fresh page
                // we should already be faded out, so start us faded out
                event.detail.newBody.classList.add('turbo-loading');
            }
        });
        document.addEventListener('turbo:render', () => {
            if (!this.isPreviewRendered()) {
                // if this is a preview, then we do nothing: stay faded out
                // after rendering the REAL page, we first allow the .turbo-loading to
                // instantly start the page at lower opacity. THEN remove the class
                // one frame later, which allows the fade in
                requestAnimationFrame(() => {
                    document.body.classList.remove('turbo-loading');
                });
            }
        });
    }
// ... lines 92 - 95
```

This at least gives all this code down here a name so that future "us" can better remember what it does.

Oh, and while we're cleaning things up, don't forget to take the `sleep` out of `public/index.php` ... and inside of `styles/app.css`, change the transition to something more realistic, like 200 milliseconds. Also change the opacity to something less extreme, like .8.

Let's see what this - more "real-world" - setup looks like. The reload is faster and the transition is... a nice, subtle effect! If we click to a preview page, that's good... and hitting back also works.

Next: let's try something kind of crazy. What if, when a user hovers over a link, we prefetch that URL so that Turbo can display it even *faster*. This little trick - which is *super* fun with Turbo - can actually be used to speed up *any* site.

Chapter 18: Prefetching the Next Page

I have a crazy idea. What if, when the user hovers over a link, we *prefetch* that page via Ajax and saved it to the snapshot cache? Then, assuming the user *does* click that link, Turbo would show the page *instantly* via its preview system.

Is that possible? Well, not *officially*. But thanks to some clever people on the Internet, it is! Let's learn two different ways that can we can make the performance of our site *even* faster... and the caveats that go with both - neither is perfect out-of-the box. But both are *super* interesting.

Prefetching on Hover

If you downloaded the course code, you should have a `tutorial/` directory with a `prefetch.js` file inside. Copy that and paste it into `assets/turbo/`.

```
assets/turbo/prefetch.js
// https://gist.github.com/vitobotta/8ac3c6f65633b5edb2949aeff0dec69b

// This code is to be used with https://turbo.hotwire.dev. By default Turbo keeps visited pages in its cache
// so that when you visit one of those pages again, Turbo will fetch the copy from cache first and present that to the user, then
// it will fetch the updated page from the server and replace the preview. This makes for a much more responsive navigation
// between pages. We can improve this further with the code in this file. It enables automatic prefetching of a page when you
// hover with the mouse on a link or touch it on a mobile device. There is a delay between the mouseover event and the click
// event, so with this trick the page is already being fetched before the click happens, speeding up also the first
// view of a page not yet in cache. When the page has been prefetched it is then added to Turbo's cache so it's available for
// the next visit during the same session. Turbo's default behavior plus this trick make for much more responsive UIs (non SPA).

import * as Turbo from '@hotwired/turbo';

let lastTouchTimestamp
let delayOnHover = 65
let mouseoverTimer

const pendingPrefetches = new Set()

const eventListenersOptions = {
    capture: true,
    passive: true,
}

class Snapshot extends Turbo.navigator.view.snapshot.constructor {
}

// ... lines 28 - 153
```

Ok: this is *not* my script: it comes from a gist that I attributed on top. This script automatically makes an Ajax call whenever a user hovers over an anchor tag and saves the response as a Turbo snapshot. Then, if the user *does* click that link, the page will be displayed *instantly* thanks to the preview. To avoid *totally* spamming the server with requests, this code waits for the user to hover for 65 milliseconds before sending the Ajax request. The idea is to take advantage of the brief pause between when a user starts to hover over a link and when they actually *click* that link. This approach does have some downsides, but let's see it in action before we chat about them.

Open up `app.js` and import this: `import './turbo/prefetch'`. That's enough to activate the new behavior.

```
assets/app.js
// ... lines 1 - 12

import './turbo/turbo-helper';
import './turbo/prefetch';
```

Also open up `styles/app.css` and comment-out the `opacity` transition that we added before. The pages are going to be *so* fast that this won't be needed.

```
assets/styles/app.css
// ... lines 1 - 8
/*
body.turbo-loading {
    opacity: .8;
}
*/
/*
[data-turbo-preview] body {
    opacity: .2;
}
*/
// ... lines 19 - 175
```

Moment of truth. At your browser, refresh. I'm going "casually" click on the Furniture category. Woh - that was fast! All these pages are now loading as *if* we've already visited them... because... we actually have! The perceived performance of our site just took another huge step forward.

The Downsides of the Hover Prefetch

But that was *too* easy! So what are the downsides? There are a few. The first is that your site is going to get hit by a lot more requests. If you hover over a link but never click it, that's an extra, unnecessary request! But worse, even if you *do* click the link, *two* requests are made! Watch, I'll refresh, then clear my network tools. Hover over "Office Supplies", then click. Check it out: *two* requests were made for the same page! The prefetch script made the first request to store the page as a snapshot for the preview. But then, like normal preview functionality, after showing the preview, Turbo made a *second* request to load a "fresh" version of the page. That's a bummer.

Another downside is that, if your page doesn't load fast enough, this won't make any difference! For example, let me clear the network tools again. I'm going to hover and then click "Breakroom" *really* fast. Watch: that time, the page did *not* load instantly because the first prefetch request had *not* finished by the time I clicked.

In fact, when you look at the second request that Turbo made, it "stalled": the second request waited for the first. To be fully honest, I'm not actually sure *why* my browser waits like this... but it means that if the user clicks before the prefetch request finishes, it may actually be slowing *down* the experience.

The *last* problem is that the prefetch script will also try to prefetch links that we don't want it to - like a "log out" link. Yup, right now, if we hovered briefly over a log out link, that... would log us out.

In the script, search for `dataset`. You *can* add a `data-prefetch="false"` attribute to any link to disable the behavior for that link. Or, by customizing this line a little, you could *disable* the prefetch behavior by default and only *enable* it if the link has `data-prefetch=true`. That would be a safe way to enable this only on links that make sense to you.

assets/turbo/prefetch.js

```
// ... lines 1 - 91

function isPreloadable(linkElement) {
    if (!linkElement || !linkElement.getAttribute("href") || linkElement.dataset.turbo == "false" || linkElement.dataset.prefetch == "false") {
        return
    }

// ... lines 98 - 153
```

The "prefetch" link Hint

There's also another way to use this script, which you can see at the bottom. If you add a `data-prefetch-with-link="true"` attribute, instead of making an Ajax call, it will add a `<link rel="prefetch">` element to your `head` tag.

```
assets/turbo/prefetch.js
// ... lines 1 - 133
function preload(link) {
    const url = link.getAttribute("href")
    const loc = new URL(url, location.protocol + "//" + location.host)
    const absoluteUrl = loc.toString()

    if (link.dataset.prefetchWithLink == "true") {
        const prefetcher = document.createElement('link')
        prefetcher.rel = 'prefetch'
        prefetcher.href = url
        document.head.appendChild(prefetcher)
        pendingPrefetches.delete(absoluteUrl)
    } else if (!Turbo.navigator.view.snapshotCache.has(loc)) {
        fetchPage(url, responseText => {
            const snapshot = Snapshot.fromHTMLString(responseText)
            Turbo.navigator.view.snapshotCache.put(loc, snapshot)
            pendingPrefetches.delete(absoluteUrl)
        })
    }
}
```

What does that do? It enables a really neat feature that's native to your browser. Let's learn about it next.

Chapter 19: <link rel="prefetch">

Looking at the code of this `prefetch` script, there is *another* way this can be used. If you add a `data-prefetch-with-link="true"` attribute to a link, instead of making an Ajax call, it will add a `<link rel="prefetch">` element to the `head` tag of the page.

```
assets/turbo/prefetch.js
// ... lines 1 - 133
function preload(link) {
    const url = link.getAttribute("href")
    const loc = new URL(url, location.protocol + "//" + location.host)
    const absoluteUrl = loc.toString()

    if (link.dataset.prefetchWithLink == "true") {
        const prefetcher = document.createElement('link')
        prefetcher.rel = 'prefetch'
        prefetcher.href = url
        document.head.appendChild(prefetcher)
        pendingPrefetches.delete(absoluteUrl)
    } else if (!Turbo.navigator.view.snapshotCache.has(loc)) {
        fetchPage(url, responseText => {
            const snapshot = Snapshot.fromHTMLString(responseText)
            Turbo.navigator.view.snapshotCache.put(loc, snapshot)
            pendingPrefetches.delete(absoluteUrl)
        })
    }
}
```

Hello

What does that do? Great question! To explain, let's back up a little. So far, this whole prefetch script has been pure Turbo magic: it makes an Ajax call and stores it into Turbo's snapshot cache. But actually, your browser has a "prefetch" feature built into it! And *that* is what this `data-prefetch-with-link` code is leveraging.

To see how it works, close the prefetch script and comment out its import in `app.js`. I want to see how true prefetching works without *any* Turbo magic... because prefetching can be used on any site - even if it doesn't use Turbo.

```
assets/app.js
// ... lines 1 - 10
// start the Stimulus application
import './bootstrap';

import './turbo/turbo-helper';
//import './turbo/prefetch';
```

Here's the deal: imagine that, when a user goes to a specific page on our site, we're *fairly* sure that you know what the next page - or pages - will be that the user will go to. In that case, we can *hint* to the user's browser that, if it has some extra time, it can *prefetch* that URL so that if the user *does* navigate to it, it will load instantly from cache.

Let's try this. Add an item to your cart and then head to the cart page. It might be obvious that, once a user visits this page, they often click the "Check out" link next. So let's add a *hint* that the browser should "prefetch" that page.

Adding a prefetch link

How? Open the template for this page: `templates/cart/cart.html.twig`. On top, override a block called `metas`. This is *not* a standard Symfony block. But earlier in the tutorial, in `base.html.twig`, we added this.

Inside the block, add `link` - but instead of `rel="stylesheet"`, use `rel="prefetch"`. Then set the `href` to the checkout URL: `{{ path() }}` then name of that route, which is `app_checkout`.

```
templates/cart/cart.html.twig
{% extends 'base.html.twig' %}

{% block metas %}
    {{ parent() }}
    <link rel="prefetch" href="{{ path('app_checkout') }}">
{% endblock %}

// ... lines 8 - 32
```

That's it! By the way, Symfony has a web-link component that can help with this and can even help your server *push* resources - including CSS and JS files - via a server push. However, when it comes to prefetching another *page*, I recommend avoiding it and adding the link manually... because pages that are prefetched via the web-link component won't have access to the session cookie.

Anyways, let's go see what happens. Refresh the page and, on the network tools, click to see *all* types of requests... and scroll to the top. The top request, of course, is for `/cart`. But now... scroll down... there it is! A request for `/checkout` that took 360 milliseconds! This happens thanks to the `prefetch` link we just added. And even though you don't see it here, your browser knows to fetch this with the *lowest* priority: requests for other things - like CSS and JS files - have a higher priority.

So what happens *now* when we go to the checkout page? Let's find out: click "Check out"... then scroll back up to the top of the requests. Cool. Turbo - which doesn't know or care that we're doing this `prefetch` stuff - made its Ajax call like normal. But when it did, our browser was smart enough to *instantly* pull that from the *prefetch* cache: no second request was *actually* made! Instead of waiting 360 milliseconds for the Ajax request to finish and *then* rendering, Turbo started rendering, effectively, immediately.

Best of Both Worlds?

So this method of manually adding a `link` tag isn't as fancy as the hover technique we saw earlier. But it also avoids making *two* requests whenever we click a link. On the negative side, when we go to the cart page, a request will be made for the checkout page *regardless* of whether or not the user even gets *close* to clicking the checkout link.

So... neither approach is perfect. Could we... combine the two ideas? Yep! And that's exactly what the `data-prefetch-with-link` attribute attempts to do: it waits until you hover, and *then* adds the `prefetch` link. There are other tiny libraries that do something similar - like "instant.page" and "quicklink" - which makes sense... since adding a prefetch `link` tag has *nothing* to do with Turbo.

But... the devil is in the details. Suppose that we use this `prefetch` script - or one of those other libraries - to dynamically inject a `<link rel="prefetch">` into our `head` element whenever we hover over a link. That will work great. But when we navigate to a new page with Turbo, that `<link>` tag will *not* be included on the next page.

Watch: if we click to the cart page, and look in the head... actually, let me refresh to avoid any surprises. Here's the `<link rel="prefetch"`. But now click to another page... then look in

the `<head>`. Uh, duh, I'm *still* on the cart page - click to the homepage. *Now* the `prefetch` link is gone! This is just how Turbo works: when we navigate, the JavaScript and CSS tags inside the `head` element *do* persist across pages. But everything else is removed and replaced with whatever is on the *new* page only.

This has a big impact on `prefetch`. Our browser *did* `prefetch` the checkout page a minute ago when we were on `/cart`. But because the `link` tag is gone, our browser basically "forgets" that it did that. In a perfect world, as we navigate with Turbo, any `prefetch` links that were dynamically added would *remain* in your `head` element. That's probably possible by keeping track of all the links that you've prefetched and leveraging a Turbo event listener, but I haven't experimented with it yet. If you *do* play around with this and get some nice results, I would love to hear about it.

Here's the takeaway: even though these prefetch options are really cool and they can make your site mega fast, none of these are perfect yet. So use them wisely. In the real-world, I would probably use a link-by-link "opt-in" approach with the hover logic that leverages native `prefetch` links.

Okay: we are *done* with Turbo Drive! So let's turn to Turbo Frames: a feature that allows us to separate our site into little pieces that can navigate independently.

Chapter 20: Turbo Frames: Lazy Frames

Time to move on to part two of Turbo: Turbo frames. This is a brand new feature - it did *not* exist in the old Turbolinks library. To put it simply, Turbo frames allow you to treat part of your page, well, basically like an `iframe`! If you've never worked with iframes or IE6, I'm jealous. Turbo frames are a native, non-weird way to get the goodness of iframes... without actually using iframes, which are a pain in the butt.

Imagine that this category sidebar were inside a Turbo frame. If it were, you could click these links or even submit forms and only the *frame's* content would change: the rest of the page would be unaffected.

Frames are super cool, but I *do* want us to keep something in mind: they're an "extra" feature. Turbo Drive gives us the single page app experience. Frames give us the ability to make the user experience even *better*. But using frames *does* mean that you'll need to write some extra code. Frames are form of progressive enhancement... which basically means that you should get your site working first, *then* come back to see where a tool like Turbo frames can *enhance* it further.

The 2 Use Cases for Frames

Ok, so there are basically 2 use-cases for Turbo frames. The first is what we just talked about: you want navigation in just *one* area of your page to happen *inside* that area without affecting the rest of your page.

The second use-case is when you want a part of your page to load lazily. Literally, an area of your site would be empty on page load... then that Turbo frame would make an Ajax call to populate itself.

Upgrading to the Latest Turbo

Before we jump into an example, I'm going to find my terminal and run:

```
yarn upgrade @hotwired/turbo
```

As a reminder `@hotwired/turbo`, is a normal library and you can find it in the `package.json` file.

```
package.json
// ... line 1
    "devDependencies": {
        "@babel/preset-react": "^7.0.0",
        "@fortawesome/fontawesome-free": "^5.15.3",
        "@hotwired/turbo": "^7.0.0-beta.5",
        "@popperjs/core": "^2.9.1",
        "@symfony/stimulus-bridge": "^2.0.0",
// ... lines 8 - 34
```

This line *was* added automatically when we installed the `symfony/ux-turbo` PHP package, but we have *complete* control over managing its version. When I originally downloaded it, I got version `beta.5`. The latest version at the time of recording, which you can see over here, is `beta.7`. Not a lot has changed between the two versions, but there *was* one tweak to how JavaScript works in frames that I want to get.

Setting up a Lazy Frame

Okay, at your browser, head to the cart page. We're going to talk about the *second* use-case for Turbo frames first: lazy frames. See this featured product sidebar? Let's pretend that rendering this is kind of a heavy. If we could load it lazily - so via an Ajax call - then the rest of the cart page could load *faster* because it wouldn't need to do the work of preparing and rendering that section.

To lazily load this, we first need a route and controller that renders the sidebar. Open the template for this page: `templates/cart/cart.html.twig`. Let's see... this is where we render the featured sidebar. And you can see that it's *already* isolated into its own template. So all *we* need to do is create a route & controller that *render* this template.

```
templates/cart/cart.html.twig
// ... lines 1 - 7
{% block body %}
    <div class="container-fluid container-xl mt-4">
        <div class="row">
            <aside class="col-12 col-md-4 order-2 order-md-1">
                {% if featuredProduct %}
                    {{ include('cart/_featuredSidebar.html.twig') }}
                {% endif %}
            </aside>
// ... lines 16 - 32
```

Let's do that in `src/Controller/CartController.php`. This top method is the cart page itself. Copy that, paste below, rename it to `_cartFeaturedProduct()` and change the URL to `/cart/_featured`. I like to use that `_` prefix when something only renders *part* of a page. Below, instead of rendering `cart.html.twig`, render `_featuredSidebar.html.twig`. And... we don't need to pass the `cart` variable... and so we don't need this `CartStorage`. Oh, and the route needs a unique name, like `_app_cart_product_featured`.

```
src/Controller/CartController.php
// ... lines 1 - 34

    /**
     * @Route("/cart/_featured", name="_app_cart_product_featured")
     */
    public function _cartFeaturedProduct(ProductRepository $productRepository): Response
    {
        $featuredProduct = $productRepository->findFeatured();
        $addToCartForm = $this->createForm(AddItemToCartFormType::class, null, [
            'product' => $featuredProduct,
        ]);

        return $this->renderForm('cart/_featuredSidebar.html.twig', [
            'featuredProduct' => $featuredProduct,
            'addToCartForm' => $addToCartForm,
        ]);
    }

// ... lines 52 - 121
```

Cool. Now, up in the cart action, this will load faster because we can do *less* work... because we don't need to prepare the `addToCartForm` or fetch the `featuredProduct` anymore. We can even remove this argument.

The Custom <turbo-frame> Element

We can do all of this because, in the template for this action - `cart.html.twig` - we're not going to include this sidebar anymore. Instead, we're going to add a Turbo Frame... which is... just a custom HTML element - `<turbo-frame>` - which always has at *least* an `id` attribute that identifies it, like `id="cart-sidebar"`.

```
templates/cart/cart.html.twig
// ... lines 1 - 10
            <aside class="col-12 col-md-4 order-2 order-md-1">
                <turbo-frame id="cart-sidebar" src="{{
path('_app_cart_product_featured') }}">
                    Loading...
                </turbo-frame>
            </aside>
// ... lines 16 - 32
```

PhpStorm highlights this as an unknown tag, but the Turbo library *does* register it as a custom element.

If we stopped here, this would render an empty `<turbo-frame>` element on the page... and would do nothing. To make this a "lazy" frame, add a `src` attribute set to the URL that it should request to get its contents. In this case, that's `{{ path() }}` then `_app_cart_product_featured`. Inside the `turbo-frame`, we can put some loading text: this will show on page load while the Ajax call is being made.

That's it! With any luck, Turbo will see the frame, initiate the Ajax call and pop the response inside. Let's try it! Refresh and watch closely. Woh: the "Loading..." was there for *just* a second, then it disappeared! Check the console. Error!

> *“Response has no matching `<turbo-frame id="cart-sidebar">` element.”*

Interesting: it made the Ajax call and then looked for a `turbo-frame` element in the response with the same id as our frame. Why? The answer goes to the *core* of how Turbo frames work. Let's dive into that next and get this thing working.

Chapter 21: Turbo Frames Look for & Load the Matching Frame

On page load, Turbo *did* notice our new `<turbo-frame>` element and it *did* make an Ajax request to fetch the contents. But then, for some reason, it gave us this error. Why?

This is a *super* important detail of Turbo frames. When a frame makes an Ajax call, it looks in the response for a `<turbo-frame>` element that has the same *id* as itself and uses *its* content only. If it does *not* find a matching `<turbo-frame>`, in the response, then you get this error.

Ok, but... why? If you look in the network tools, the response from the Ajax call contains the *exact* HTML we want. Why doesn't it just take the *entire* HTML from the response and put it into the frame?

Well, we're not leveraging it in *this* example, but one of the super powers of the frame system is that you can point a frame at a URL that returns an *entire*, full HTML page. So if you pretend that this returns a full HTML page, the frame system is smart enough to only find and use the matching frame. This allows you to create full, normal pages and then *reuse* those full normal pages to power your frames.. avoiding the need to create extra endpoints for your frames like we did. If this doesn't make sense yet, don't worry. Our next example will illustrate this.

Adding the turbo-frame in the Response

Anyways, what we need to do is make sure that the response contains a `<turbo-frame>` element with `id="cart-sidebar"`. I'll copy that from `cart.html.twig`, open `_featuredSidebar.html.twig`, add that... and indent everything.

templates/cart/_featuredSidebar.html.twig

```
<turbo-frame id="cart-sidebar">
    <div class="component-light product-show p-3 mb-5">
        <h5 class="text-center">Featured Product!</h5>
        <a href="{{ path('app_product', { id: featuredProduct.id }) }}">
            <img
                alt="{{ featuredProduct.name }}"
                src="{{
asset('/uploads/products/'~featuredProduct.imageFilename) }}"
                class="d-block"
            >
        </a>
        <div class="pt-3">
            <h6 class="d-flex justify-content-between mb-3">
                <strong>{{ featuredProduct.name }}</strong>

                {{ featuredProduct.priceString|format_currency('USD') }}
            </h6>

            {{ include('product/_cart_add_controls.html.twig') }}
        </div>
    </div>
</turbo-frame>
```

Notice that we don't have a `src=""` on *this* frame: this is *not* a lazy frame... it's just a normal frame that already has its final content.

Ok: let's try it again. Refresh and... yes! It works! It looked in the response for the `<turbo-frame>` with the id, found it and used its HTML. If you inspect element and find the `turbo-frame`, you can see the `src=""` attribute is still there, but now it's filled with content.

At this point, if you click any links or submit the form on the sidebar... it might not work like you expect because the frame will keep any navigation *inside* the frame. That's the first use-case for Turbo Frames - and we'll come back in a few minutes to address this.

Using fragment_uri()

Oh, and by the way, if you're using Symfony 5.3 and you create a controller - like this one - that just renders part of a page, you don't *have* to give this a route. There's another option. Remove this route.

```
src/Controller/CartController.php
// ... lines 1 - 29

    public function _cartFeaturedProduct(ProductRepository
$productRepository): Response
    {
        $featuredProduct = $productRepository->findFeatured();
        $addToCartForm = $this->createForm(AddItemToCartFormType::class,
null, [
            'product' => $featuredProduct,
        ]);

        return $this->renderForm('cart/_featuredSidebar.html.twig', [
            'featuredProduct' => $featuredProduct,
            'addToCartForm' => $addToCartForm,
        ]);
    }
// ... lines 43 - 113
```

Now, in `cart.html.twig`, instead of `{{ path() }}`, use `{{ fragment_uri() }}` and then `controller()` and *then* the name of the controller: `App\\Controller\\CartController::` and then the method name... which is `_featuredProduct`.

```
templates/cart/cart.html.twig
// ... lines 1 - 10
            <aside class="col-12 col-md-4 order-2 order-md-1">
                <turbo-frame id="cart-sidebar" src="{{
fragment_uri(controller('App\\Controller\\CartController::_cartFeaturedProdu
}}">
                    Loading...
                </turbo-frame>
            </aside>
// ... lines 16 - 32
```

This is a bit longer - and those double slashes are ugly and needed because backslash is an escape character. Behind the scenes, this will generate a signed URL - called a fragment URL - that renders our controller. To get this to work, make sure that you have the fragment system activated: that's in `config/packages/framework.yaml`. Uncomment `fragments: true`.

```
config/packages/framework.yaml
// ... lines 1 - 14
    #esi: true
    fragments: true
    php_errors:
        log: true
// ... lines 19 - 25
```

Let's try this. Move over, refresh the page and cool! It still works! If you look at the `turbo-frame`, the `src=""` is now set to a long, weird looking `_fragments` URL.

Next: let's look at a second lazy frame example. But this time, instead of creating a controller that renders *just* the frame, we're going to populate a frame by *reusing* an existing, full HTML page.

Chapter 22: Using a Full HTML Page to Populate a Frame

I want to show one more lazy frame example. But before we do, I'm going to find my terminal and, yes, once again, run:

```
yarn upgrade @hotwired/turbo
```

This time I get beta version 8, which is *actually* the release I was waiting for. This changes how JavaScript is handled inside frames, which will be important for what we're about to do.

But for a minute, I want you to completely forget about frames. Let's pretend that we, being the nerds that we are, want to add a weather page to our site! Sure, we have this weather footer on the bottom of every page, but we *also* want people to be able to go to `/weather` and see the weather report front and center.

Creating a Normal Weather Page

Over in `src/Controller/`, create a new class called `WeatherController`. Make it extend `AbstractController` and add a public function `weather()` with a route above it: `@Route('/weather')`, `name="app_weather"`. Inside, return `$this->render('weather/index.html.twig')`.

src/Controller/WeatherController.php

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class WeatherController extends AbstractController
{
    /**
     * @Route("/weather", name="app_weather")
     */
    public function weather()
    {
        return $this->render('weather/index.html.twig');
    }
}
```

Cool! Let's go make that template! Down in `templates/`, create a new directory called `weather/`, and, inside, the new file: `index.html.twig`. Give this the basic structure `{% extends 'base.html.twig' %}`, `{% block body %}`, `{% endblock %}` and an `<h1>`.

Now go into `base.html.twig` and... at the bottom, steal all of the weather stuff: the anchor tag and the script element. In `index.html.twig`, paste.

```
templates/weather/index.html.twig
{% extends 'base.html.twig' %}

{% block body %}
    <h1>The Weather!</h1>

    <a class="weatherwidget-io"
href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW
YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
    <script>
    !function (d, s, id) {
        var js, fjs = d.getElementsByTagName(s)[0];
        if (!d.getElementById(id)) {
            js = d.createElement(s);
            js.id = id;
            js.src = 'https://weatherwidget.io/js/widget.min.js';
            fjs.parentNode.insertBefore(js, fjs);
        }
    }(document, 'script', 'weatherwidget-io-js');
    </script>
{% endblock %}
```

Done! Oh, but in `base.html.twig`, let's add a link to this... find the cart link - there it is - copy it, paste, change the route to `app_weather` and... for the text, I'll use a FontAwesome icon: `fas fa-sun`.

```
templates/base.html.twig
// ... lines 1 - 29
                    <ul class="navbar-nav">
                        <li class="nav-item">
                            <a class="nav-link" href="{{
path('app_weather') }}">
                                <span class="fas fa-sun"></span>
                            </a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('app_cart')
}}">
                                Shopping Cart ({{ count_cart_items() }})
                            </a>
                        </li>
// ... lines 41 - 110
```

Let's go check it out! Move over, refresh and... there's our sunshine! When we click the icon, we have a weather page. Amazing!

Though... having *two* weather widgets on the page *does* look weird. Let's remove the one in the footer for *just* this page. In `base.html.twig`, scroll back down to that area. Surround this in a new `{% block weather_widget %}` and, on the other side, `{% endblock %}`.

templates/base.html.twig

```
// ... lines 1 - 88
        {% block weather_widget %}
        <a class="weatherwidget-io" href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
        <script>
        !function (d, s, id) {
            var js, fjs = d.getElementsByTagName(s)[0];
            if (!d.getElementById(id)) {
                js = d.createElement(s);
                js.id = id;
                js.src = 'https://weatherwidget.io/js/widget.min.js';
                fjs.parentNode.insertBefore(js, fjs);
            }
        }(document, 'script', 'weatherwidget-io-js');
        </script>
        {% endblock %}
// ... lines 103 - 112
```

Back in `index.html.twig`, anywhere, override that block... but make it empty.

templates/weather/index.html.twig

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

{% block weather_widget %}{% endblock %}
```

Ok, refresh again and... cool!

At this point, we *do* have some code duplication between `index.html.twig`, and `base.html.twig`. We could easily fix that by isolating the weather widget code into its own template... and then using the Twig `{{ include() }}` function in both templates to bring that in.

Creating the Lazy Turbo Frame

But like we did with the featured product sidebar, I want you to pretend that it takes a lot of work to generate this HTML... maybe we make some database calls or API calls to generate it. And so, if we could convert the weather widget that's on the footer of every page into a lazy turbo *frame*, well, that would make *every* page load faster!

When we created a lazy turbo frame for the featured product sidebar, we started by making a route and a controller that rendered just that *part* of the page: just the featured product itself - *without* the layout. But this time, we're *not* going to do that.

Why not? Because we already have a page that contains the HTML we need! The weather page! Sure, it contains a lot of *extra* stuff that we *don't* want... like the HTML layout and the `<h1>` tag... but the turbo-frame system can ignore all that. Yup, we can jump *straight* to adding the turbo frame with zero extra work.

In `base.html.twig`, remove all the duplicated code and instead say, `<turbo-frame id="">`, how about, `weather_widget`. Then, because we want this to be a *lazy* frame, add `src=""` and point this at the *full* HTML page that we want to target: the weather page.

templates/base.html.twig

```
// ... lines 1 - 87

        {% block weather_widget %}
            <turbo-frame id="weather_widget" src="{{ path('app_weather') }}"></turbo-frame>
        {% endblock %}

// ... lines 93 - 101
```

If we try this... I'll go to the homepage... it's not going to work. In the console, we see a familiar error!

> *"Response has no matching `<turbo-frame id="weather_widget">` element."*

Of course! We need to tell the Turbo frame system *which* part of the weather page to use for this frame. Over in `index.html.twig` - the template for the full weather page - wrap the entire weather section in a `<turbo-frame>` that has `id="weather_widget"`. I'll put the closing tag down here... and indent.

```
templates/weather/index.html.twig
// ... lines 1 - 2
{% block body %}
    <h1>The Weather!</h1>

    <turbo-frame id="weather_widget">
        <a class="weatherwidget-io"
href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW
YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
        <script>
        !function (d, s, id) {
            var js, fjs = d.getElementsByTagName(s)[0];
            if (!d.getElementById(id)) {
                js = d.createElement(s);
                js.id = id;
                js.src = 'https://weatherwidget.io/js/widget.min.js';
                fjs.parentNode.insertBefore(js, fjs);
            }
        }(document, 'script', 'weatherwidget-io-js');
        </script>
    </turbo-frame>
{% endblock %}
// ... lines 21 - 23
```

Testing time! Refresh again and... it works! That's amazing! We're now able to reuse just *parts* of existing pages simply by wrapping those parts inside a `<turbo-frame>`. If you look at the network tools... and find the Ajax call for the weather page, there's no magic here: the Ajax call for the frame *did* return the full HTML.

And this is really how frames are meant to be used. You have an existing page like the weather page, and then you're able to reuse parts of that page inside a frame instead of needing to build an extra endpoint that returns only the *part* you want.

Truly Lazy Frames: Load only when Visible

Ok, ready to be *more* amazed? Check out the homepage: this is a *long* page. Don't you think it's kind of a wasteful to load the weather widget in the footer... even if the user never scrolls down that far? It is wasteful! And we can fix that!

In `base.html.twig`, on the `turbo-frame`, add a new attribute: `loading="lazy"`,

```
templates/base.html.twig
// ... lines 1 - 87

        {% block weather_widget %}
            <turbo-frame id="weather_widget" src="{{ path('app_weather') }}" loading="lazy"></turbo-frame>
        {% endblock %}

// ... lines 93 - 101
```

Let's see what that did. Scroll to the top of the homepage, refresh and make sure you're looking at the Ajax calls in the network tools. Notice that Turbo has *not*, yet, made an Ajax request for the weather page. But keep an eye on this. If we scroll down... there it is! Yup, when you add `loading="lazy"`, the request isn't made until the frame becomes *visible*. That's *super* cool.

But... there's a lingering bug in our code. It's more about the *JavaScript* for the weather widget than about the turbo-frame we created. Let's find out what the bug is next and create a Stimulus controller that will make the weather JavaScript finally, fully functional, no matter how we load it.

Chapter 23: Reliably Load External JS with Stimulus

Thanks to the turbo-frame system, we're now lazily-loading just *part* of the weather page down in the footer. And... notice that this *is* working... which actually proves something: `script` tags inside frames *are* executed.

script Tags in Frames Are Executed

Let me find that frame... here it is. Ok, so no surprise: if you have a `<script>` tag that's included in a `turbo-frame`, Turbo *does* execute that... *exactly* like how Turbo *Drive* executes any `script` tags found inside the *body* element.

That's great! But... we have a bug that's hiding. Well, sort of *two* bugs. Yikes! To see the first, scroll to the top of the page, refresh but don't scroll down. Now click to the weather page... and check out the console. Error!

> *“Uncaught reference error: `__weatherwidget_init()` is not a function”*

And it's coming from `turbo-helper`. Go open that file - `turbo/turbo-helper.js` and scroll down to line 71. Here we are: `initializeWeatherWidget()`.

```
assets/turbo/turbo-helper.js
// ... line 1

const TurboHelper = class {
    constructor() {
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
        });

        document.addEventListener('turbo:render', () => {
            this.initializeWeatherWidget();
        });

        this.initializeTransitions();
    }

// ... lines 17 - 39

    initializeWeatherWidget() {
        __weatherwidget_init();
    }

// ... lines 45 - 95
```

If you scroll back up, this `initializeWeatherWidget()` function is called when the `turbo:render` event is dispatched. Its job is to *reinitialize* the weather widget on the next page. The problem is that, in this case, the weather widget JavaScript hasn't *quite* yet been loaded onto the page... because it didn't load at all on the first page. And the *real* problem is that... well... I didn't code defensively.

Fix this by adding an if: if `typeof __weatherwidget_init === 'function'`, *then* call this. Otherwise, it means the JavaScript hasn't been loaded... so no reason to do anything.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 39

    initializeWeatherWidget() {
        if (typeof __weatherwidget_init === 'function') {
            __weatherwidget_init();
        }
    }

// ... lines 47 - 97
```

The Weather Widget JavaScript is not Always Reinitialized

So... this would fix *one* problem... but not our *bigger* problem. To see that one, over on the product page, below the sidebar, I want to add a *second* weather widget. Open the template for this page: `templates/product/index.html.twig`. Oh, but actually, the sidebar is in `productBase.html.twig`.

Cool: right here, I'm going to add `<turbo-frame>` with `id="weather_widget"` - to match the id that we've been using so far - and `src="{{ path('app_weather') }}"`.

Try it! Refresh and... bah! It works - but I put it in the wrong spot! I meant to put it in the `<aside>`. Let's try that again. Refresh now and... beautiful.

templates/product/productBase.html.twig

```
// ... lines 1 - 5
        <div class="row">
            <aside class="col-12 col-md-3 order-2 order-md-1">
                {{ include('product/_categoriesSidebar.html.twig') }}

                <turbo-frame id="weather_widget" src="{{
path('app_weather') }}"></turbo-frame>
            </aside>

            <div class="col-12 col-md-9 product-show order-1 order-md-2">
                {% block productBody %}{% endblock %}
            </div>
        </div>
// ... lines 17 - 19
```

Now scroll to the footer. It's busted! Hmm... the turbo frame did its job - the HTML is here - but the JavaScript didn't initialize! What happened?

Let's remember how this is *supposed* to work... because it's getting kind of complicated. On page load, or really, anytime that the weather JavaScript is first executed, it adds a `<script>` tag to the page, which downloads an external JavaScript file. *That* JavaScript finds any elements on the page with a `weatherwidget-io` class and initializes the weather widget inside of them.

But... when we surf to another page, this external JavaScript file is *not* re-executed... because this function is smart enough to not add the same script tag multiple times. We hit this problem earlier. To fix it, back in `turbo-helper.js`, we added this `__weatherwidget_init()`

code, which is executed on `turbo:render`. So basically, each time Turbo renders the page, we call `__weatherwidget_init()` and *that* reinitializes the weather widget for that page.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 39

    initializeWeatherWidget() {
        if (typeof __weatherwidget_init === 'function') {
            __weatherwidget_init();
        }
    }

// ... lines 47 - 97
```

This worked *great* when the *only* way that a weather widget tag could be added to a page was as a result of a Turbo Drive navigation. But now, this tag is sometimes loaded onto the page via Ajax by a Turbo Frame... and that does *not* trigger the `turbo:render` event... because we're not rendering a full page. In other words, when a Turbo frame loads, nothing is calling the `__weatherwidget_init()` function!

If you're watching *really* closely, you might be wondering how the weather widget in this lazy frame was *ever* working... since we were *never* calling the `__weatherwidget_init()` function after it loaded. It worked simply thanks to some smart code that lives inside that function. If you looked at the external JavaScript in detail - which we did a bit earlier - you would see that when you call the `__weatherwidget_init()` function, if it does *not* find any `weatherwidget-io` elements on the page, it automatically recalls itself every 1.5 seconds *until* it finds one. This... almost accidentally... made sure that once our lazy frame in the footer loaded, the JavaScript was initialized within 1.5 seconds. But... it wasn't a very robust solution, and it stopped working as soon as there was a *second* widget on the page that loaded earlier.

So let's fix *all* of this and simplify our code a *bunch*... because it took *way* too long to explain how this has been *barely* working.

How can we improve this? By creating a Stimulus controller! I know, this tutorial is about Turbo... but since Turbo *really* works best when you have *no* inline script tags, let's see how Stimulus could help us manage this external JavaScript.

Creating the Stimulus Controller

Here's the idea: let's attach a Stimulus controller to the `weatherwidget-io` anchor tag. By doing that, *whenever* this element appears on the page... no matter *how* or *when* it appears, we can run some code... like `__weatherwidget_init()`.

In `assets/controllers/`, create a new file called, how about, `weather-widget_controller.js`. I'm going to cheat... as usual... and steal the code from another controller, paste... then clear everything out. Start with a `connect()` function and `console.log('🌦️')`.

assets/controllers/weather-widget_controller.js

```javascript
import { Controller } from 'stimulus';

export default class extends Controller {
    connect() {
        console.log('?');
    }
}
```

Next, over in `weather/index.html.twig`, find the anchor tag and add `data-controller=""` and the name of our new controller: `weather-widget`.

templates/weather/index.html.twig

```twig
// ... lines 1 - 5
    <turbo-frame id="weather_widget">
        <a data-controller="weather-widget" class="weatherwidget-io" href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
        <script>
        !function (d, s, id) {
            var js, fjs = d.getElementsByTagName(s)[0];
            if (!d.getElementById(id)) {
                js = d.createElement(s);
                js.id = id;
                js.src = 'https://weatherwidget.io/js/widget.min.js';
                fjs.parentNode.insertBefore(js, fjs);
            }
        }(document, 'script', 'weatherwidget-io-js');
        </script>
    </turbo-frame>
// ... lines 20 - 23
```

Okay! Let's make sure that's connected. Head over, scroll up... refresh the homepage and check the console. Perfect! This log is coming from the weather widget on the sidebar. Now watch what happens when we scroll down... a second emoji!

The next step is to move all of this JavaScript into our Stimulus controller. Copy everything and delete the `<script>` tag entirely.

```
templates/weather/index.html.twig
// ... lines 1 - 5
    <turbo-frame id="weather_widget">
        <a data-controller="weather-widget" class="weatherwidget-io" href="https://forecast7.com/en/40d71n74d01/new-york/" data-label_1="NEW YORK" data-label_2="WEATHER" data-theme="original" >NEW YORK WEATHER</a>
    </turbo-frame>
// ... lines 9 - 12
```

In the controller, after `connect()`, paste! That is *totally* invalid JavaScript... and my build system *and* editor are freaking out.

```
assets/controllers/weather-widget_controller.js
import { Controller } from 'stimulus';

export default class extends Controller {
    connect() {
        console.log('?');
    }

    !function (d, s, id) {
        var js, fjs = d.getElementsByTagName(s)[0];
        if (!d.getElementById(id)) {
            js = d.createElement(s);
            js.id = id;
            js.src = 'https://weatherwidget.io/js/widget.min.js';
            fjs.parentNode.insertBefore(js, fjs);
        }
    }(document, 'script', 'weatherwidget-io-js');
}
```

Let's turn this into a function called `initializeScriptTag()`. Copy these three arguments and remove them. Cool.

Up in `connect()`, instead of logging a cloud, say `this.initializeScriptTag()` and *pass* those three arguments.

```
assets/controllers/weather-widget_controller.js
// ... lines 1 - 2
export default class extends Controller {
    connect() {
        this.initializeScriptTag(document, 'script', 'weatherwidget-io-js');
    }

    initializeScriptTag (d, s, id) {
        var js, fjs = d.getElementsByTagName(s)[0];
        if (!d.getElementById(id)) {
            js = d.createElement(s);
            js.id = id;
            js.src = 'https://weatherwidget.io/js/widget.min.js';
            fjs.parentNode.insertBefore(js, fjs);
        }
    }
}
```

So... this isn't perfect yet... but it's closer: each time Stimulus sees a matching anchor tag, it's going to run this.

Let's try it. Scroll back up to the top, refresh and... awesome! The fact that this loads means that our Stimulus controller *did* just execute and add the script tag. If you look in the `head` of our page... there it is!

But... if we scroll to the bottom of the page... that still *doesn't* work. It's ok, we expected that: we still need to move the `__weatherwidget_init()` code into Stimulus.

Copy the entire if statement, delete the `initializeWeatherWidget()` function, scroll up and remove the event listener entirely. Over in the `weather-widget` controller, up in `connect()`, paste that and then move the `initializeScriptTag()` call, which I *totally* misspelled... let me fix that - move that into the `else`.

```
assets/controllers/weather-widget_controller.js
// ... lines 1 - 2
export default class extends Controller {
    connect() {
        if (typeof __weatherwidget_init === 'function') {
            __weatherwidget_init();
        } else {
            this.initializeScriptTag(document, 'script', 'weatherwidget-io-js');
        }
    }
// ... lines 11 - 22
```

So *if* the `__weatherwidget_init()` function already exists, just call it! Else, run the code to add the original `script` tag to the page.

I think we're ready! Scroll back up to the top of page and refresh. The sidebar works... the footer works... and, if we go to the weather page, that works too!

I *love* this approach. Even though our external JavaScript is *not* written in Stimulus, we can still *use* Stimulus to activate this JavaScript *exactly* when we want to. At this point, we can add this anchor tag *anywhere* on our site, and it *will* instantly do the work to initialize itself.

Next: let's investigate the second-use case for Turbo Frames... and really the *main* use case: the ability to keep navigation isolated to one section of the page.

Chapter 24: Targeting Links in or out of the Frame

Head to the cart page and click the feature product to go to its page. Whoa. It disappeared! And... we're still on the cart page. Head to the console. Ah, that's a familiar error!

Response has no matching `<turbo-frame id="cart-sidebar">`. This shows off the true main property of a `<turbo-frame>`: any navigation inside of a frame - whether you click a link or fill out a form - will *stay* inside that frame.

Refresh. When we click this link, it *does* make an Ajax request to the "inflatable sofa" product page: you can see it down here in the network tools. It *then* looked for a `cart-sidebar` turbo frame on that page because it wants to find which *part* of this page it should render inside of the `cart-sidebar` frame.

But... in this case, that is *not* what we wanted! We wanted to leverage the nice, lazy-loading coolness of the turbo frame... but after that... we kind of want all its links and forms to navigate like normal.

target="_top" For "Normal" Navigation

No problem. Open the template for the cart page: `templates/cart/cart.html.twig`. On the `<turbo-frame>`, add `target="_top"`.

```
templates/cart/cart.html.twig
// ... lines 1 - 10
            <aside class="col-12 col-md-4 order-2 order-md-1">
                <turbo-frame id="cart-sidebar" src="{{ fragment_uri(controller('App\\Controller\\CartController::_cartFeaturedProdu
}}" target="_top">
                    Loading...
                </turbo-frame>
            </aside>

// ... lines 17 - 32
```

That's it! The `_top` means that any links or forms inside of this frame should target the *main* page. You can also change the target on just a *specific* link or form instead of the *entire* frame...

and we'll see how later.

Anyways, if we refresh now... and click. It's back to normal. If you go back to the shopping cart and click to add the item to your cart, this *also* works. That just submitted a form... which was *also* broken a minute ago before we added `target="_top"`.

Adding Attributes on the Initial Frame or Ajax-Loaded Frame?

But... wait a second. We just added `target="_top"` to the turbo frame in `cart.html.twig`. But what about the `turbo-frame` over here in `_featuredSidebar.html.twig`? This is the frame that's actually loaded via Ajax.

Let's talk about a small - but important - detail about turbo frames. When we initially load the cart page, all its HTML comes from `cart.html.twig`. This means that what we're originally loading on the page is a `turbo-frame` with a `src` attribute and a `target` attribute.

But what happens after it makes the Ajax request? Does the `turbo-frame` from the Ajax request *replace* the existing one that loaded on the page originally? Or... does it keep the original `turbo-frame` tag and only use the new frames *inner* HTML?

The answer is that a turbo frame only uses the *inner* HTML. So whatever attributes your frame *starts* with - like `src` and `target` - it will keep those, regardless of the attributes on any `turbo-frame` that it loads later via Ajax. Well, the the `src` attribute changes to the new URL, but that's it.

We can see this over in our browser. Inspect this frame: this `turbo-frame` has `src` and `target="_top"`. So, when the new frame loaded via Ajax, that frame didn't *replace* this one: we know that because only the *original* frame has `target="_top"`.

Anyways, this is why we added `target="_top"` to the frame in `cart.html.twig`: our *original* frame.

But... in `_featuredSidebar.html.twig`, I'm *also* going to add `target="_top"` here.

```
templates/cart/_featuredSidebar.html.twig
```

```
<turbo-frame id="cart-sidebar" target="_top">
    <div class="component-light product-show p-3 mb-5">
        <h5 class="text-center">Featured Product!</h5>
        <a href="{{ path('app_product', { id: featuredProduct.id }) }}">
            <img
                alt="{{ featuredProduct.name }}"
                src="{{
asset('/uploads/products/'~featuredProduct.imageFilename) }}"
                class="d-block"
            >
        </a>
        <div class="pt-3">
            <h6 class="d-flex justify-content-between mb-3">
                <strong>{{ featuredProduct.name }}</strong>

                {{ featuredProduct.priceString|format_currency('USD') }}
            </h6>

            {{ include('product/_cart_add_controls.html.twig') }}
        </div>
    </div>
</turbo-frame>
```

Why? Well, functionally-speaking, it makes no difference. But conceptually, if you look at this frame in isolation, its links - like this link and the form down here - are not designed to navigate in the frame. Both are really meant to target the main page. Adding `target="_top"` here makes that clear.

And also, if we ever simply use Twig's `include()` function to include this template directly on a page, the frame would already have the `target="_top"` that it needs. Though, an even *better* way to guarantee that a link has the right target is to add it *to* the link itself - which we'll see soon.

So now that we've made this turbo frame *not* to keep its navigation inside of itself, let's see a real example of when keeping the normal turbo-frame behavior is awesome.

Chapter 25: Adding a "Read More" Ajax Frame

On the cart page, let's make this feature product sidebar a bit more useful by adding the product's description. Except... we probably don't want to add the whole description because it's kind of long. So we'll just show a preview.

Using & Installing twig/string-extra

Head over to `templates/cart/_featuredSidebar.html.twig` and, down here, right before the cart controls, add `{{ featuredProduct.description }}`. To show only *part* of the description, pipe this to a special `|u` filter and say `.truncate(25)`. I'm also going to add `|trim` on the end.

templates/cart/_featuredSidebar.html.twig

```
// ... lines 1 - 10
        <div class="pt-3">
            <h6 class="d-flex justify-content-between mb-3">
                <strong>{{ featuredProduct.name }}</strong>

                {{ featuredProduct.priceString|format_currency('USD') }}
            </h6>

            {{ featuredProduct.description|u.truncate(25)|trim }}...

            {{ include('product/_cart_add_controls.html.twig') }}
        </div>
// ... lines 22 - 24
```

This `u` filter comes from a Twig extension library, which... we don't actually have installed yet. But, pff, let's try it anyways. When we refresh.... nothing happens! But down on the web debug toolbar, you can see that an Ajax call failed!

So... we all know that - despite our awesomeness - errors happen. When you work with turbo frames, these errors are harder to *see* since everything happens in a background Ajax call. To see it, you can go to the network tab, find the request, right click and hit "open in new tab". There it is. *Or* you can go use the web debug toolbar to open the profiler for that request... which opens straight to the exception.

Either way, once we can *see* the error, it's really clear! It says run `composer require twig/string-extra`. Ok! Copy that, find your terminal and paste:

```
composer require twig/string-extra
```

Once this finishes... move back over, close the profiler, refresh and nice! Wait, hmm... I meant to put a little `...` at the end of the shortened description. And... yea: that looks better.

Replacing our Featured Sidebar Route

But *now*, let's make this *way* cooler by adding a "read more" link after the description that, on click, will show the *entire* description. We're going to do that with *zero* JavaScript thanks to Turbo frames.

But before we implement that, head over to `src/Controller/CartController.php`. On `_cartFeaturedProduct()`, I'm going to *re-add* the route that we had earlier: `@Route("/cart/_featured", name="_app_cart_product_featured")`.

```
src/Controller/CartController.php
// ... lines 1 - 30
    /**
     * @Route("/cart/_featured", name="_app_cart_product_featured")
     */
    public function _cartFeaturedProduct(ProductRepository
$productRepository): Response
    {
// ... lines 36 - 116
```

Copy the route name then, over in the cart template - so `cart.html.twig` - instead of using the `fragment_uri()` function, go back to using `{{ path() }}` and then `_app_cart_product_featured`.

```
templates/cart/cart.html.twig
// ... lines 1 - 10
            <aside class="col-12 col-md-4 order-2 order-md-1">
                <turbo-frame id="cart-sidebar" src="{{
path('_app_cart_product_featured') }}" target="_top">
                    Loading...
                </turbo-frame>
            </aside>
// ... lines 16 - 32
```

Doing this is *totally* unnecessary to accomplish our new goal. The reason I'm doing this is because, in a few minutes, it'll make it easier to play with the frame's URL.

Setting up the Target Endpoint

Now let's get to work. Back over in `CartController`, here's the idea: if someone requests this URL - but with a `?description=1` on the *end* of that URL - then we'll render the *full* description. Otherwise, we'll render the truncated description like we are now.

To do that, add a `Request` argument - the one from HttpFoundation - and then pass a new variable into the template called `showDescription` set to `$request->query->get('description')`.

```
src/Controller/CartController.php
// ... lines 1 - 30
    /**
     * @Route("/cart/_featured", name="_app_cart_product_featured")
     */
    public function _cartFeaturedProduct(ProductRepository $productRepository, Request $request): Response
    {
        $featuredProduct = $productRepository->findFeatured();
        $addToCartForm = $this->createForm(AddItemToCartFormType::class, null, [
            'product' => $featuredProduct,
        ]);

        return $this->renderForm('cart/_featuredSidebar.html.twig', [
            'featuredProduct' => $featuredProduct,
            'addToCartForm' => $addToCartForm,
            'showDescription' => $request->query->get('description'),
        ]);
    }
// ... lines 47 - 117
```

Next, in `_featuredSidebar.html.twig`, if `showDescription`, then render the full description: `featuredProduct.description`. Else, render the preview.

Now here's the *big* question: how do we create that "read more" link? Remember that we're *inside* of a turbo-frame... and one of the properties of a turbo frame is that navigation stays *inside* that frame. So if we create a link to a page, or page partial, that renders this frame, Turbo will handle all the heavy lifting of making the Ajax request, finding the frame and putting its content *right* here.

In other words, all we need to do is create a boring, normal link to `{{ path() }}` `_app_cart_product_featured` *with* `description: true`.

Hmm... PhpStorm is confused, so I'll delete and re-add this quote to reset the highlighting. Inside the link, say "(read more)".

```
templates/cart/_featuredSidebar.html.twig
// ... lines 1 - 16

            {% if showDescription %}
                {{ featuredProduct.description }}
            {% else %}
                {{ featuredProduct.description|u.truncate(25)|trim }}...

                <a href="{{ path('_app_cart_product_featured', {
                    description: true,
                }) }}">(read more)</a>
            {% endif %}

// ... lines 28 - 32
```

Done... or done-ish. If we refresh the page... we have a link! But when we click it... the whole page navigates as if we were *not* in a turbo frame! Click back.

This happened because, in `cart.html.twig`, our turbo frame has `target="_top"`. That makes it behave, kind of *not* like a frame: all link clicks and form submits apply to the *whole* page. But we now *want* this one link - this read more link - to "yes" behave like a *normal* turbo frame: we want it to *keep* its navigation inside the frame.

```
templates/cart/cart.html.twig
// ... lines 1 - 11
                <turbo-frame id="cart-sidebar" src="{{ path('_app_cart_product_featured') }}" target="_top">
                    Loading...
                </turbo-frame>
// ... lines 15 - 32
```

To override the `target="_top"`, find the link in `_featuredSidebar`. Let's put this onto multiple lines. Add `data-turbo-frame=""` and then the name of our frame: `cart-sidebar`.

```
templates/cart/_featuredSidebar.html.twig
// ... lines 1 - 16

            {% if showDescription %}
                {{ featuredProduct.description }}
            {% else %}
                {{ featuredProduct.description|u.truncate(25)|trim }}...

                <a
                    data-turbo-frame="cart-sidebar"
                    href="{{ path('_app_cart_product_featured', {
                    description: true,
                }) }}">(read more)</a>
            {% endif %}

// ... lines 30 - 34
```

That's it! We also could have done the opposite... which in some ways would have been more natural. We could have left *off* the `target="_top"` - so that our entire frame behaves like normal - and then added `data-turbo-frame="_top"` to the link and form that *should* navigate the whole page.

Either way, the result would be the same. Refresh now... and click. Beautiful! Let's try that again. Oh, that's nice *and* simple: an Ajax system entirely powered by small changes in PHP and Twig only.

Manually Changing the src Attribute

Ooh, and now that this is working, I want to show you something cool. Inspect the turbo-frame. Notice that when you click a link, it *changes* the `src=` attribute to the new URL.

This is actually the *way* that turbo frames work. Each turbo frame *watches* its `src` attribute. When it changes, it *notices* that and makes an Ajax call *to* that new URL. In a normal situation, you click a link inside a frame, *that* changes the `src` attribute and *that* triggers the Ajax call.

But you can also change this by hand... it's kind of fun. Take out the `?description=1` and... cool! It made an Ajax request for the URL and rendered it! Our "read more" link is back! If we *click* that link, it makes another Ajax call and loads back.

That's a really neat, conceptual, thing to realize about turbo frames: they really *do* work like iframes.

Next: let's make this frame a little bit smoother by adding a loading animation between the time that we click the link and when the description actually renders.

Chapter 26: Frame Loading Animations

With Turbo Drive, when we click a link or submit a form, and that takes *longer* than 500 milliseconds to load, we get a loading animation on the top of the page... which we don't see here because this is all loading *fast*, but we saw it earlier. It's a built-in, global loading indicator that we don't even need to think about.

But the same thing does *not* happen for Turbo frames. When you click the read more link, that loads pretty fast, but there *is* a slight delay when nothing happens. And if clicking this loaded a heavier page.... it might *not* load so fast. It's pretty normal to add a loading indicator in situations like this. Can we add one with Turbo frames?

The "busy" Attribute

Sure! And we already have what we need. Head over to `src/Controller/CartController.php`. In `_cartFeaturedProduct()`, let's sleep for three seconds to fake a slow page.

Back at the browser, inspect this `turbo-frame` and make sure it's highlighted. Watch the element closely when I refresh. Look! It has a `busy` attribute! Yup, whenever a `turbo-frame` is loading, it gets this attribute. If we click the "read more" link, we'll see it again.

This simple attribute makes it possible to add all *sorts* of loading indicators. For example, we could create two classes to help us hide or *show* an element during loading.

Hiding / Showing Elements During Loading

Open up `templates/cart/_featuredSidebar.html.twig`. Ok, let's pretend that we want to hide the "read more" link once we click it. Add `class=""` and let's invent a new class called `frame-loading-hide`. We'll add the CSS for this in a minute. After this, add a `<span>` and give it a different, new, class - `frame-loading-show` - that will cause this element to only *show* when loading. Also give this `fas fa-spinner fa-spin` to render a FontAwesome loading animation.

```
templates/cart/_featuredSidebar.html.twig
// ... lines 1 - 17
                {% if showDescription %}
                    {{ featuredProduct.description }}
                {% else %}
                    {{ featuredProduct.description|u.truncate(25)|trim }}...

                    <a
                        data-turbo-frame="cart-sidebar"
                        class="frame-loading-hide"
                        href="{{ path('_app_cart_product_featured', {
                        description: true,
                    }) }}">(read more)</a>
                    <span class="frame-loading-show fas fa-spinner fa-spin">
</span>
                {% endif %}
// ... lines 31 - 36
```

To add styling for these, open up `assets/styles/app.css`. Target the `busy` attribute with `turbo-frame[busy]`. So *if* there's a turbo-frame element that has a `busy` attribute, then for any elements inside with a `frame-loading-hide` class, `display: none`.

For the *other* class - the `frame-loading-show` - we want this to *hide* by default and then only *show* when loading. First, to hide it, copy the CSS selector, paste, make it apply to *all* turbo-frame elements, and look for the `frame-loading-show` class. So, hide these by default.

And, whoops! That jumped a bit. Anyways, below this, *override* that: inside a `turbo-frame[busy]` element, if you have a `frame-loading-show` class, `display: inline-block`.

```
assets/styles/app.css
// ... lines 1 - 18
turbo-frame[busy] .frame-loading-hide, turbo-frame .frame-loading-show {
    display: none;
}
turbo-frame[busy] .frame-loading-show {
    display: inline-block;
}
// ... lines 25 - 180
```

It's a little complicated, but that *should* get the job done and give us two classes that we can reuse across our site. Let's try it! Find your browser, refresh and... perfect! You can already see that my FontAwesome icon is *not* showing up because it's hidden by default. Now click this link. Beautiful!

Loading Opacity

And... that's it! You can leverage this `busy` attribute to do whatever you want. For example, we can give *every* frame on our site loading behavior by lowering their opacity. This is pretty easy. Copy the turbo-frame from above to say that any `turbo-frame` with a `busy` attribute should have `opacity` set to .2. That's an extreme level - but it'll be easy to see.

```
assets/styles/app.css
// ... lines 1 - 18
turbo-frame[busy] {
    opacity: .2;
}
// ... lines 22 - 183
```

When we refresh now, we should even see this during the *initial* load. And... we do! When we click the "read more" link... uh... hmm. I did *not* see the lower opacity. That's weird. Inspect the element... and hack a `busy` attribute on the end of this.

turbo-frame is an Inline Element by Default

Hmm. When I do this, our browser *does* see the correct opacity CSS... it just doesn't seem to be doing anything! Hover over the element... let me scroll up a bit. Check it out: it has no height! I see the arrow in the upper left... but it's not highlighting the element. You'd expect it to go *around* the element like this... but it's not!

So this is interesting. The problem is that `<turbo-frame>` is a custom HTML element. And by default, your browser renders it as an *inline* element. You can see this over in the computed CSS: it has `display: inline`. And so, when you put block elements inside of it, it just... doesn't expand in the way you'd expect it to. That's why it appears to have no height. And *that's* why nothing gets the lower opacity.

To fix this, we can make this element `display: block`. As soon as I hack this in, the opacity *does* take effect. To make this work everywhere, we can make our turbo-frames `display:` `block` by default with `turbo-frame`, `display: block`.

```
assets/styles/app.css
// ... lines 1 - 18
turbo-frame {
    display: block;
}
// ... lines 22 - 186
```

Try it now. The opacity on loading still works and when we click... that works too!

So now that this looks spectacular, let's go and make the opacity a little less @dramatic... and over in `CartController`, take out the sleep.

```
assets/styles/app.css
// ... lines 1 - 18
turbo-frame {
    display: block;
}
turbo-frame[busy] {
    opacity: .7;
}
turbo-frame[busy] .frame-loading-hide, turbo-frame .frame-loading-show {
    display: none;
}
turbo-frame[busy] .frame-loading-show {
    display: inline-block;
}
// ... lines 31 - 186
```

Let's go play with the page. That feels much more natural.

Fixing the Checkout Page

Before we keep going and doing other cool Turbo frame stuff, we accidentally broke the checkout page! It... was my fault.

> *"Variable `showDescription` does not exist"*

Coming from `_featuredSidebar.html.twig`. The template for this page lives at `templates/checkout/checkout.html.twig`.

```
templates/checkout/checkout.html.twig
// ... lines 1 - 10
        <div class="row">
            <aside class="col-12 col-lg-4">
                {% if featuredProduct %}
                    {{ include('cart/_featuredSidebar.html.twig') }}
                {% endif %}
            </aside>

// ... lines 18 - 66
```

Ooooh. This page *also* has a featured product sidebar... and it is *still* using the `include` directly. When we added our new `showDescription` variable, I didn't realize this was being included directly and... well... now things are mad.

We could fix this by passing in the variable... or even coding defensively inside `_featuredSidebar.html.twig`. But, pfff. We have a working, lazy Turbo Frame! So let's just use that! In `cart.html.twig`, steal the lazy frame and paste it inside `checkout.html.twig`.

```
templates/checkout/checkout.html.twig
// ... lines 1 - 10
        <div class="row">
            <aside class="col-12 col-lg-4">
                <turbo-frame id="cart-sidebar" src="{{
path('_app_cart_product_featured') }}" target="_top">
                    Loading...
                </turbo-frame>
            </aside>

// ... lines 18 - 66
```

Celebrate by opening up the controller for this page, which is `CheckoutController`, and removing some variables that we don't need anymore: `addToCartForm` and `featuredProduct`... which means we can delete both variables... and we don't need to inject this argument.

```
src/Controller/CheckoutController.php
// ... lines 1 - 20
    /**
     * @Route("/checkout", name="app_checkout")
     */
    public function checkout(Request $request, CartStorage $cartStorage, EntityManagerInterface $entityManager, SessionInterface $session): Response
    {
        $checkoutForm = $this->createForm(CheckoutFormType::class);

        $checkoutForm->handleRequest($request);
        if ($checkoutForm->isSubmitted() && $checkoutForm->isValid()) {
            /** @var Purchase $purchase */
            $purchase = $checkoutForm->getData();
            $purchase->addItemsFromCart($cartStorage->getCart());

            $entityManager->persist($purchase);
            $entityManager->flush();

            $session->set('purchase_id', $purchase->getId());
            $cartStorage->clearCart();

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

        return $this->renderForm('checkout/checkout.html.twig', [
            'checkoutForm' => $checkoutForm,
        ]);
    }
// ... lines 47 - 65
```

Cool! Refresh now and... all good. The "read more", of course, even works here because Turbo & Stimulus are awesome.

Next: below each product, if you're logged in, users can post a review. We can make this a bit more awesome by leveraging a turbo frame.

Chapter 27: Review this Product... in a turbo-frame!

We have a new mission. But before we jump in, we need to *log* in. Use the delightful cheating links... then head over to a product page and scroll down.

Okay: every product has reviews and we can even *post* a review from right here. There is *nothing* fancy about this: this is a normal HTML form with no custom JavaScript and no turbo frame. And, mostly, it works great! Fill out the form... and submit. Ooh, that's smooth... just because Turbo Drive is awesome.

But notice that we *are* taken to a different page, a `/reviews` page. This is on purpose: management wants to show the reviews below each product... but they also want a dedicated "reviews" *page* for each product. And so, we decided to make the review form *submit* to this page.

This *is* working great... but it *could* be even better if, when we submit a review *from* the product page, we *stayed* on the product page. This is a type of progressive enhancement: everything is cool right now, but we're going to *choose* to enhance things to a higher "coolness level". Doing this is going to require two lines of code.

Adding the Frame

The template for this page is `templates/product/show.html.twig`. At the bottom, the reviews are rendered via this `_reviews.html.twig` template partial. Open that and scroll down to the form. The reason all of this lives in its own partial is that this is *also* included from the reviews page template - `reviews.html.twig`. That lets us show the same list of reviews and form on both pages without duplication.

So let's think: when the "new review" form submits, we want the page to *not* navigate away: we want everything to happen *in* this reviews area. Isn't that... *exactly* what Turbo Frames are for? If we wrapped this entire template in a `<turbo-frame>`... wouldn't that do it? I think it would!

At the top of the template, add `<turbo-frame id="">`, how about, `product-review`. Take the closing tag and put it on the bottom.

```
templates/product/_reviews.html.twig
```

```twig
<turbo-frame id="product-review">
{% for review in product.reviews %}
    <div class="component-light my-3 p-3">
        <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }}
<i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
        <div>
            {{ review.content }}
        </div>
    </div>
{% else %}
    <p>This product has not been reviewed yet!</p>
{% endfor %}

<hr>

{% if reviewForm|default(false) %}
    <h4>Post your own review</h4>
    {{ form_start(reviewForm, {
        'action': path('app_product_reviews', { id: product.id })
    }) }}
        {{ form_row(reviewForm.stars) }}
        {{ form_row(reviewForm.content) }}

        <button class="btn btn-primary" formnovalidate>Add Review</button>
    {{ form_end(reviewForm) }}
{% elseif not is_granted('ROLE_USER') %}
    <p><a href="{{ path('app_login') }}">Log in</a> to post your
review</p>
{% endif %}
</turbo-frame>
```

Those are the 2 lines! Testing time. Refresh, scroll to the bottom of the product show page and submit the form empty. Yes! That was perfect! We see the validation errors but we are *still* on the product show page. This is my favorite example yet of the power of turbo frames. With two lines of code, the entire review system is now self-contained.

Behind the scenes, when we submit this form, it *does* submit to the `/reviews` page. You can see this down in the network tools under the Ajax calls. Here it is: this was a POST request to `/reviews`.

If you look closely at the "preview" for this, it *did* render the *full* reviews page - with header, footer and all. But our `turbo-frame` is smart enough to find *just* the `product-review` frame *inside* this response, grab it and use it.

I love this product *so* much that I think we should publish another another 5 star review. When we submit... gorgeous! Our new review even popped up right above the form!

Changing The Flash Message to Render In the Frame

Though, hmm. There's no success message anywhere on the page. There *was* one before... but now it's gone! What happened?

Look back at the network tools. There are *two* new requests.

The first is a POST request to `/reviews`. That processed our form, was successful, and returned a 302 redirect *back* to the same URL. This caused a *second* Ajax request to be made to `/reviews` and *this* is what was used to fill in the `turbo-frame`.

Look at the preview for this request closely. Near the top - here! The page *does* have a success message! Then, way below, we see the reviews. Can you spot the problem? The success message is being printed *outside* of the turbo-frame. And so, we never see it.

Fortunately, we can fix this pretty easily. Open up the controller that handles the review form submit and renders the reviews page: `src/Controller/ProductController.php`. Here it is: `productReviews()`.

```
src/Controller/ProductController.php
// ... lines 1 - 68
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager)
    {
        $reviewForm = null;

        if ($this->getUser()) {
            $reviewForm = $this->createForm(ReviewForm::class, new Review($this->getUser(), $product));
        }

        if ($request->isMethod('POST')) {
            $this->denyAccessUnlessGranted('ROLE_USER');

            $reviewForm->handleRequest($request);

            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
                $entityManager->persist($reviewForm->getData());
                $entityManager->flush();

                $this->addFlash('success', 'Thanks for your review! I like you!');

                return $this->redirectToRoute('app_product_reviews', [
                    'id' => $product->getId(),
                ]);
            }
        }

        return $this->renderForm('product/reviews.html.twig', [
            'product' => $product,
            'currentCategory' => $product->getCategory(),
            'categories' => $categoryRepository->findAll(),
            'reviewForm' => $reviewForm?: null,
        ]);
    }
// ... lines 104 - 113
```

Let's see: if this is a `POST` request and it's successful, then we set a `success` flash message. Over in `templates/base.html.twig`, we already have code that renders any `success` flash messages near the top of the page.

Now that we're leveraging a frame, what we *really* want to do is render the success message *inside* that frame. Back in the controller, change the flash type from `success` to, how about, `review_success`.

```
src/Controller/ProductController.php
// ... lines 1 - 84
            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
                $entityManager->persist($reviewForm->getData());
                $entityManager->flush();

                $this->addFlash('review_success', 'Thanks for your review! I like you!');

                return $this->redirectToRoute('app_product_reviews', [
                    'id' => $product->getId(),
                ]);
            }
// ... lines 95 - 113
```

Right now, nothing is rendering `review_success` flash messages. But go into the template - `_reviews.html.twig` - and, above the form, render it: for `flash` in `app.flashes('review_success')`. Inside, and an alert div with `alert-success` and print the `flash` variable.

```
templates/product/_reviews.html.twig
// ... lines 1 - 13

{% for flash in app.flashes('review_success') %}
    <div class="alert alert-success">{{ flash }}</div>
{% endfor %}

// ... lines 19 - 33
```

If you want to be fancier, you could isolate the flash logic from `base.html.twig` into its own template and include it from both the base layout *and* `_reviews.html.twig`. That'd be pretty sweet!

Let's go review our product one more time. Do a full page refresh just to be safe, recommend this product to all your friends, submit and... that's *lovely*.

Making One Link target="_top"

Back at the top of the page, click to log out... because there is one *tiny* little detail left. Go back to the product and scroll down to the reviews. You need to be logged in to post a review. But when we click the "log in" link... it's busted!

Check out the console, it's a familiar error:

> *"Response has no matching `<turbo-frame id="product-review">` element."*

Of course. Refresh the page to reset things. When we click the "log in" link, it's now *inside* of a turbo frame. And so, Turbo makes an Ajax call to the login page and looks for a `product-review` frame *on* that page. That is... *not* what we want. We want this link to target the *whole* page. And we know how to do that!

Over in `_reviews.html.twig`, all the way on the bottom, find the link and add `data-turbo-frame="_top"`.

```
templates/product/_reviews.html.twig
// ... lines 1 - 28
{% elseif not is_granted('ROLE_USER') %}
    <p><a href="{{ path('app_login') }}" data-turbo-frame="_top">Log in</a> to post your review</p>
{% endif %}
// ... lines 32 - 33
```

Now when we refresh... and click... we're good!

Next: let's add a bonus feature to our site! Whenever *any* form is submitted on our site for *any* reason, let's automatically disable the submit button to avoid double submits.

Chapter 28: Globally Disable Buttons on Form Submit

Log back in... and head to any product page. Thanks to the work that we did earlier, when we submit the review form, the opacity *does* go lower while the frame is loading. You can see this fairly well on the button... but it *is* still a bit subtle. So here's an idea: what if we also *disabled* this submit button while the frame was loading? That would give us an even *better* loading indicator *and*, as a bonus, it would help prevent double submits. The best part? We can make this happen for *every* form on our site by leveraging an event that Turbo dispatches.

Listening to turbo:before-submit

In your editor, open up `assets/turbo/turbo-helper.js`. Anywhere in the constructor, listen to a new event: `document.addEventListener('turbo:submit-start')`. Pass this an arrow function with an event argument. Inside, let's `console.log()` the string `submit-start` and also the event object.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 8

        document.addEventListener('turbo:submit-start', (event) => {
            console.log('submit-start', event);
        })

// ... lines 14 - 91
```

Turbo triggers this `turbo:submit-start` event whenever *any* form is submitted with turbo, whether it's inside of a Turbo frame or just a normal form that Turbo Drive is handling.

Let's go see if this works. Move over, refresh, submit, and go check the console. There it is!

Now *some* Turbo events have a `detail` key inside them with extra info. And this *is* one of those events. This `formSubmission` key holds all kinds of information about the form submit that's about to start. Most importantly, for us, it has a `submitter` key set to the button that *triggered* the submit. That's this button right here!

This is awesome because we can use that to add a `disabled` attribute! The path to this is `detail.formSubmission.submitter`.

Disabling the Submitter Button

Head back to our code and replace the log with `event.detail.formSubmission.submitter`. Add the `disabled` attribute with `.toggleAttribute('disabled', true)`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 8

        document.addEventListener('turbo:submit-start', (event) => {
            event.detail.formSubmission.submitter.toggleAttribute('disabled', true);
        })

// ... lines 14 - 91
```

When you use `toggleAttribute` with a second argument of `true`, it means:

> *"I want you to add this attribute... but I don't need it to be `disabled="something"`. I just need the `disabled` attribute."*

Let's try that. Refresh the page... and then inspect the button element. Watch it when I click. Yes! Perfect! For just a moment, it had a `disabled` attribute, which made it even *more* obvious that it was loading. *And*, we can't click to submit it twice.

Behind the scenes, *our* code added the `disabled` attribute. Then, when the frame finished loading, the entire contents of the frame were replaced with a new, non-disabled form to give us the exact effect we want.

Fixing Disabled Forms in Turbo Snapshots

Scroll up, log out, then go to the registration form. This form does *not* live in a Turbo frame. But it *still* gets the new submit behavior! Yup, with just a few lines of code, *every* form on our site just got a little fancier.

But... there is one... super edge case. If you submitted the form and navigated away from the page *while* the form was still submitting, that would cause Turbo to take a snapshot of the page *with* the disabled button. If the user then clicked *back* on their browser, the button would *still* be disabled.

This is probably *such* a rare edge case that... maybe we don't care. But let's code for it.

Back in `turbo-helper.js`, create a new variable: `const submitter =`. Copy the `event.detail` line from below, paste here, and just use `submitter` below.

We're doing this so we can *also* give this button a new class: `submitter.classList.add('turbo-submit-disabled')`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 9

        document.addEventListener('turbo:submit-start', (event) => {
            const submitter = event.detail.formSubmission.submitter;
            submitter.toggleAttribute('disabled', true);
            submitter.classList.add('turbo-submit-disabled');
        })

// ... lines 17 - 101
```

This class doesn't do anything and doesn't have any CSS attached to it. I just invented it as a way to *mark* that this button was disabled *because* of our loading logic.

Why is that helpful? Above this, we're listening to `turbo:before-cache`. This is called right *before* Turbo takes a snapshot of the page. We can *use* the `turbo-submit-disabled` class to *find* the disabled button and *remove* that attribute.

But let's not put the logic here: let's call a new function: `this.reenableSubmitButtons()`.

Copy that method name, scroll *all* the way to the bottom and paste to create it. Inside, use `document.querySelectorAll()` to find any element with the `turbo-submit-disabled` class that we added. Foreach over this, pass a callback with a `button` argument, and then say: `button.toggleAttribute('disabled', false)`. Fully clean things up by removing the class: `button.classList.remove('turbo-submit-disabled')`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 4
        document.addEventListener('turbo:before-cache', () => {
            this.closeModal();
            this.closeSweetalert();
            this.reenableSubmitButtons();
        });
// ... lines 10 - 90

    reenableSubmitButtons() {
        document.querySelectorAll('.turbo-submit-disabled').forEach((button) => {
            button.toggleAttribute('disabled', false);
            button.classList.remove('turbo-submit-disabled');
        });
    }
// ... lines 98 - 101
```

It's pretty hard to actually *repeat* the edge case we just fixed... but let's at least make sure we didn't break anything. Submit the form. Yup! That still looks great!

Next: there's another place that we can leverage a Turbo Frame to do something cool. While viewing a product, *if* we're an admin, it would be awesome to be able to click an "edit" button that would Ajax load the "product form" right into this space. So... let's do it!

Chapter 29: Frame-Powered Inline Editing

Make sure you're logged in... and then head over to any product page. We already have a product admin section. And since we *are* an admin - lucky us - we can use it to edit any product. To make life cooler for admin users, let's add an edit link right on the public show page.

Easy enough: open the template for this page - `templates/product/show.html.twig` - find the `h1` and move it onto multiple lines. Then add if `is_granted('ROLE_ADMIN')` and `endif`. Inside, we can create a boring anchor tag that points to the edit page: `path('product_admin_edit')` with `id` set to `product.id`.

Oh, but I'm going to put this onto multiple lines in a slightly different way... so that we can cleanly give this a few classes. For the text, say "Edit".

```
templates/product/show.html.twig
// ... lines 1 - 16
            <h1>
                {{ product.name }}
                {% if is_granted('ROLE_ADMIN') %}
                    <a
                        href="{{ path('product_admin_edit', {
                            id: product.id
                        }) }}"
                        class="btn btn-sm btn-secondary"
                    >Edit</a>
                {% endif %}
            </h1>
// ... lines 28 - 52
```

Nothing magic yet. When we refresh, there's our link... a fantastically boring edit link. Thanks to Turbo Drive, clicking it feels pretty good. And with a bit more work, we could add a link back to the public show page. Heck, we could even attach a query parameter when we click this edit button - like `?from=` - and use that on the admin page to dynamically link back to the admin index page - like it is now - *or* back to the product show page *if* that's where we originally came from. We could even go further and also make the form *redirect* back to that page after success. My point is, thanks to the smoothness of Turbo Drive, there are many ways that we could make this process *even* smoother simply by writing a little Twig & PHP code.

But instead of doing any of those, let's progressively enhance this in a different way: by making the edit link load the form *right* onto the public show page. That sounds like a job for a turbo frame!

Adding the turbo-frame

Head back to the template and scroll to the top. Okay: we have a `col-4` and a `col-8` - that's the left and right sides of this page. Our new mission is to wrap that *entire* area in a `turbo-frame` so that it can be *replaced* by the edit form. So basically, we need a frame right *inside* of this "row" div.

Say `<turbo-frame id=""` and call it, how about `product-info`. I'm also going to add a `target="_top"` so that everything inside, at least for now, will behave *completely* normally: as if there were *no* frame.

Take the `turbo-frame` closing tag and... put it all the way down here: I think this is the right spot.

```
templates/product/show.html.twig
// ... lines 1 - 2
{% block productBody %}
    <div class="row pt-3 product-show">
        <turbo-frame id="product-info" target="_top">
        <div class="col-4">
            <img
                alt="{{ product.name }}"
                src="{{ asset('/uploads/products/'~product.imageFilename) }}"
                class="d-block"
            >
            <div class="p-2">
                <small>brought to you by </small>
                <small class="d-inline">{{ product.brand }}</small>
            </div>
        </div>
        <div class="col-8 px-3">
            <h1>
                {{ product.name }}
                {% if is_granted('ROLE_ADMIN') %}
                    <a
                        href="{{ path('product_admin_edit', {
                            id: product.id
                        }) }}"
                        class="btn btn-sm btn-secondary"
                    >Edit</a>
                {% endif %}
            </h1>
            <div>
                {{ product.description }}
            </div>
            <div class="p-3 mt-4 d-flex justify-content-between flex-wrap flex-lg-nowrap">
                <div>
                    <strong>{{ product.priceString|format_currency('USD') }}</strong>
                    <br>
                    <strong>{{ product.reviews|length }}</strong> Reviews
                    <br/>
                    <strong>{{ product.averageStars }}/5</strong><i class="fas fa-star ms-2"></i>
                </div>
                <div>
                    {{ include('product/_cart_add_controls.html.twig') }}
                </div>
            </div>
            </div>
        </turbo-frame>
    </div>
// ... lines 47 - 54
```

Let's see how things look so far. Refresh and... whoa! That *completely* messed up our styling! Why? Inspect element on this area. The problem is that we added an element *between* the row and the columns... and with CSS Flexbox, sometimes the direct relationship between elements *is* important. By putting this `turbo-frame` in the middle, we angered the Flexbox gods!

Using turbo-frame as a Normal Element

So what can we do? One obvious idea is to move the `turbo-frame` *around* the `row` div so that we don't interrupt the row-column relationship. That *would* work.

But... `turbo-frame` is just a normal HTML element... so we could also *change* the `row` element from a `div` to a `turbo-frame`!

Check it out: delete the `turbo-frame` closing tag. Then, on top, copy the guts from the `turbo-frame`, change the `div` to a `turbo-frame` and re-add `id` and `target`. Down on the closing tag, ah nice! PhpStorm already changed that for me.

```
templates/product/show.html.twig
// ... lines 1 - 3
    <turbo-frame id="product-info" target="_top" class="row pt-3 product-show">
        <div class="col-4">
            <img
                alt="{{ product.name }}"
                src="{{ asset('/uploads/products/'~product.imageFilename) }}"
                class="d-block"
            >
            <div class="p-2">
                <small>brought to you by </small>
                <small class="d-inline">{{ product.brand }}</small>
            </div>
        </div>
        <div class="col-8 px-3">
            <h1>
                {{ product.name }}
                {% if is_granted('ROLE_ADMIN') %}
                    <a
                        href="{{ path('product_admin_edit', {
                            id: product.id
                        }) }}"
                        class="btn btn-sm btn-secondary"
                    >Edit</a>
                {% endif %}
            </h1>
            <div>
                {{ product.description }}
            </div>
            <div class="p-3 mt-4 d-flex justify-content-between flex-wrap flex-lg-nowrap">
                <div>
                    <strong>{{ product.priceString|format_currency('USD') }}</strong>
                    <br>
                    <strong>{{ product.reviews|length }}</strong> Reviews
                    <br/>
                    <strong>{{ product.averageStars }}/5</strong><i class="fas fa-star ms-2"></i>
                </div>
                <div>
                    {{ include('product/_cart_add_controls.html.twig') }}
                </div>
            </div>
        </div>
    </turbo-frame>
// ... lines 45 - 52
```

When we refresh now... it looks good again! But because our frame has `target="_top"`... the frame doesn't *do* anything yet: the edit link still navigates the *entire* page.

To fix that, find the link... which is down here... and make it target the frame: `data-turbo-frame="product-info"`.

```
templates/product/show.html.twig
// ... lines 1 - 16
            <h1>
                {{ product.name }}
                {% if is_granted('ROLE_ADMIN') %}
                    <a
                        href="{{ path('product_admin_edit', {
                            id: product.id
                        }) }}"
                        class="btn btn-sm btn-secondary"
                        data-turbo-frame="product-info"
                    >Edit</a>
                {% endif %}
            </h1>
// ... lines 29 - 53
```

Will this work? Not *quite*... and you may remember why. Refresh and click Edit. The whole area disappeared! And we see our favorite error in the console:

> *“Response has no matching `<turbo-frame id="product-info">` element.”*

Of course! The page that that we're navigating to - the product admin edit page - *must* also have a `product-info` frame.

The template for that product admin edit page lives at `templates/product_admin/edit.html.twig`. The actual *form* lives inside `_form.html.twig`. So we *could* add the `turbo-frame` here around the form. But I kind of *do* want the "edit product" `h1` and the "delete form" button to *also* be loaded when we click "edit". So let's add the `turbo-frame` right here.

After the back button - because we don't want to include that - add `<turbo-frame id="product-info">`. I'm also going to add `target="_top"` here to guarantee that, by default, any links or forms inside here continue to behave like normal when we navigate directly to the product admin page.

Add the closing frame tag and indent everything.

```
templates/product_admin/edit.html.twig
```

```
// ... lines 1 - 4
{% block body %}
<div class="container mt-4">
    <a href="{{ path('product_admin_index') }}"><i class="fas fa-caret-left"></i> Back to list</a>
    <turbo-frame id="product-info" target="_top">
        <div class="d-flex justify-content-between">
            <h1 class="mt-3">Edit Product</h1>
            {{ include('product_admin/_delete_form.html.twig') }}
        </div>

        {{ include('product_admin/_form.html.twig', {'button_label': 'Update'}) }}
    </turbo-frame>
</div>
{% endblock %}
```

That should do it! Refresh the page... and click edit. Sweet! We see the form but we're *still* on the product show page!

So far, this has been pretty easy: a perfect use-case for Turbo Frames. Let's take a victory lap!

Except... something isn't quite right. If we change the title and submit the form... woh! That looked like a full page refresh! Let's find out what's going on next, fix it, and complete our inline editing destiny!

Chapter 30: Frames & Form "action" Attributes

Something isn't right. We *can* click this "edit" link to inline-load the product form into the Turbo Frame. But when we save, something weird happens. Watch the console closely down here. Whoa! It was fast, but it looked the Ajax request failed! And then, the whole page reloaded?

Time to put on our detective hats! Let's start by getting more information about why the form submit failed. Click any link on the web debug toolbar to jump into the profiler... and then click the "last 10" link to see the last 10 requests.

Ah, here! A 405 error. Open the profiler for that page:

> *“No route found for POST /product/1: Method not allowed”*

Wait: look at the URL. That is *not* the right URL! The form should submit to the product admin area, which... if you navigate there, looks like this: `/admin/product/12/edit`. But the form *actually* submitted to the public product show page. Why?

Close this tab and hit edit again. Actually, refresh, hit edit and inspect element on the form. Ah ha! The form element does *not* have an `action` attribute. Normally this is fine! If you go to the product admin page and click to edit a product, the form doesn't have an `action` attribute here either. That's ok because when a form doesn't have an `action` attribute, it tells your browser to submit to the URL that it's currently on. For this page, that's perfect.

But when we're on the public product show page... and we load the same form, having that missing `action` attribute is *not* okay: our browser incorrectly thinks it should submit to `/product/1`.

Here's the takeaway: if you're planning to load a form into a `turbo-frame`, that form *does* need an `action` attribute. We can't be lazy like we normally are.

Setting the Form action

We can set the action attribute in a few places, but I like to do it in the controller where we create the form. Open the controller for the product admin area: `src/Controller/ProductAdminController.php`. Right now we're only dealing with the edit page, but I'll set the action on both the new *and* edit actions to be safe. Add a third argument to `createForm()` and pass an option called `action` set to the URL to *this* action: `$this->generateUrl('product_admin_new')`.

Now scroll down to the one that we really care about: the edit action. Same thing here: pass a third argument with `action` set to `$this->generateUrl('product_admin_edit')`... but this needs an `id` wildcard set to `$product->getId()`.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 31
    /**
     * @Route("/new", name="product_admin_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    {
        $product = new Product();
        $form = $this->createForm(ProductType::class, $product, [
            'action' => $this->generateUrl('product_admin_new'),
        ]);
        $form->handleRequest($request);

// ... lines 43 - 62
    /**
     * @Route("/{id}/edit", name="product_admin_edit", methods=
{"GET","POST"})
     */
    public function edit(Request $request, Product $product): Response
    {
        $form = $this->createForm(ProductType::class, $product, [
            'action' => $this->generateUrl('product_admin_edit', [
                'id' => $product->getId(),
            ]),
        ]);
        $form->handleRequest($request);
// ... lines 74 - 101
```

Time to give this a try! Refresh the page, click edit, change the title and submit the form. Very nice... kind of. If you scroll down to find this product... yes! It *did* update the title!

But, as we can see, it redirected to the product admin list page, not the product show page. When we click this "edit" button, that *does* load the form into the Turbo frame. But then, because the frame has `target="_top"`, when we submit the form, it submits to the *whole* page and *navigates* the whole page. That's why hitting save redirects us to a totally different page.

Redirecting to the Product Show Page

And that's maybe okay: this is already a better experience than when we started. But we could make it a bit more awesome by redirecting back to the public product show page. Let's try that: I'll do it in just the edit action. On success, change the index route to `app_product` - the route for the show page - and pass this the `id` wildcard that it needs.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 73

        if ($form->isSubmitted() && $form->isValid()) {
            $this->getDoctrine()->getManager()->flush();

            return $this->redirectToRoute('app_product', [
                'id' => $product->getId(),
            ]);
        }
// ... lines 82 - 103
```

Let's see how this feels. Open up the floppy disk public show page, hit edit, change the title and submit. That's very nice!

Edit the product again, but empty the title so that we fail validation. When we submit now, this navigate us away from the show page and puts us in the admin section. That makes complete sense: we know that the form is *still* submitting to the full page, not to the frame. And so, again, this is probably okay! We should probably stop and say "good enough!".

Submitting the Form in the Frame

Or... we could *also* make the form *submit* in the frame.

To do this, we have two options. Over in `show.html.twig`, we have `target="_top"` on the `turbo-frame`. The first way that we could make the form submit to the frame would be to remove this target so that *everything* navigates inside the frame. Of course, if we did that, we would need to make sure to add `data-turbo-frame="_top"` to any links or forms that *should* target the full page.

The other option is to leave the `target="_top"` and then, on *just* the product form, add `data-turbo-frame="product-info"`.

For me, the *best* option is still... not totally clear. Is it better to add `target="_top"` on the frame and then target the frame on individual links and forms? Or should we leave `target="_top"` *off* the frame and add `target="_top"` to the individual links and forms that need it?

I don't have a perfect answer. But my rule of thumb is to determine this based on the *main* purpose of a frame. In this case, I would expect *most* links to navigate the whole page, so the `target="_top"` on the *frame* feels safer.

So let's go change the target of *just* the form. The edit page template is `edit.html.twig`, but the form lives in `_form.html.twig`. Pass a second argument to `form_start` with an `attr` variable set to an object. Inside *that*, set `data-turbo-frame` to `product-info`.

templates/product_admin/_form.html.twig

```
{{ form_start(form, {
    attr: { 'data-turbo-frame': 'product-info' }
}) }}
    {{ form_widget(form) }}
    <button class="btn btn-primary" formnovalidate>{{
button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Let's try the flow! Refresh. We have a `turbo-frame` with `target="_top"`... but inside, an edit link that specifically targets the frame. When we click this, the new form is *still* in the frame with `target="_top"`... but it *also* targets the `product-info` frame.

Thanks to this, if we empty the title and submit... woohoo! That keeps us on the page! That submitted *into* the frame. And if we put the title back, change it and submit. Beautiful!

Next: when we submit a form inside a frame... and that request redirects to *another* page, what happens? Does that redirect the entire page and change the URL in the address bar? Or does it *only* update the frame? Let's find out and fix a related bug with our new inline edit frame system.

Chapter 31: Frame Redirecting & Dynamic Frame Targets

It was subtle, but we just saw one important property of Turbo frames. When we submitted this form successfully, it submitted to the edit action inside of `ProductAdminController`. This code handled the form submit and, because it was successful, it redirected to the public product show page.

It turns out, if you submit a form in a frame and that Ajax request *redirects* to another page, Turbo does *not* follow the redirect and navigate the entire page. Well, let me be more clear.

Redirects Do Not Move the Entire Page

Check out the network tools. This POST request was for the *unsuccessful* form submit we did a minute ago: the one that failed validation. This second request was for our *successful* form submit. And you can see that it returned a 302 redirect. When Turbo sees a redirect, it *does* follow it in a sense... it makes a second Ajax call to the redirected URL: the product show page. This is also how Turbo Drive works... but with one key difference: after making the second Ajax request, a Turbo frame does *not* navigate the entire page and update the URL in our browser to match the redirected URL.

Nope, because we submitted to a turbo frame, it reads the HTML of this redirected page, finds the `product-info` frame and loads just *that* into the frame.

This is... kind of hard to see in *this* case, because it's redirecting back to the URL that is *already* in our address bar. But this *is* the behavior: if you submit a form inside a frame, even if that request redirects, all navigation will *stay* inside the frame.

Actually, there is a *super* obvious place where we can see this. Go to the product admin area and edit a product. Like with the show page, the frame is targeting `_top` but the form is targeting `product-info`. If we clear out the title and submit, it submits to the frame and looks fine.

But if we put the title back, change it and submit, watch what happens. Ah! Frankenstein page! Half of the public product page just exploded onto this admin page!

Unfortunately... the turbo frame is doing *exactly* what we're asking it to do. Look at the network tools... and scroll up a bit. We submitted successfully to the edit page and that redirected to the public show page. Then, because we're submitting in a turbo-frame, the frame found the `product-info` frame *on* that page - which is all this product info - grabbed it, and popped it right here.

In the admin area... this is *not* what we want. And things are getting a bit complicated as a result of us *really* pushing for the best possible user experience.

So let's stop and think. When we load the form from the product show page and hit edit, we *do* want this form to submit *into* the frame. But when we load that same form in the product admin area, we kind of just want this to behave like *normal*, by submitting *to* the entire page. Could we do that? Could we make the same form behave *differently* based on the situation? Totally!

The Turbo-Frame Request Header

Head to `ProductAdminController`'s edit action. Whenever turbo is navigating inside a frame, it sends an extra header called `Turbo-Frame` with the name of the frame. So when we click the edit link from the product show page, that Ajax request *will* add a `Turbo-Frame` header. You can see it all the way down here under request headers... there it is: `Turbo-Frame: product-info`.

But when navigate directly to the product admin area and look at *that* Ajax request, down here, there is *no* `Turbo-Frame` header. This means we can detect whether a request is being loaded inside a turbo frame from inside of Symfony!

Back in the controller, when we render the template, pass in a new variable called `formTarget` set to `$request->headers->get('Turbo-Frame')`. If that header was *not* sent, add a second argument to default this to `_top`.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 61

    /**
     * @Route("/{id}/edit", name="product_admin_edit", methods={"GET","POST"})
     */
    public function edit(Request $request, Product $product): Response
    {
// ... lines 68 - 82
        return $this->renderForm('product_admin/edit.html.twig', [
            'product' => $product,
            'form' => $form,
            'formTarget' => $request->headers->get('Turbo-Frame', '_top')
        ]);
    }
// ... lines 89 - 104
```

Now in `_form.html.twig`, instead of setting the target to `product-info`, use the `formTarget` variable. And because this template is *also* included on the new product page... and we're *not* setting this variable there, code defensively by defaulting it to `_top`.

```
templates/product_admin/_form.html.twig
{{ form_start(form, {
    attr: { 'data-turbo-frame': formTarget|default('_top') }
}) }}
    {{ form_widget(form) }}
    <button class="btn btn-primary" formnovalidate>{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

I *think* that's going to do it! Refresh the product admin page and hit save. Beautiful! That submitted to the entire page and *redirected* the entire page. Now click edit, empty the title and hit enter. Yes: this *still* navigates inside the frame. If you inspect element on the form, you can see that it *does* have the extra `data-turbo-frame` attribute set to `product-info`.

So, inline product admin form done! I included this example both because it's really cool to load the form inline... but also because it shows a situation where turbo frames *can* get a bit complex. It's up to you to balance the added complexity with the user experience that you want.

Next: what about using a turbo frame inside of a modal? After all, you often want navigation - like links and form submits inside of a modal - to *stay* inside of that modal... which is what turbo frames are really good at. So let's transform this modal into a turbo-frame powered modal.

Chapter 32: turbo-frame inside a Modal

Let's do one more big thing with the frame system. Go to the product admin page and click to add a new product. In the last tutorial, we used Stimulus to open this in a modal, make this form *submit* via Ajax inside the modal, make the modal close on success and then reload the list with Ajax. An entire experience with no full page refreshes.

The stimulus controller for this lives at `assets/controllers/modal-form_controller.js`. This `openModal()` is called when we click to add a new product: it opens the modal and makes an Ajax call to *populate* that modal with the form HTML. The `submitForm()` is called when the form is submitted and its job is to Ajax-submit the form and close the modal on success.

We're revisiting this example because, by leveraging Turbo frames, I think we can simplify this... like, a lot. And you can probably guess how: we can use a turbo frame to load the initial contents of the modal *and* to make the form submit *stay* in the modal.

Refactoring to a turbo-frame

The modal's markup lives in `templates/_modal.html.twig` and this is meant to be reusaable in multiple places. This `modal-body` element holds the actual *content*.

Let's transform this into a `<turbo-frame>`. To keep things usable, set the frame's `src=""` to a new `modalSrc` variable that we will pass *into* this template.

```
templates/_modal.html.twig
<div
    class="modal fade"
    tabindex="-1"
    aria-hidden="true"
    data-modal-form-target="modal"
>
// ... lines 7 - 14
            <turbo-frame
                class="modal-body"
                data-modal-form-target="modalBody"
                data-action="submit->modal-form#submitForm"
                src="{{ modalSrc }}"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
// ... lines 23 - 32
</div>
```

Now open the template for the product admin list page: `templates/product_admin/index.html.twig`. There's a lot going on here: we activate the `modal-form` Stimulus controller here. We also have a Stimulus controller for `reload-content`. It's job was to reload the product list after the modal closed successfully. We're going to be removing *a lot* of this stuff soon.

What I want to focus on right now is down here where we *include* that modal. Pass in that new `modalSrc` variable set to `path('product_admin_new)` because that's the page that holds the "new product form" that we want.

```
templates/product_admin/index.html.twig
{% extends 'base.html.twig' %}

{% block title %}Product index{% endblock %}

{% block body %}
<div
    class="container-fluid container-xl mt-4"
    {{ stimulus_controller('reload-content', {
        url: path('product_admin_index', { ajax: 1 })
    }) }}
    data-action="modal-form:success->reload-content#refreshContent"
>
// ... lines 13 - 15
        <div
            {{ stimulus_controller('modal-form', {
                formUrl: path('product_admin_new')
            }) }}
        >
            <button
                class="btn btn-primary btn-sm"
                data-action="modal-form#openModal"
            >+ Add new product</button>

            {{ include('_modal.html.twig', {
                modalTitle: 'Add a new Product',
                modalSrc: path('product_admin_new'),
            }) }}
        </div>
// ... lines 31 - 37
</div>
{% endblock %}
```

Before we try this, let's delete some code in `modal-form_controller.js`. In `openModal()`, we don't need to set the `innerHTML` to "Loading" - that can live directly in the frame - and... we don't need to manually make an Ajax call at all! That's going to happen automatically just because we're setting the `src` attribute on the `<turbo-frame>`.

Also `submitForm()`... yea, we're not going to need this at *all*. The turbo frame will handle the form submit all on its own. And thanks to these changes, one of the targets up on top - `modalBody` - is no longer used. So we can remove that too.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 4

export default class extends Controller {
    static targets = ['modal'];
    static values = {
        formUrl: String,
    }
    modal = null;

    connect() {
        useDispatch(this);
    }

    async openModal(event) {
        this.modal = new Modal(this.modalTarget);
        this.modal.show();
    }
}
```

Yup, the job of this controller is getting... pretty simple!

Back in `_modal.html.twig`, to finish our cleanup, we don't need the `modalBody` target... and we also don't need the `data-action` that called the `submitForm` method that we just deleted.

```
templates/_modal.html.twig
// ... lines 1 - 14
            <turbo-frame
                class="modal-body"
                src="{{ modalSrc }}"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
// ... lines 21 - 32
```

Forgetting the id Attribute

Ok team: let's try this! Refresh the page. Hmm, nothing happened. In the console, whoa!

> *"Failed to execute `querySelector` on element: `turbo-frame#` is not a valid selector."*

What is that? Well, it's not a great error, but something is looking for a `turbo-frame` with a certain id - that's this `#` part. But oh! I forgot to give our frame an id! Whoops.

Head back to `_modal.html.twig`. I want to keep this dynamic because different modals may need different frame ids. So say `id="{{ id }}"`.

```
templates/_modal.html.twig
// ... lines 1 - 14
            <turbo-frame
                class="modal-body"
                src="{{ modalSrc }}"
                id="{{ id }}"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
// ... lines 22 - 33
```

Over in `index.html.twig`, pass in the new `id` variable set to `product-info`. That's the `id` we've been using... and it really could be *anything*, as long as it matches a frame on the new product page.

```
templates/product_admin/index.html.twig
// ... lines 1 - 24

            {{ include('_modal.html.twig', {
                modalTitle: 'Add a new Product',
                modalSrc: path('product_admin_new'),
                id: 'product-info',
            }) }}
// ... lines 31 - 41
```

Ok: let's keep trying. Refresh and add a new product. Error!

> *“Response has no matching `<turbo-frame id="product-info">` element.”*

Ah, I remember. In `edit.html.twig`, we added a `<turbo-frame>` *there*... but we never added the `<turbo-frame>` in `new.html.twig`. We could just move the `turbo-frame` into `_form.html.twig` because that's included on both pages. The disadvantage is that we added the frame in `edit.html.twig` on purpose so that our inline editing feature would include the "edit product" `h1` tag and the delete button. So instead, let's just add the same `<turbo-frame>` over here in `new.html.twig`.

```
templates/product_admin/new.html.twig
// ... lines 1 - 4
{% block body %}
<div class="container mt-4">
    <a href="{{ path('product_admin_index') }}"><i class="fas fa-caret-left"></i> Back to list</a>
    <h1 class="mt-3">Create new Product</h1>

    <turbo-frame id="product-info" target="_top">
        {{ include('product_admin/_form.html.twig') }}
    </turbo-frame>
</div>
{% endblock %}
```

Attempt number 3! Refresh and click. Got it!

Using Real Buttons vs Modal Footer

But if we try to submit this... error!

> *"Error invoking action `click->modal-form#submitForm`."*

Ok, so something is *still* trying to call the `submitForm()` method that we deleted a few minutes ago. In `_modal.html.twig`, this is coming from the `modal-footer`. In this last tutorial, we added a button down here to submit the form. But this button is actually *outside* of the form, which lives in the `turbo-frame`. What we need to do, yet again, is simplify. Remove the `modal-footer` entirely.

templates/_modal.html.twig

```
<div
    class="modal fade"
    tabindex="-1"
    aria-hidden="true"
    data-modal-form-target="modal"
>
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">{{ modalTitle }}</h5>
                <button type="button" class="btn-close"
                        data-bs-dismiss="modal"
                        aria-label="Close"></button>
            </div>
            <turbo-frame
                class="modal-body"
                src="{{ modalSrc }}"
                id="{{ id }}"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
        </div>
    </div>
</div>
```

If you refresh and open the form... the footer buttons are gone... but there is now *no* submit button on the form! Well, there *is* one, but it's hiding: you can see it if you inspect element and do some digging. Yup, we hid this button in the last tutorial when it's inside a modal via CSS so that the modal-footer buttons could take precedence. Now, we're going to *undo* that so that our form is perfectly boring and normal: a form... with a button.

Open `assets/styles/app.css` and search for `modal-body`. Delete this section.

Try the modal again... and... it works! And it's *so* boring, I absolutely love it. Try to submit the form. Um, well... that *did* work, but it submitted the whole page! Next, let's fix this, make the modal load lazily and delete even *more* code from the modal system.

Chapter 33: Lazy Modal & Big Cleanup

Our modal *is* now powered by a `turbo-frame`: the form was Ajax loaded by the frame system. But when we submit, wah, wah. It submits to the whole page.

Let's see what's going on. Reopen the modal and inspect it. Hmm. Ah, look at the `form`. It has `data-turbo-frame="_top"`! That's coming from `_form.html.twig`.

```
templates/product_admin/_form.html.twig
```

```twig
{{ form_start(form, {
    attr: { 'data-turbo-frame': formTarget|default('_top') }
}) }}
    {{ form_widget(form) }}
    <button class="btn btn-primary" formnovalidate>{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Remember: a few minutes ago, we set the `data-turbo-frame` attribute to a dynamic `formTarget` variable. The point of this was so that *if* the form is being loaded into a frame, then we *target* that frame. Else, if the form is being loaded via a normal page load, target `_top`.

The problem is that... we only set the variable for the *edit* page. Open `src/Controller/ProductAdminController.php`. Right here - this is the `edit()` action - we *did* pass in the `formTarget` variable that's set to the `Turbo-Frame` request header. Go us! But... I did *not* do that for the `new` action. And since that does *not* pass a `formTarget` variable, it defaulted to `_top`.

```
src/Controller/ProductAdminController.php
```

```php
// ... lines 1 - 81

        return $this->renderForm('product_admin/edit.html.twig', [
            'product' => $product,
            'form' => $form,
            'formTarget' => $request->headers->get('Turbo-Frame', '_top')
        ]);
    }
// ... lines 89 - 104
```

So let's pass that variable in for the new page as well. This is yet *another* spot where, to get this turbo-frame-powered modal working, we're making things simpler and more consistent.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 31
    /**
     * @Route("/new", name="product_admin_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    {
// ... lines 37 - 55

        return $this->renderForm('product_admin/' . $template, [
            'product' => $product,
            'form' => $form,
            'formTarget' => $request->headers->get('Turbo-Frame', '_top')
        ]);
    }
// ... lines 63 - 105
```

Ok: refresh again, open the modal, submit and... oh, that is positively heart-warming.

Lazy Modal Loading

We still need to work on what happens when we submit the form *successfully*... but before we do, let's do something cool. Refresh the page and inspect element on the button. Dig a little to find the `turbo-frame` that contains the modal. Here it is. If you expand this, you'll notice that Turbo *has* already made the Ajax request for the form and put the HTML here. That happens as *soon* as the page loads.

But we don't *really* need to make that Ajax call until the modal opens. Could we somehow *delay* that? Totally! And we did this earlier.

In `_modal.html.twig`, on the `turbo-frame`, add `loading="lazy"`.

```
templates/_modal.html.twig
// ... lines 1 - 14
            <turbo-frame
                class="modal-body"
                src="{{ modalSrc }}"
                id="{{ id }}"
                loading="lazy"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
// ... lines 23 - 26
```

Let's see how this looks. Refresh and inspect the frame. It still says "Loading": it has *not* made the Ajax request yet. Open your network tools and watch the Ajax requests. Click to open the modal! There's the Ajax call!

Remember: with `loading="lazy"`, the frame system won't make the Ajax request until the frame becomes *visible* in the viewport. And... that works pretty awesomely with modals which don't become visible until you open them.

Big Ol' Cleanup

At this point, if you look at the `modal-form` controller, its only job is to... open the modal! The `turbo-frame` inside handles the rest... and that's *pretty* cool. Let's cleanup a few more things: we don't need `useDispatch` anymore: we're not dispatching any events... whoops. And... we don't need to import `useDispatch` or `jQuery`... and we can also delete the `formUrl` value.

```
assets/controllers/modal-form_controller.js
```

```javascript
import { Controller } from 'stimulus';
import { Modal } from 'bootstrap';
import $ from 'jquery';
import { useDispatch } from 'stimulus-use';

export default class extends Controller {
    static targets = ['modal'];
    static values = {
        formUrl: String,
    }
    modal = null;

    connect() {
        useDispatch(this);
    }

    async openModal(event) {
        this.modal = new Modal(this.modalTarget);
        this.modal.show();
    }
}
```

Cool. In the template for the product index page, we still *do* need the `modal-form` controller but we do *not* need to pass in the `formUrl` variable.

Above this, we have some fanciness with the `reload-content` controller. That helped us reload the product list via Ajax after the modal closed. We're going to completely replace that with something simpler in a few minutes. So delete *all* of that stuff.

Finally, near the bottom, remove this target, which was for the `reload-content` controller.

Honestly, I'm wondering if it might have been easier to start this feature from scratch! Because most of the work we just did was deleting and simplifying.

```
templates/product_admin/index.html.twig
// ... lines 1 - 4
{% block body %}
<div
    class="container-fluid container-xl mt-4"
>
    <div class="d-flex flex-row">
        <h1 class="me-3 mb-4">Product index</h1>

        <div
            {{ stimulus_controller('modal-form') }}
        >
// ... lines 15 - 25
    </div>

    <div class="table-responsive">
        {{ include('product_admin/_list.html.twig') }}
    </div>

    <a href="{{ path('product_admin_new') }}">Add a new product</a>
</div>
{% endblock %}
```

Let's make sure we didn't break anything. Refresh, open the modal and submit the form empty. That feels great!

But what happens on a *successful* form submit? Fill in a title, price and... go! Woh. That's... interesting. It says "loading". Next, let's figure out what just happened. And then, we'll code up the *real* solution: after a successful form submit, we want to close the modal and reload the list behind us. We're about to bend the frame system to our will!

Chapter 34: Close the Modal after turbo-frame Success

We just submitted the form in the modal *successfully* and... well, this happened. Weird. If you refresh, the submit *did* work: this is our new product on top. Inspect element on the frame so we can see what's going on... it's interesting and... subtle. Dig a little to find the frame.

Ok, the `src` starts set to `/admin/product/new`, which means that when we open the modal, we see the contents of the `turbo-frame` from that page. Fill in some data and then submit.

Hmm, the `src` changed to `/admin/product/`. Well, that *does* make sense: if you look in `ProductAdminController`, after success, the controller redirects to `/admin/product` - this is inside of the `new` action.

So we submit to `/admin/product/new` and it redirects to `/admin/product/`. When that happens, the frame system does *two* things. First, it makes a second request to the redirected URL - `/admin/product`. We've seen that before. And second, it updates the `src` attribute to match the redirected URL.

This is all perfectly expected. Open the network tab. The second to last request is the POST request to `/admin/product/new`. That's the form submit. And the *last* request is Turbo following the redirect to `/admin/product/`.

Look at the response for that request... let's actually look at the raw HTML. Let's see if we can dig and find the `turbo-frame`. There it is! Yup, it contains nothing more than "Loading...". *That* is what we're seeing in the modal.

Remember: when the frame system finds a matching `turbo-frame`, it only takes the frames *HTML*: it does *not* also use the new frame's `src` attribute or anything else. So even though this frame has `src="/admin/product/new`, that is *not* used. It grabs the "Loading..." text and... that's it!

So once again, Turbo is behaving exactly like it should... but not necessarily how we want!

Speaking of that... how *do* we want this to work? If we wanted the modal to *stay* open but show a new, empty form, we could simply change the controller to redirect *back* to the new product page. Done.

Doing Something After a Form Submit

But I want to do something different: after a successful form submit, I want to *close* the modal. How can we do something *after* a turbo frame navigates?

Tip

Starting in Turbo 7.0 RC2, there *are* two frame-specific events: `turbo:frame-render` and `turbo:frame-load`.

We already know that Turbo triggers a bunch of events... but there aren't any events *specific* to turbo frames. There's no, `turbo:frame-start` or anything like that. *However*, Turbo *does* trigger an event right before and after a form submits.

In `modal-form_controller.js`, add a `connect()` method. Until now, we've listened to all of our turbo events inside of `assets/turbo/turbo-helper.js`. The reason is that all of this code represents *global* behaviors: stuff that we we're adding to the *entire* page.

turbo:submit-end in Stimulus

But in this case, we want to listen to an event *only* when a specific controller is active... so we can run some custom code that affect *just* that controller. Say `this.element.addEventListener()` and listen to an event called `turbo:submit-end`. Pass this an arrow function with an `event` argument.

Earlier we listened to `turbo:submit-start`. As you can see, there is *also* a `turbo:submit-end` event, which happens after the submit Ajax call has finished. Let's `console.log(event)` to see what it looks like.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 3
export default class extends Controller {
// ... lines 5 - 7
    connect() {
        this.element.addEventListener('turbo:submit-end', (event) => {
            console.log(event);
        });
    }
// ... lines 13 - 17
}
```

Oh, and you probably noticed one big difference between this event and the other events that we've listened to. Most Turbo events are dispatched directly on `document`. But the form events - like `turbo:submit-start` and `turbo:submit-end` - are actually dispatched on the `form` element. Then, they bubble up.

This means that you *can* attach a listener to `document`... or *any* element that *contains* the form, including the form itself. By attaching the event listener to `this.element`, our callback will only be executed when a form is submitted *inside* of this: so inside of the modal. That's... pretty awesome.

Ok, let's see what this event looks like. Move over, refresh the page, open the modal and submit. Go check the console. There it is! Like other events, this has a `detail` key with a `formSubmission` inside. Oh, but there's also a `success` key set to `false`! That would be true if this was a *successful* form submit. *That's* handy: we can use it to know if the submit was successful and then close the modal.

Let's go do it! If `event.detail.success`, then `this.modal.hide()`.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 3
export default class extends Controller {
// ... lines 5 - 7
    connect() {
        this.element.addEventListener('turbo:submit-end', (event) => {
            console.log(event);
            if (event.detail.success) {
                this.modal.hide();
            }
        });
    }
// ... lines 16 - 20
}
```

Cool. Refresh, open the modal, fill in some details and submit. Go team!

Next: even though we closed the modal, the frame system *still* followed the redirect and updated the HTML in the modal. In this case, that's not a problem. In other cases, it could cause an error. Let's find out when and dive even deeper into the event system to fix it.

Chapter 35: Prevent a turbo-frame from Rendering

As usual, I'm going to complicate things! But I have a good reason: I really want us to get the most out of frames... *and* we have a bug hiding.

Head over to `ProductAdminController`. As we just talked about, this redirects to the `product_admin_index` page. Let's pretend that we want to redirect this to the "reviews" page for the new product. Change this to `app_product_reviews` and pass the `id` wildcard set to the new id: `$product->getId()`.

src/Controller/ProductAdminController.php

```
// ... lines 1 - 31
    /**
     * @Route("/new", name="product_admin_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    {
// ... lines 37 - 51
            return $this->redirectToRoute('app_product_reviews', [
                'id' => $product->getId(),
            ]);
        }
// ... lines 56 - 107
```

Cool. But this change won't affect our modal. When the modal submit is successful, we're simply closing the modal, staying on the page and completely *ignoring* the frame that lives in the now-closed modal. This new redirect would only affect us if we went directly to the `/new` admin page where the form targets the full page.

So, since this won't affect us, it shouldn't break anything! Famous last words. Refresh, open the modal, add some details and submit. Oh! The modal *did* close... but we have an error in the console!

> *“Response has no matching `<turbo-frame id="product-info">` element.”*

Ah, the problem is that, even though we closed the modal, the `turbo-frame` *still* followed the redirect to the product review page. Then, like it *always* does, it looked for a `<turbo-frame>`

with `id="product-info"`... which that page doesn't have.

So what we *really* want to do is just... close the modal and tell turbo to *not* follow the redirect. Unfortunately, the `turbo:submit-end` event is too late to tell Turbo to do that!

We could ignore this error... or hack an empty turbo-frame onto the reviews page... but let's fix this properly. It's a good challenge.

Order of Turbo Events

When we submit this form, *four* events are triggered in this order: `turbo:before-fetch-request`, `turbo:submit-start`, `turbo:before-fetch-response` and finally `turbo:submit-end`. *Then* the frame is rendered.

But, wait a second. If the frame isn't rendered until *after* `turbo:submit-end`, why is it too late to tell Turbo to *not* render the frame? The truth is that `turbo:submit-end` isn't *actually* too late. The *real* problem is that Turbo doesn't give us a way to *cancel* rendering from this event. But it *does* give us this power from the event right *before* this: `turbo:before-fetch-response`.

turbo:before-fetch-response

This event is triggered right *after* the Ajax call finishes, actually after *both* Ajax calls have finished: the form submit POST *and* the second request to the redirected page. But at this point, the frame has *not* been re-rendered.

Tip

Starting in Turbo 7 RC4 (and so also in the stable Turbo 7), the `turbo:before-fetch-response` event *is* now triggered from whatever element triggered the Ajax call. This means that you can now use `this.element.addEventListener` instead of attaching it to `document`. Nice!

This time, we *do* need to attach the event to `document` because this event is dispatched directly there - not on the form. For now, I'm going to *not* hide the modal.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 3
export default class extends Controller {
// ... lines 5 - 7
    connect() {
        document.addEventListener('turbo:before-fetch-response', (event)
=> {
            console.log(event);
            if (event.detail.success) {
                //this.modal.hide();
            }
        });
    }
// ... lines 16 - 20
}
```

Refresh, open the modal and fill out the form so we can see what the event looks like for a successful form submit. Cool. In the console, we see *two* of these events. The first happened when we opened the modal: that's the GET request to load the form. The second is from the form submit.

Open this up and look at the `detail` property: it has a `fetchResponse` object and inside of it that... awesome! A `succeeded` key and a `redirected` key! So it tells us if the request was successful and *also* if it was redirected.

So here's the plan: when this event happens, *if* a modal is open *and* the Ajax call was successful *and* the Ajax call was a redirect, we'll *assume* that a form was just submitted and hide the modal.

Back in the listener function, delete the code. Then, if *not* `this.modal` - so if the modal has never opened - or if *not* `this.modal._isShown` - an internal way to detect whether a modal is visible - then we don't need to do anything. Just `return`.

But if the modal *is* open, set `const fetchResponse` to `event.detail.fetchResponse`: that's the object we were just looking at. If `fetchResponse.succeeded` *and* `fetchResponse.redirected`, then we're going to assume this was a successful form submit and hide the modal.

Cancelling the Frame Render

If we stopped now, this would do the *exact* same thing as before... just with more code. It would hide the modal... but then the frame would *still* try to render and give us that annoying error. But there's a key difference between this event and `turbo:submit-end`: *this* event is *cancellable*. In this event we're allowed to say `event.preventDefault()`.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 3
export default class extends Controller {
// ... lines 5 - 7
    connect() {
        document.addEventListener('turbo:before-fetch-response', (event)
=> {
            console.log(event);
            if (!this.modal || !this.modal._isShown) {
                return;
            }

            const fetchResponse = event.detail.fetchResponse;
            if (fetchResponse.succeeded && fetchResponse.redirected) {
                event.preventDefault();
                this.modal.hide();
            }
        });
    }
// ... lines 22 - 26
}
```

Normally, we use `event.preventDefault()` to prevent form submits or link clicks. Some custom events - like this one - *also* allow you to call this method... and it could mean *anything* based on the event. In this case, it communicates to Turbo that we would like to *prevent* this response from rendering.

Let's try it. Refresh, open, fill out the form and submit. Yes! The modal closed... *this* time with *no* error!

We're amazing! Oh, except... hmm... this still isn't quite what we want. The modal closed... but the page didn't reload or refresh... so we don't see the new product in the list immediately. Let's fix that next and finish our Turbo-powered modal system.

Chapter 36: Full Page Redirect from a Frame

Our Turbo-frame-powered modal is now *almost* perfect. When we submit successfully, it closes the modal. But... dang! That's *all* it did. The product list did *not* update... so it's not *super* obvious that this worked!

Look at the console log of the event for the successful form submit. Let's see. Inside `response`, ooh! We can see what URL the frame was redirected to! You can also get this from `fetchResponse`: this `fetchResponse.location` is an object that points to the final, redirected URL.

So the reason we're looking at this is that what we *really* want to do is, after the form submits successfully, read this URL and navigate the *entire* page to it with Turbo! We want a frame that's, sort of a "hybrid". We want the form submit to *stay* in the frame... but then once the submit is successful, we want to navigate the whole page to the redirected URL as if we were *not* in a frame.

Navigating the Redirect with Turbo

And... yea! We can do that! At the top of the controller, import Turbo: `import * as Turbo from '@hotwired/turbo'`.

Below, remove the `console.log`, then `Turbo.visit(fetchResponse.location)`.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 2
import * as Turbo from '@hotwired/turbo';

export default class extends Controller {
// ... lines 6 - 8
    connect() {
        document.addEventListener('turbo:before-fetch-response', (event)
=> {
            if (!this.modal || !this.modal._isShown) {
                return;
            }

            const fetchResponse = event.detail.fetchResponse;
            if (fetchResponse.succeeded && fetchResponse.redirected) {
                event.preventDefault();
                Turbo.visit(fetchResponse.location);
                this.modal.hide();
            }
        });
    }
// ... lines 23 - 27
}
```

Let's do this! Refresh, open the modal, typy, typy, submit and... cool! The whole page navigated to the reviews page! Oh, and back in our code, we can remove `this.modal.hide()`. We don't need that anymore: we're navigating the entire page, so that will naturally replace the modal.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 4
export default class extends Controller {
// ... lines 6 - 8
    connect() {
        document.addEventListener('turbo:before-fetch-response', (event) => {
            if (!this.modal || !this.modal._isShown) {
                return;
            }

            const fetchResponse = event.detail.fetchResponse;
            if (fetchResponse.succeeded && fetchResponse.redirected) {
                event.preventDefault();
                Turbo.visit(fetchResponse.location);
            }
        });
    }

// ... lines 23 - 27
```

"Binding" this for a Listener Method

I'm pretty happy with this, but let's clean things up a bit. Copy the code inside the arrow function, scroll down, and create a new method called `beforeFetchResponse()` with an `event` argument. I'm doing this for readability.

In `connect()`, call that. We don't even need an arrow function: just reference `this.beforeFetchResponse`.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 4
export default class extends Controller {
// ... lines 6 - 8
    connect() {
        document.addEventListener('turbo:before-fetch-response',
this.beforeFetchResponse);
    }
// ... lines 12 - 17
    beforeFetchResponse(event) {
        if (!this.modal || !this.modal._isShown) {
            return;
        }

        const fetchResponse = event.detail.fetchResponse;
        if (fetchResponse.succeeded && fetchResponse.redirected) {
            event.preventDefault();
            Turbo.visit(fetchResponse.location);
        }
    }
}
```

There *is* a problem with this... but let's try it! Refresh, go back to the admin page, open up the modal and fill this out with real data. Submit!

It didn't redirect! And we have that error back in the console. What happened? It's not super obvious at first, but in our new method, the `this` variable is no longer referencing the `controller` object. This is the classic problem with callback functions, and we normally work around "this" by passing an arrow function. But if you *do* want to point directly to the method, you *can* by *binding* the method.

Check it out: say `this.boundBeforeFetchResponse` - I'm actually creating a new property `= this.beforeFetchResponse.bind(this)`. Then, below, point to the bound method.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 4
export default class extends Controller {
// ... lines 6 - 8
    connect() {
        this.boundBeforeFetchResponse =
this.beforeFetchResponse.bind(this);
        document.addEventListener('turbo:before-fetch-response',
this.boundBeforeFetchResponse);
    }
// ... lines 13 - 31
```

This creates a new property that points to the method.... but where we have *guaranteed* that the `this` variable in that method will point to `this` object. That's the job of `bind`. And this isn't a Stimulus problem, it's a problem you run into whenever you combine JavaScript, callbacks and objects.

It looks weird at first... but when we submit the form... it *does* solve our issue: back to the good behavior!

Disconnecting the Event Listener

Oh, but I do want to handle one small detail. Over in the controller, add a `disconnect()` method. Then copy the `document.addEventListener()` line, paste, and change it to `document.removeEventListener()`.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 4
export default class extends Controller {
// ... lines 6 - 13
    disconnect() {
        document.removeEventListener('turbo:before-fetch-response', this.boundBeforeFetchResponse);
    }
// ... lines 17 - 33
}
```

Why are we doing this? If we add an event listener to a controller's element, like `this.element`, then if that element is removed from the page, it's no big deal that our listener is still technically attached to it. Nothing can interact or trigger events on that element anymore. And your browser will probably garbage collect that element - and the listener - anyways.

But if we add an event listener to the `document`, then *every* time a new `data-controller="modal-form"` appears on the page, our connect method will be called and we'll attach yet *another* listener. Even after a controller's element is removed from the page, its `beforeFetchResponse()` would *still* be called!

So, to be the responsible developers that we are, we *remove* the listener in `disconnect()`, which is called when the element attached to this controller is removed from the page.

Changing the Redirect Back to the List Page

Anyways, to put the cherry on top of our new feature, head back to `ProductAdminController`. Change the redirect *back* to `product_admin_index`, which just makes more sense.

```
src/Controller/ProductAdminController.php
// ... lines 1 - 31
    /**
     * @Route("/new", name="product_admin_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    {
// ... lines 37 - 42
        if ($form->isSubmitted() && $form->isValid()) {
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($product);
            $entityManager->flush();

            if ($request->isXmlHttpRequest()) {
                return new Response(null, 204);
            }

            return $this->redirectToRoute('product_admin_index');
        }
// ... lines 54 - 61
    }
// ... lines 63 - 105
```

Time to try the entire process. Go to the admin area and do a full refresh. Click to open the modal - that loaded via the frame - hit save - that submitted via the frame - and if fill in some real data. This is going to submit - like normal - *to* the frame. Then, we'll detect that it was *successful* and... boom! The new product shows up! That's because we just *navigated* to this page with Turbo. That's smooth.

Next: we just did something pretty custom. We submitted a form *into* a turbo frame... but then navigated the entire *page* on success. This is not something a turbo frame does natively... but it's kind of handy. So let's add a reusable way to do this whenever we want.

Chapter 37: Redirecting the Full Page from a Frame

We just did something pretty custom. Normally, if you submit a form into a frame, if that frame redirects, the new content will be loaded *into* the frame only. The URL in the address bar won't change and the rest of the page won't be affected. That's usually what you want!

But sometimes, we *do* want to navigate the entire page, like in a modal. Or, imagine that you have a sidebar with a form. When you submit and fail validation, you *do* want that to show in the sidebar. But once the form is successful, you want to navigate the *entire* window to a confirmation page.

So let's make our frame-redirecting system something that we can use *anywhere*. Here's the plan: if a `turbo-frame` - like the `turbo-frame` in `_modal.html.twig` - has a `data-turbo-form-redirect="true"` attribute - which I *totally* just invented - then we will redirect the whole page if we detect a redirect in that frame.

templates/_modal.html.twig

```
<div
    class="modal fade"
    tabindex="-1"
    aria-hidden="true"
    data-modal-form-target="modal"
>
    <div class="modal-dialog">
        <div class="modal-content">
// ... lines 9 - 14
            <turbo-frame
                class="modal-body"
                src="{{ modalSrc }}"
                id="{{ id }}"
                loading="lazy"
                data-turbo-form-redirect="true"
            >
                {{ modalContent|default('Loading...') }}
            </turbo-frame>
        </div>
    </div>
</div>
```

Moving Code to turbo-helper

Because this new redirect behavior will be something that will work *anywhere* on our site, we need to move the logic *out* of our `modal-form` controller and into `turbo-helper` where the rest of our global Turbo stuff lives.

Copy the `beforeFetchResponse()` method and delete it. Then, in `turbo-helper`, paste this at the bottom. Cool.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 103
    beforeFetchResponse(event) {
        if (!this.modal || !this.modal._isShown) {
            return;
        }

        const fetchResponse = event.detail.fetchResponse;
        if (fetchResponse.succeeded && fetchResponse.redirected) {
            event.preventDefault();
            Turbo.visit(fetchResponse.location);
        }
    }
}
// ... lines 116 - 118
```

Back in `modal-form_controller`, we don't need the `disconnect()` method anymore. We're going to register this listener just *once* inside of `turbo-helper`. Copy part of `connect()`, delete the rest... and we can also remove the Turbo import.

```
assets/controllers/modal-form_controller.js
// ... lines 1 - 3
export default class extends Controller {
    static targets = ['modal'];
    modal = null;

    async openModal(event) {
        this.modal = new Modal(this.modalTarget);
        this.modal.show();
    }
}
```

Over in `turbo-helper`, go up to the constructor - here it is - and paste. To call the method, pass an arrow function with an event argument and call `this.beforeFetchResponse(event)`.

```
assets/turbo/turbo-helper.js
```

```
// ... lines 1 - 3
const TurboHelper = class {
    constructor() {
// ... lines 6 - 16

        document.addEventListener('turbo:before-fetch-response', (event)
=> {
            this.beforeFetchResponse(event);
        });

// ... line 22
    }
// ... lines 24 - 114
}
// ... lines 116 - 118
```

Finding the "Active" Frame, if any, for a Request

Ok - go back down to that method. This is *not* going to work yet... because it's still coded to work with a modal. To bring this to life, we need determine three things. One: was the Ajax call redirected? Two: did this navigation happen *inside* of a Turbo frame? And three: does that frame have the `data-turbo-form-redirect` attribute?

Tip

Starting in Turbo 7 RC4 (and so also in the stable Turbo 7), the `turbo:before-fetch-response` event *is* now passed which element the Ajax call was triggered on, as `event.target`. You could use this to find the "current turbo-frame" via `event.target.closest('turbo-frame')`.

The trickiest of these three is actually figuring out if this Ajax call is happening *inside* of a turbo frame. This event doesn't give us any indication of what *initiated* the Ajax call - like which link was clicked or which form was submitted. But, we can use a trick. Remember: whenever a frame is loading, turbo gives that frame a `busy` attribute. We can use that.

Create a new convenience method called `getCurrentFrame()`. This is going to return the `turbo-frame` Element that is currently loading or null. And it's as simple as return `document.querySelector()` looking for `turbo-frame[busy]`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 115
    getCurrentFrame() {
        return document.querySelector('turbo-frame[busy]');
    }
}
// ... lines 120 - 122
```

It *is* theoretically possible that *two* frames could be loading at the same time. But other than on initial page load if you had multiple lazy frames, I think that's pretty unlikely.

Above, let's use this. Remove all of this modal stuff... and then move the `event.preventDefault()` and `Turbo.visit()` to the end of the method... because we're going to *reverse* the `if` logic to keep things clean. If the `fetchResponse` did *not* succeed or it's *not* a redirect, then return and do nothing.

But if the response *was* successful and was a redirect, we need to see if we are inside of a frame *and* make sure that the frame has our data attribute. If not `this.getCurrentFrame()`, then return and do nothing. And if the current frame does *not* have `.dataset.turboFormRedirect`, *also* do nothing.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 103
    beforeFetchResponse(event) {
        const fetchResponse = event.detail.fetchResponse;
        if (!fetchResponse.succeeded || !fetchResponse.redirected) {
            return;
        }

        if (!this.getCurrentFrame() ||
!this.getCurrentFrame().dataset.turboFormRedirect) {
            return;
        }

        event.preventDefault();
        Turbo.visit(fetchResponse.location);
    }
// ... lines 117 - 120
}
// ... lines 122 - 124
```

At this point, we know that the Ajax call *did* happen inside of a frame with our `data` attribute *and* that the Ajax call *did* redirect to another page. And so, we prevent the frame from rendering and navigate the entire page.

Let's try it! Refresh, open the modal, fill in some info, submit and... got it! I *know* that worked because the new product showed up thanks to the Turbo visit.

Yay! But... was that too easy? It... kind of was. There are two annoying bugs that are hiding inside of our new system. Let's add one more turbo frame next that will expose both of them. Don't worry, by the end, we're going to have a beautiful bug-free way to force a frame to navigate the whole page.

Chapter 38: Frame Redirecting and Clearing the Snapshot Cache

Where else could we use our new turbo frame redirect system? Go to the cart. On the featured product sidebar, we could leverage a frame around the cart controls. Right now, this form submits to the whole page. And so, on success, the entire page is redirected to that product with a nice flash message. I love it! That's exactly the behavior I want.

But go back to the cart. This time, let's change the color... and be annoying: try to buy a *negative* quantity. Hit add. It *still* changed the entire page... which isn't as smooth as I'd like. It would be *way* cooler if the error showed up in the sidebar on the cart page.

Adding the turbo-frame

Time to add a frame! The template for this "add to cart" section lives at `templates/product/_cart_add_controls.html.twig`. This template is included on two pages: the product show page and also over on the sidebar. When we submit the form, as we just saw, it's handled by the product show page. This means that if we added a frame around this entire template, when we submit, the response *will* contain a matching `turbo-frame`... since the product show page renders this template.

In other words... adding a frame here should... just work. On top, add `<turbo-frame>` with `id="add-to-cart-controls"`. Add the closing frame at the bottom.

```
templates/product/_cart_add_controls.html.twig
<turbo-frame id="add-to-cart-controls">
{{ form_start(addToCartForm, {
    attr: { class: 'cart-add-controls d-flex align-items-center justify-content-baseline' }
}) }}
    {% if addToCartForm.color is defined %}
// ... lines 6 - 25
    {% endif %}
// ... lines 27 - 34
{{ form_end(addToCartForm) }}

<div>
    {{ form_errors(addToCartForm) }}
    {% if addToCartForm.color is defined %}
        {{ form_errors(addToCartForm.color) }}
    {% endif %}
    {{ form_errors(addToCartForm.quantity) }}
</div>
</turbo-frame>
```

Just with that, refresh the page and go to the cart. Submit with a negative quantity. That is *so* much nicer. Now change to red change, set the quantity to 5 and hit add.

Um, did that work? The color changed back and the quantity reset... and I don't see any errors. But that wasn't very obvious. I also don't see the item in the cart until I refresh.

As usual, this behavior makes sense if you think about it. When we submit the form, it redirects to the product show page. And *that* renders a success message - which we don't see - and a "reset" add to cart form. Ya know, this would all work much better if we could go *back* to the original success behavior where we navigate the *entire* page after adding an item.

Activating data-turbo-form-redirect

Fortunately, that's exactly what our new frame system does! Let's add the attribute that we invented to this frame. I'll move it onto multiple lines to keep my sanity, then add `data-turbo-form-redirect="true"`.

```
templates/product/_cart_add_controls.html.twig
<turbo-frame
    id="add-to-cart-controls"
    data-turbo-form-redirect="true"
>
// ... lines 5 - 46
</turbo-frame>
```

Testing time! Refresh the cart page. If we submit the form with errors, everything stays right here. But if we submit it *successfully*... yes! That redirected to the product show page!

We Preventing the Snapshot Cache From Clearing!

Though... dang! There are two weird things going on. First, we're missing our flash message! We'll talk about that later.

To see the second, watch the shopping cart header as we add more and more items. Yikes! It jumps backwards and forwards!

This is a result of the preview system. When we submit this form, for *just* a moment, it shows the cached preview of this page. For example, at this moment, the cached version - which comes from the last time we navigated *away* from this page - still holds the value 14. So when we hit add, it jumps back to 14 and ahead to 16. Now, a page with 15 sits in the cache.

This is *not* normally a problem. If you submit a POST request with Turbo and the response is successful, Turbo *automatically* clears its internal snapshot cache. It does that *precisely* to avoid this problem: a successful form submit typically means that something has *changed* on the server. So, to be safe, Turbo decides that it shouldn't use any of its old, cached pages.

Manually Clearing the Cache

So... if that's true, why are we seeing this problem? In `turbo-helper`, we're calling `event.preventDefault()` in the `turbo:before-fetch-response` listener. This tells Turbo to prevent rendering this response. But... it has a side effect: it *also* prevents it from clearing its snapshot cache!

But now that we know that, it's no problem: we can clear it manually by saying `Turbo.clearCache()`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 103
    beforeFetchResponse(event) {
// ... lines 105 - 113
        event.preventDefault();
        Turbo.clearCache();
        Turbo.visit(fetchResponse.location);
    }
// ... lines 118 - 121
}
// ... lines 123 - 125
```

Refresh and watch the cart header. *Much* better.

Bug with Not Clearing the Current Page

By the way, there *is* still one spot where this jumpy cart thingy happens. Go to the cart page and add an item. Watch the number when I click back to the shopping cart... it's 21 right now. See that? It temporarily jumped back to 20.

This happens due to, what I think is, a fairly straightforward bug in Turbo that I hope will be fixed in the future. Here's the scoop. As we just talked about, when you successfully submit a POST form, Turbo clears its snapshot cache. And we even manually did that a minute ago. But right *after* it clears the snapshot cache, as we're navigating away, it *re-caches* the page that we just submitted!

This means that, when we hit add, it clears the snapshot cache but then re-caches this page with a shopping cart number set to 21.

This is pretty rare situation. To trigger this, you need a form that submits to *another* page. And then the problem only happens if you navigate *back* to that form later. I'm going to ignore this.

Next: the *bigger* weird issue with our new system is that, when we add an item, it redirects... but there's no success flash message. This page actually *does* have a flash message... we saw it a few minutes ago: it should be showing right here. But something unexpected is happening behind the scenes that's hiding it.

Let's find out what next and improve our system to prevent it.

Chapter 39: Manual "Restore" Visit

Refresh, go to the cart page and add another item from the sidebar. A few minutes ago, after doing this, we saw a nice green success flash message on the top of the page. Where did it go?

Look at the network tools and scroll up. Ah, here's the problem. When we submitted the add to cart form into the frame, our controller redirected and the turbo frame *followed* that redirect. This request is the POST to `/cart`... and this is the Ajax request for the redirect. That response *does* contain a success flash message: "Item added!".

But remember: flash messages are only rendered *one* time. Or, to be more precise, as soon as we render a flash message, Symfony *removes* it so that it's never rendered again.

The problem is that... we never actually *see* this response on the page. Nope. We detect that this redirect happened, cancel the render - which only would have rendered inside the frame anyways - and then use Turbo to navigate to this URL. That's the *second* identical request. Unfortunately, once we get there, the flash message is gone... because it was already rendered... even though *we* never saw it.

Yep, our system works great except that the redirected page is requested twice... and we only render the second one.

Ajax Calls and Redirects: A Conundrum

This is actually tough to fix... and it's mostly not Turbo's fault. We *could* try to work around this by adding some code to our flash logic. Like, *if* the request is for a turbo frame, don't render the flash message. That way, it won't get used and will render on the next *full* request.

But... that feels hacky to me. The *real* solution is harder, but more correct: avoid the second, duplicate request!

Internally Turbo uses the `fetch()` function to make its Ajax calls. When we return a redirect, `fetch` automatically follows that and makes a second Ajax request, which we see down here. So, this "follow the redirect" behavior does *not* come from Turbo... it's just how `fetch` works.

The ideal solution would be for `fetch()` to... *not* follow the redirect: to make only the *first* request, stop, then tell us the redirect URL so that *we* can visit it with Turbo.

Unfortunately... that's literally *not* possible. For complex reasons that might change someday, you *can* tell `fetch()` to *not* follow a redirect. But if you do, `fetch()` purposely *hides* the URL that it *would* have redirected to... which means we have no idea what URL to make Turbo navigate to! Yup, our ideal solution is entirely *not* possible in browsers as of today. What a mess!

Fortunately, there are still two ways to solve this correctly, and I'll show you both. The first is quick, easy and... involves using an internal option in Turbo that the documentation specifically tells you *not* to use. Exciting! The second solution involves some work in our Symfony app, but avoids using that option.

Upgrading Turbo... Again

So let's start with the pure Turbo solution. It's beautifully simple and... it all starts with a question: if the turbo-frame already makes the Ajax request to the redirected page, could we simply tell Turbo to navigate to that page and use *that* HTML... without making a second request? Think about it: over in `turbo-helper.js`, this `fetchResponse` already contains the HTML we want! We just need Turbo to put that onto the page and update the address bar.

Doing this *is* possible... mostly. Start by finding your terminal and, once again, running:

```
yarn upgrade @hotwired/turbo
```

The Internal "restore" Option

This upgrades Turbo to RC-1. Turbo seems to always release a new feature just *before* I need it. In this case, it's a `PageSnapshot` class we'll use later.

Now, over in `turbo-helper.js`, add a second argument to `Turbo.visit()` - an options argument. One option here is called `action`.... and one of the values you can set it to is `restore`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 103
    beforeFetchResponse(event) {
// ... lines 105 - 115

            Turbo.visit(fetchResponse.location, {
                action: 'restore'
            });
    }
// ... lines 121 - 124
}
// ... lines 126 - 128
```

The action `restore` tells Turbo to visit this URL, but with the same behavior as if you clicked the back or forward buttons in your browser. Specifically, if the page is already in the snapshot cache, use that snapshot and make *no* network request. If it's not already in the snapshot cache, then it *will* make a network request.

This is the part where we're breaking the rules. "Restoration visits" are reserved for clicking the back and forward buttons. Setting this action to `restore` *will* work... but the documentation says that this is "internal" and that we should *not* use this `action` directly.

But... let's ignore that for now. Refresh the page, head back to the cart and add another item. Hmm, we *still* don't see the flash message. Oh, that's because even though Turbo *has* made a request for this URL - via the redirect - that response was never put into the snapshot cache. Remember: a snapshot of a page is normally taken the moment you navigate *away* from that page. We're going to need to put the HTML into the snapshot cache manually.

Here's how... and some of this *is* pretty deep in Turbo. Say `const snapshot = Turbo.PageSnapshot.fromHTMLString()` and pass it the response HTML, which we can get by saying `fetchResponse.responseHTML`. Except... `responseHTML` returns a Promise... so we need to `await` that. And as soon as we await *that*, we need to make the method `async`.

This gives us a Snapshot object from that HTML. To put this into the cache, say `Turbo.navigator.view.snapshotCache.put()` and pass this the URL - or "location" - of the page - `fetchResponse.location` - and then the `snapshot` object.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 103
    async beforeFetchResponse(event) {
// ... lines 105 - 113
        event.preventDefault();
        Turbo.clearCache();
        const snapshot = Turbo.PageSnapshot.fromHTMLString(await
fetchResponse.responseHTML);
        Turbo.navigator.view.snapshotCache.put(fetchResponse.location,
snapshot)

        Turbo.visit(fetchResponse.location, {
            action: 'restore'
        });
    }
// ... lines 123 - 126
}
// ... lines 128 - 130
```

This is... pretty low-level, but *that* is how you can manually add a page to the cache. Let's try it!

Do the whole flow again: refresh the page, go to the cart, submit, and... we got it! The flash message shows up and, down in the network tools, we see only one request for this page. That's awesome!

Is this Internal Option Safe?

So... maybe we just stick with this solution and hope it won't break in the future. Even though the action `restore` is meant as an internal flag, I couldn't find any conversation about *why* it's internal or what risks there are: the note in the documentation was added years ago when the feature was first introduced.

But... if you want to play it safe, we have another solution. Change this back to a normal visit... and also take off the `async`.

Next: let's solve this problem again by doing some fancy communication between Turbo and Symfony.

Chapter 40: Adding a Custom Request Header Based on the Frame

Okay, so if we don't want to cheat and use the internal `restore` action with a Turbo visit, how *else* can we solve our problem? Well, there's really only one option. Let me reopen my network tools. Right now, when we successfully submit into a `<turbo-frame>`, like this modal, the frame follows the redirect, meaning it makes a request to the redirected URL. *Then* we navigate to that same URL, which causes a *second* request to it. Somehow, we need to avoid having these *two* requests.

So if we can't force Turbo to directly use the response from this first Ajax call, because we don't want to use the internal `restore` action, then our only choice is to somehow *prevent* that first Ajax call from happening at all. But since the JavaScript `fetch()` function *always* follows redirects, the only real way to do this is to make Symfony *not* return a real redirect after a successful form submit.

So here's the idea... it's kind of crazy. In Symfony, we're going to *detect* if a request is being sent via a `turbo-frame` *and* if that frame has the `data-turbo-form-redirect` attribute. If both of these are true and *if* the `Response` from the controller is a redirect, we will change the `Response` to... *not* be a redirect! We'll return a normal 200 status code but store the URL that we *want* to redirect to as header on the response. Then, we'll prevent Turbo from rendering that response, like we already are, read the URL from the header, navigate with Turbo and voilà! We redirect the page *without* the duplicate request.

Sending data-turbo-form-redirect to the Server

So where do we start? Turbo already adds a `Turbo-Frame` header to any Ajax request that happen inside a frame. We can see this, for example, down on the POST request. All the way near the bottom... there it is: `turbo-frame: product-info`. We can read that in Symfony.

But what we *can't* yet read in Symfony is whether or not this frame has the `data-turbo-form-redirect` attribute. To make that possible, let's hook into Turbo and add that information as a *new* request header.

In `turbo-helper.js`, we need to listen to another event. Head up to the `constructor()`... and say `document.`. Actually, cheat. Steal the event listener code from below... and change the event to `turbo:before-fetch-request`.

Remember: Turbo dispatches this event right *before* it makes *any* Ajax request. Inside, call a new method - `this.beforeFetchRequest()` - and pass the `event`.

Copy that method name, head down to the methods... and add that with the `event` argument. Inside, `console.log(event)` so we can see what it looks like.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 3
const TurboHelper = class {
    constructor() {
// ... lines 6 - 17
        document.addEventListener('turbo:before-fetch-request', (event) =>
{
            this.beforeFetchRequest(event);
        });
// ... lines 21 - 26
    }
// ... lines 28 - 106

    beforeFetchRequest(event) {
        console.log(event);
    }
// ... lines 111 - 130
}
// ... lines 132 - 134
```

Back at our browser, refresh. This logs *every* time Turbo makes an Ajax request, like when we navigate... or a frame loads. This is from the weather frame. And I think if we go down to the bottom... yep! It fires again when the *second* weather frame loads.

Head over to the cart page, clear the console, then add an item to the cart. Ooh, the event triggered *three* times. One was for the submit, one for the navigation to the next page and the last was for the weather widget that loaded on this page.

Detecting if the Frame Request has data-turbo-form-redirect

Check out the first log, which is from the POST request when we submit the form into the frame. Ah, `event.detail` has a `fetchOptions` key! This is the collection of options that are *about* to be passed to the `fetch()` function. And it has a `headers` key with `Turbo-Frame` inside.

That's no surprise... but we can use that in JavaScript to figure out if this frame has the special `data-turbo-form-redirect` attribute.

Check it out: say `const frameId =` and *read* that header: `event.detail.fetchOptions.headers`... and we're looking `Turbo-Frame`. We need to use square brackets instead of `.` because the key has a dash in it.

Now, if there is *not* a `frameId`, then this request is *not* happening inside a frame. In that case, do nothing.

But if we *do* have a `frameId`, we can *use* that to find this element: `const frame = document.querySelector()`... and then use ticks so we can look for `#` then `${frameId}`.

Yep, we're literally finding that `<turbo-frame>` element on the page! If we can't find the frame for some reason - which shouldn't happen - *or* if the frame does *not* have the dataset of `turboFormRedirect`, then do nothing. Whoops - make sure that's `turboFormRedirect`.

Go back to the cart page and inspect element on the frame. As a reminder, this *does* have the `data-turbo-form-redirect="true"` attribute. That's what we're looking for.

At this point, we know that the request *is* happening in a frame and that the frame *does* have the `data-turbo-form-redirect` attribute. And so, we're going to add a new header. Use `event.detail.fetchOptions.headers` again to invent a new header called, how about, `Turbo-Frame-Redirect`. Set it to `1`.

```
assets/turbo/turbo-helper.js
// ... lines 1 - 107
    beforeFetchRequest(event) {
        const frameId = event.detail.fetchOptions.headers['Turbo-Frame'];
        if (!frameId) {
            return;
        }

        const frame = document.querySelector(`#${frameId}`);

        if (!frame || !frame.dataset.turboFormRedirect) {
            return;
        }

        event.detail.fetchOptions.headers['Turbo-Frame-Redirect'] = 1;
    }
// ... lines 122 - 145
```

Cool! Let's go check it! At your browser, any normal request - even a request inside a frame like for the weather widget - will *not* have the new header. Check the weather frame request. All the way down... yep! It *does* have a `turbo-frame` header... but *not* `turbo-frame-redirect`.

But now go back to the cart and clear the requests. Submit the form... scroll up to that request... and scroll down. There it is! `turbo-frame-redirect`! We can now *detect* - from Symfony - when a request is going through this type of a frame. Oh yes, we're dangerous.

Next, let's turn to the Symfony side of things where we'll use this header to magically transform redirect responses into something that we can better handle in JavaScript.

Chapter 41: Smart Frame Redirecting with the Server

When an Ajax request happens via a `<turbo-frame>` *and* that frame has our `data-turbo-form-redirect` attribute, we're now communicating that to Symfony by sending a new header on the request called `Turbo-Frame-Redirect`. We're now going to *use* that to *change* any redirect responses to, sort of, "fake redirects" so that the `fetch()` function in JavaScript doesn't automatically follow them.

Creating the Event Subscriber

We're going to add this magic with an event subscriber. In the `src/` directory, let's create a new `EventSubscriber/` directory... and inside, a new PHP class called, how about, `TurboFrameRedirectSubscriber`. Make this implement `EventSubscriberInterface`... and then go to the "Code -> Generate" menu - or "Command + N" a Mac - and select "Implement Methods" to generate the *one* method we need: `getSubscribedEvents()`. Inside, return one event - `ResponseEvent::class` - set to `onKernelResponse`.

`ResponseEvent` is one of the *last* events that happens during the request-response process. It happens *after* our controller has been called... so the `Response` object *has* already been created.

Above this, add the `public function onKernelResponse()` method with a `ResponseEvent $event` argument.

```
src/EventSubscriber/TurboFrameRedirectSubscriber.php
<?php

namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\ResponseEvent;

class TurboFrameRedirectSubscriber implements EventSubscriberInterface
{
    public function onKernelResponse(ResponseEvent $event)
    {
    }

    public static function getSubscribedEvents()
    {
        return [
            ResponseEvent::class => 'onKernelResponse',
        ];
    }
}
```

Cool. So the logic inside of here will be fairly simple: if the request has the `Turbo-Frame-Redirect` header *and* the response is a redirect, then we're going to *change* the response to something else.

Replacing the Response

To keep things organized, add a new private method called `shouldWrapRedirect()`. This will need the `Request` object - so we can read the header - and the `Response` object that the controller created. This will return a `bool`.

Before we work on that method, back in `onKernelResponse()`, call this: if *not* `$this->shouldWrapRedirect()`... passing `$event->getRequest()` and `$event->getResponse()`. If we should *not* wrap the redirect, return and do nothing.

In a minute we'll add the logic down here to change the response.

But let's finish `shouldWrapRedirect()`. Start by checking to see if the `$response` is *not* a redirection. If it's not, return false. The only responses we need to change are *redirects*: we don't want to change normal frame loads or frame form submits that are returning with validation errors.

```
src/EventSubscriber/TurboFrameRedirectSubscriber.php
// ... lines 1 - 9
class TurboFrameRedirectSubscriber implements EventSubscriberInterface
{
    public function onKernelResponse(ResponseEvent $event)
    {
        if (!$this->shouldWrapRedirect($event->getRequest(), $event->getResponse())) {
            return;
        }
    }
// ... lines 18 - 25
    private function shouldWrapRedirect(Request $request, Response $response): bool
    {
        if (!$response->isRedirection()) {
            return false;
        }

        if (!$request->headers->has('Turbo-Frame')) {
            return false;
        }

        if ($request->headers->get('Turbo-Frame-Redirect')) {
            return true;
        }

        return false;
    }
}
```

The only other check we need is for the header. Copy the header name from `turbo-helper.js`. Then `return $request->headers->get('Turbo-Frame-Redirect')`. So if the header exists and is set to something "truthy" like 1, this method will return true. Else, it will return false. Actually, I'm missing a tiny detail, but I'll fix it in a minute.

Finally, back in `onKernelResponse()`, at this point, we know that this request was made inside of a frame that has our `data-turbo-form-redirect` attribute *and* we know that the controller returned a redirect.

And so, create a *new* response object: new `Response()`, passing `null` for the content - we don't need to return anything - a `200` status code - so *not* a redirect - and then an array of headers. Invent a new header called `Turbo-Location` set to the URL that we *want* to redirect to. We can get that from the original response: `$event->getResponse()->headers->get('Location')`.

Finally, to use this response instead of the original, say `$event->setResponse($response)`.

```
src/EventSubscriber/TurboFrameRedirectSubscriber.php
// ... lines 1 - 9
class TurboFrameRedirectSubscriber implements EventSubscriberInterface
{
    public function onKernelResponse(ResponseEvent $event)
    {
        if (!$this->shouldWrapRedirect($event->getRequest(), $event->getResponse())) {
            return;
        }

        $response = new Response(null, 200, [
            'Turbo-Location' => $event->getResponse()->headers->get('Location'),
        ]);
        $event->setResponse($response);
    }
// ... lines 23 - 30
    private function shouldWrapRedirect(Request $request, Response $response): bool
    {
        if (!$response->isRedirection()) {
            return false;
        }

        return (bool) $request->headers->get('Turbo-Frame-Redirect');
    }
}
```

Ok! That's all we need to do in Symfony: we're now replacing the redirect response in this situation with something different.

Reading the Response Header and Navigating

The last little piece of work is back in JavaScript. We already have a `beforeFetchResponse()` method, which is currently looking to see if a request was successful and redirected... and checking for the `turboFormRedirect` data attribute.

We can simplify this a *lot*. All we need to do *now* is check to see if the response has this `Turbo-Location` header. If it does, then we know that we should read that header and navigate.

Remove most of the code on top and add `const redirectLocation =` set to `fetchResponse.response.headers.get('Turbo-Location')`.

Then, if we do *not* have a `redirectLocation`, we know this is not a situation where we need to do anything fancy. So, just return.

Then, the rest is perfect, except instead of `fetchResponse.location`. use `redirectLocation`.

assets/turbo/turbo-helper.js

```
// ... lines 1 - 3
const TurboHelper = class {
// ... lines 5 - 122
    beforeFetchResponse(event) {
        const fetchResponse = event.detail.fetchResponse;
        const redirectLocation =
fetchResponse.response.headers.get('Turbo-Location');
        if (!redirectLocation) {
            return;
        }

        event.preventDefault();
        Turbo.clearCache();
        Turbo.visit(redirectLocation);
    }
}
// ... lines 135 - 137
```

That's it. We don't even need our `getCurrentFrame()` method anymore. It took more work inside of Symfony, but the JavaScript side of things is nice!

Oh, but before we try this, back in our subscriber, before the return statement, add a `(bool)` type-cast. This will guarantee the method returns a boolean.

```
src/EventSubscriber/TurboFrameRedirectSubscriber.php
// ... lines 1 - 9
class TurboFrameRedirectSubscriber implements EventSubscriberInterface
{
// ... lines 12 - 30
    private function shouldWrapRedirect(Request $request, Response $response): bool
    {
        if (!$response->isRedirection()) {
            return false;
        }

        return (bool) $request->headers->get('Turbo-Frame-Redirect');
    }
}
```

Ok, *now* let's try it: Go back to the cart page and refresh. Remember: the whole goal is to be able to submit this form and have it *not* make duplicate requests to the redirected page. If we accomplish that, we'll be rewarded by seeing the success flash message. And... yes! There it is!

Look up here on the Ajax requests. We submitted the cart form here... and then there was only *one* request for the product show page, not two. Mission accomplished!

Thanks to our new fancy system, we can also - easily - solve an annoying problem. What happens if the user tries to open something in a `<turbo-frame>` - like a modal - but they got logged out in the background... maybe after taking a really long coffee break. Instead of just having this load broken, let's write about 10 lines of code to gracefully handle this everywhere.

Chapter 42: Automatically Redirect Ajax Calls to /login

All sites that loads things via Ajax have one annoying problem: what happens if the user gets logged out due to inactivity? Obviously if the user gets logged out and clicks a link to navigate the *whole* page, that's no problem. They'll get redirected to the login page.

But go to a product page and scroll down to the review section. Pretend that I stop right here, go home for the day, eat a delicious dinner, watch Mystery Science Theater 3000 and come back to my computer tomorrow. During that time, my session has timed out. What would happen if I tried to submit this form - which submits into a `turbo-frame` - without refreshing first?

Well... let's try it! I'm going to imitate this situation by opening the site in a new tab... and logging out. Back over in the first tab, clear the network requests and submit. Uh, that was weird.

In the network tools, you can see that it *did* submit to the reviews page. But then, because I'm not logged in, it redirected to the login page. In the console, we see our favorite error:

> *“response has no matching `<turbo-frame id="product-review">` element.”*

That makes sense! The Ajax request redirected to the login page. And so, the frame system *followed* that redirect and then looked for a `product-review` `<turbo-frame>` *on* that page... which it obviously doesn't have.

So the user experience here is... not so great. But for any frames that have our `data-turbo-form-redirect` attribute, this problem is already fixed thanks to the system we just built!

Check it out. Refresh... log back in and head to the admin section. Remember: this modal *does* have that attribute on it. So I'm going to repeat our experiment. In the other tab, refresh, then log out. Back on the first tab, when we open the modal, the `<turbo-frame>` will *try* to make a request to a page that requires authentication. When we try it... awesome! It redirected the entire page to `/login`! That's perfect!

Wrapping the Redirect Response to /login

So this problem is fixed in some places... but not everywhere. But we *can* make this work everywhere.

In `TurboFrameRedirectSubscriber`, look at `shouldWrapRedirect()`. Let's think: if this response is a redirect to the *login* page *and* if the request is happening inside a `<turbo-frame>`, then we *definitely* know that we want to wrap the redirect so that our JavaScript redirects the whole page.

Start by checking to see if *not* `$request->headers->get('Turbo-Frame')`. In this case, return `false`. Adding this check was redundant before... because if you have the `Turbo-Frame-Redirect` header then you *definitely* have this one. But now it's going to help us detect if we're in a frame *and* if the response is redirecting to the login page.

Grab the redirect location by saying `$location = $response->headers->get('Location')`. Instead of checking to see if this equals `/login`, let's be fancier and use the URL generator.

At the top of the class, add a `__construct()` function with a `UrlGeneratorInterface` argument... which is just a more hipster way to get the router service. I'll hit Alt + Enter and go to "Initialize properties" to create that property and set it.

Back down in the method, if `$location` is equal to `$this->urlGenerator->generate()`, passing this the name of our login route - `app_login` - then return `true`.

That's it! If the response is a redirect... and the request is happening inside of a frame... and we're redirecting to the login page... then that's a problem. That's going to break the frame. And so, we'll wrap the redirect with our fake redirect so that our JavaScript can navigate things.

```
src/EventSubscriber/TurboFrameRedirectSubscriber.php
// ... lines 1 - 10
class TurboFrameRedirectSubscriber implements EventSubscriberInterface
{
    private UrlGeneratorInterface $urlGenerator;

    public function __construct(UrlGeneratorInterface $urlGenerator)
    {
        $this->urlGenerator = $urlGenerator;
    }

// ... lines 20 - 37

    private function shouldWrapRedirect(Request $request, Response
$response): bool
    {
        if (!$response->isRedirection()) {
            return false;
        }

        $location = $response->headers->get('Location');

        if ($location === $this->urlGenerator->generate('app_login')) {
            return true;
        }

        return (bool) $request->headers->get('Turbo-Frame-Redirect');
    }
}
```

Testing time! Log back in... go back to a product page, scroll down to the reviews, and then, in the other tab, refresh and log out.

Back in tab number 1, try to submit the review form. Beautiful! We are smoothly redirected to the login page! This problem just got solved for *any* `<turbo-frame>` on our site.

Okay team! Enough with turbo frames! It's time to dive into part 3 of Turbo: Turbo Streams. This feature is probably the smallest of the three, but also *the* most fun to work with.

Chapter 43: Turbo Streams

The third and final part of Turbo is Turbo Streams. These are fun!

Hello Streams

Turbo Streams are a way to return instructions on updating *any* element on the page. And there are two main use cases. First: you're submitting a form inside a frame and, on success, you want to update an element that lives *outside* of that frame. Second: you need a way to update something on your page asynchronously but *without* "polling" - where you make an Ajax call every few seconds to *constantly* check for updates. For example, if you were building a chat app where you want a new message to render as *soon* as the other person sends it, Turbo Streams can help.

Streams are another way to enhance your page. So they're an extra feature - like frames - that you can choose to add whenever you want to do something special.

For example, go to a product page and scroll down. See this review form? It lives in a frame and it works awesome. The frame surrounds both the form *and* the list of reviews above it... we can see that if we inspect the element.

This means that, when we submit, both the form *and* the review list updates. That gives us a fresh form *and* we see the new review. Awesome!

But scroll up to the product details where we show the number of reviews and the average review. These details did *not* update when we submitted the review. Watch: if I refresh the page, the 8 reviews... becomes 9.

This area lives *outside* of the `product-review` turbo frame. So we *can't* update it via the frame. But we *can* update it using Turbo Streams... because... that's their whole purpose! To update *any* element on the page from the server.

Creating & Returning our First Stream

Here's how it works. Step one: find the area on the page that you want to update and give it a unique ID. The template for this page lives in `templates/product/show.html.twig`. Let's see... here are the details. On the `<div>` around this, add, how about, `id="product-quick-stats"`.

```
templates/product/show.html.twig
{% extends 'product/productBase.html.twig' %}

{% block productBody %}
    <turbo-frame id="product-info" target="_top" class="row pt-3 product-show">
// ... lines 5 - 31
            <div class="p-3 mt-4 d-flex justify-content-between flex-wrap flex-lg-nowrap">
                <div id="product-quick-stats">
                    <strong>{{ product.priceString|format_currency('USD') }}</strong>
                    <br>
                    <strong>{{ product.reviews|length }}</strong> Reviews
                    <br/>
                    <strong>{{ product.averageStars }}/5</strong><i class="fas fa-star ms-2"></i>
                </div>
                <div>
                    {{ include('product/_cart_add_controls.html.twig') }}
                </div>
            </div>
// ... line 44
    </turbo-frame>
// ... lines 46 - 48
    <h3>Reviews</h3>

    {{ include('product/_reviews.html.twig') }}
{% endblock %}
```

Now open `Controller/ProductController.php` and find the reviews action. This is the page that we submit to when we post a new review. Down here, instead of redirecting on success, let's do something different, let's render a new template.

I'll leave the old logic for now. But above this, return `$this->render()` to render a template called `product/reviews.stream.html.twig`. We don't need to pass any variables yet, but I'm going to pass an empty second argument because we *do* need to pass a third argument: a `new TurboStreamResponse()`.

Tip

In `symfony/ux-turbo` 2.1 and higher, this code has changed. The `TurboStreamResponse` is no longer needed, but a `setRequestFormat()` call is:

```
$request->setRequestFormat(TurboBundle::STREAM_FORMAT);

return $this->render('product/reviews.stream.html.twig');
```

src/Controller/ProductController.php

```
// ... lines 1 - 18
class ProductController extends AbstractController
{
// ... lines 21 - 69
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager)
    {
// ... lines 75 - 85
            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
// ... lines 87 - 89
                return $this->render('product/reviews.stream.html.twig', [
                ], new TurboStreamResponse());
// ... lines 92 - 97
            }
        }
// ... lines 100 - 106
    }
// ... lines 108 - 115
}
```

Okay first: see the `.stream` in the template name? Yep. That has *no* technical effect. It's just a naming convention because this template will have a special format. Second, by passing a `TurboStreamResponse` as the third argument, we're telling Symfony to render the template like normal, but to put the HTML into *this* response object instead of a normal response object. I'll show you what that does in a minute.

Alright: let's go create the template. In `product/`, create the file: `reviews.stream.html.twig`. These stream templates contain HTML, but... in a special format that describes the element on the page that you want to change, *how* you want to change it and the HTML to use.

It always starts with a `<turbo-stream>` element. This needs two attributes, the first is `action=""` set to, in this case, `update`. We'll talk more about this in a minute. The second is `target=""` set to the id of the element on the page that should be updated. I'll copy `product-quick-stats` and paste that here.

Inside of the `<turbo-stream>`, we always have a `<template>` element. This... doesn't really mean anything... you just always need it. Inside of *that*, put the HTML. Start by hardcoding something.

```
templates/product/reviews.stream.html.twig
<turbo-stream action="update" target="product-quick-stats">
    <template>
        Will this <strong>really</strong> work???
    </template>
</turbo-stream>
```

Ok, let's see this in action! Find your browser, refresh and scroll down. Add a review and... submit!

Hmm. Nothing happened? It looks like the form is kind of stuck submitting. But scroll up to the quick stats area. Woh! There's our new HTML!

How a Turbo-Stream Works Under the Hood

This is a turbo stream in action. Check out the network tools and find the POST Ajax request for the form submit - this one on the bottom. As expected, when we submit the form, it now returns this special `<turbo-stream>` HTML. But check out the headers on the response. There it is: the response has a `Content-Type` header set to `text/vnd.turbo-stream.html`. That's important.

Here's the whole flow of what just happened. In our controller, we render the `reviews.stream.html.twig` template and put it into a special `TurboStreamResponse`. That response object causes a special `Content-Type` header to be set on the response: `text/vnd.turbo-stream.html`.

That's important because, as soon as we set up Turbo on our site, like the first thing we did at the *very* beginning of this tutorial, turbo added an event listener to the `turbo:before-fetch-response` event. In `turbo-helper.js`, we have our *own* listener for this event, which is dispatched after *any* Ajax call that Turbo makes has finished.

Anyways, the moment you install Turbo, it adds a listener to this event that looks at the response for *every* Ajax call and checks to see if the `Content-Type` starts with `text/vnd.turbo-stream.html`. If it *does*, instead of handling the response normally - like rendering it into the `turbo-frame` - the response is passed to the Turbo Stream system... which reads this and updates the `product-quick-stats` element.

But... that's *all* it did. The reviews frame, down here, did *not* update. We'll talk about that in a minute.

Other Stream "Actions"

In addition to the `update` action, there are a bunch of other actions that you can use to update the page. In the Turbo docs, go to the Reference section and select Streams. So you can `append` an element to the end of an existing element, `prepend`, `replace` an entire element, `update` the HTML *inside* an element, which is what we're doing, `remove` an element or even place an element `before` or `after` another element.

You can also target using a CSS selector - like a class name - instead of an `id`.

Next: let's improve our stream so that it updates the quick stats area with the *real* new information. And after submitting a new review, we *still* need the reviews area - the form and list - to update. We can *also* handle that inside the same stream.

Chapter 44: Streams: Reusing Templates

When we submit the product review form, instead of redirecting like we were before, we're now returning this `TurboStreamResponse`. When the Ajax call finishes, Turbo *notices* that we're returning this type of response... instead of a real HTML page. And so, instead of handling the HTML like a frame *normally* would, it passes it to the Turbo Stream system.

Right now, we're using it to update the quick stats area of the page with this random HTML. If you refresh, the *real* goal is to update the review count and review average as soon as the new review is submitted.

Reusing Templates in a Stream

To do that, without repeating ourselves, over in `show.html.twig` - the template for the product show page - copy the quick stats code... and create a new template in `templates/product/` called, how about `_quickStats.html.twig`. Paste the code here.

templates/product/_quickStats.html.twig

```
<strong>{{ product.priceString|format_currency('USD') }}</strong>
<br>
<strong>{{ product.reviews|length }}</strong> Reviews
<br/>
<strong>{{ product.averageStars }}/5</strong><i class="fas fa-star ms-2">
</i>
```

We can now *reuse* this in two places. In `show.html.twig` include it: `product/_quickStats.html.twig`

```
templates/product/show.html.twig
{% extends 'product/productBase.html.twig' %}

{% block productBody %}
    <turbo-frame id="product-info" target="_top" class="row pt-3 product-show">
// ... lines 5 - 31
                <div class="p-3 mt-4 d-flex justify-content-between flex-wrap flex-lg-nowrap">
                    <div id="product-quick-stats">
                        {{ include('product/_quickStats.html.twig') }}
                    </div>
                    <div>
                        {{ include('product/_cart_add_controls.html.twig') }}
                    </div>
                </div>
// ... line 40
    </turbo-frame>
// ... lines 42 - 46
    {{ include('product/_reviews.html.twig') }}
{% endblock %}
```

Then, in the *stream* template, do the same thing!

```
templates/product/reviews.stream.html.twig
<turbo-stream action="update" target="product-quick-stats">
    <template>
        {{ include('product/_quickStats.html.twig') }}
    </template>
</turbo-stream>
```

Pretty cool, right?

Let's try that. Refresh. *This* still works and shows 10 reviews. Scroll down and add review number 11. Submit and... oh! The entire reviews section is gone! My web debug toolbar is angry: that Ajax call returned a 500 error.

Open up its profiler.

> *"Variable product does not exist"*

Coming from `_quickStats.html.twig`. Of course, the problem is that we're including `_quickStats.html.twig` from `reviews.stream.html.twig`. In `ProductController`, we're not passing any variables to that template... but in quick stats, we *need* a `product`!

No problem: pass product set to `$product`, which will make it available all the way into the quick stats template.

```
src/Controller/ProductController.php
// ... lines 1 - 18
class ProductController extends AbstractController
{
// ... lines 21 - 69
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager)
    {
// ... lines 75 - 85
            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
// ... lines 87 - 89
                return $this->render('product/reviews.stream.html.twig', [
                    'product' => $product,
                ], new TurboStreamResponse());
// ... lines 93 - 98
            }
// ... lines 100 - 107
    }
// ... lines 109 - 116
}
```

Okay: take two. Refresh again. We now have *11* reviews... so let's go add number 12. Submit. The reviews section is still weird - but that's ok. Scroll up. Yes! Our Turbo Stream updated the area with the real data! That is *so* cool!

Return a Stream or HTML, not both (yet)

Now we need to fix the reviews area... because showing a filled-in form with a disabled button... doesn't exactly scream "the review was successfully saved".

The entire reviews area lives in `templates/product/_reviews.html.twig`... and *all* of it is surrounded by the `product-review` frame. So both the list of reviews *and* the form are in this frame.

Thanks to this, *before* we started messing around with turbo streams, after submit, we redirected to the reviews page. That page includes this template with this frame. And so, *the* entire frame updated, including the *new* review and a fresh, empty form.

At this point, we have two choices. First, we could redirect on success like we were doing before and let the normal turbo frame logic do its magic. *Or* we can return a turbo stream and update whatever elements we want. *But*, we can't do *both*. Our controller can only return *one* thing, so we need to choose between returning a redirect *or* returning a stream.

Well, actually we *can* do both... but let's keep that a secret between you and I for now. It's a topic for later... and requires one extra piece of technology.

So... what can we do? Well, if we want to be able to update the quick stats area *and* the reviews area, the answer is to return a stream that contains *multiple* instructions. Let's see how next.

Chapter 45: Multiple Updates in one Stream

Our goal is to be able to update the quick stats area *and* the reviews area all at once. We can't do that by redirecting or returning a normal HTML page... because that would only affect the reviews frame. So let's *continue* to return a stream... but a stream where we update the quick stats area *and* the reviews.

The entire content of `_reviews.html.twig` lives inside of an element with a `product-review` id. So, in `reviews.stream.html.twig`, add a *second* `<turbo-stream>`. Yup, we can include as *many* instructions as we want in a stream. Set the `action=""` to `replace` and the `target` to `product-review`, the id of the element that surrounds the reviews area. Inside, include the reviews template. Oh, but don't forget to include the `<template>` element - I'll remember that in a minute.

templates/product/reviews.stream.html.twig

```
<turbo-stream action="update" target="product-quick-stats">
    <template>
        {{ include('product/_quickStats.html.twig') }}
    </template>
</turbo-stream>

<turbo-stream action="replace" target="product-review">
    <template>
        {{ include('product/_reviews.html.twig') }}
    </template>
</turbo-stream>
```

We're using `replace` instead of `update` because `_reviews.html.twig` *contains* the target. So we want to *replace* the existing `product-review` element with the *new* one... instead of just updating its `innerHTML`.

Before we try this, I'll go back to `reviews.stream.html.twig` and add the `<template>` element. If you *do* forget this, you'll get a clear error that says that a `template` element was expected.

Ok: move over and refresh. Let's add another glowing review... and submit. Yes! It worked! I see my new review! But... the form is gone.

Adding a Success Message

As *so* often happens... this makes total sense. *Before*, the frame was being redirected to the reviews page. So it was being redirected to this page here... and *this* page contains a *fresh* form. So, naturally, the fresh form showed up at the bottom of the reviews frame after successfully submitting a review.

But now, over in `reviews.stream.html.twig`, when we render `_reviews.html.twig`, if you look at that template, we are *not* passing in a `reviewForm` variable. And I already have logic here that checks to see if that variable exists and conditionally renders the form. So, in our case, it renders nothing.

We *could* create a `reviewForm` object in the controller and pass it into here. But, I kind of like this... except that having a success message would help a lot.

So let's see: we check for `reviewForm` and we also check to see if the user is *not* logged in. Add an else on the bottom with a success alert. In our situation, the only way to get here is if the form *was* just submitted successfully. But you could also pass a `success` variable to the template to be more explicit.

templates/product/_reviews.html.twig

```
<turbo-frame id="product-review">
// ... lines 2 - 18
{% if reviewForm|default(false) %}
// ... lines 20 - 28
{% elseif not is_granted('ROLE_USER') %}
    <p><a href="{{ path('app_login') }}" data-turbo-frame="_top">Log in</a> to post your review</p>
{% else %}
    <div class="alert alert-success">
        Thanks for your "real" review you "human" ?!
    </div>
{% endif %}
</turbo-frame>
```

Anyways, let's test this thing out with *another* glowing review. When we submit... that's *lovely*.

A Link to Reload the Form

I'm having too much fun so here's a challenge. Imagine we want to add a link below this success message to "Add another review". When we click it, it should load a fresh form right into the frame. How could we do that?

Well... that's almost disappointingly easy! Remember: we're inside of a `turbo-frame`... so all we need to do is add a link in the frame that navigates us to the review page... because the review page *renders* this frame *with* a fresh form!

Check it out: right after the success message, add an anchor tag with `{{ path() }}` to generate a URL to the `app_product_reviews` route. This needs an `id` wildcard set to `product.id`. Put some text inside.

templates/product/_reviews.html.twig

```
<turbo-frame id="product-review">
// ... lines 2 - 18
{% if reviewForm|default(false) %}
// ... lines 20 - 28
{% elseif not is_granted('ROLE_USER') %}
    <p><a href="{{ path('app_login') }}" data-turbo-frame="_top">Log
in</a> to post your review</p>
{% else %}
    <div class="alert alert-success">
        Thanks for your "real" review you "human" ?!
    </div>

    <a href="{{ path('app_product_reviews', {
        id: product.id
    }) }}">Love the product *that* much? Add another review!</a>
{% endif %}
</turbo-frame>
```

Move back over, refresh... and, once again, profess your love - or maybe disgust - for this product: your call. Submit. There's our success message. When we click this normal link... yes! That was awesome! Go team streams and frames!

Checking for the Stream "Accept" Request Header

Finally, there's *one* last detail I want to handle... and it's minor. Imagine if, for some reason, this review form were submitted without JavaScript. And so it performs a normal full page submit, not a submit through Turbo.

Until now, that was *totally* okay! Our controller saves the new review and then redirected to a legitimate page. But now we're returning this bizarre stream HTML... which our browser wouldn't know what to do with... it would probably just render it onto the page... which is not great!

Fortunately, whenever Turbo makes an Ajax request, it adds an `Accept` header to the request that *advertises* that it supports Turbo streams. We can check for that in the controller.

Here's how it looks: wrap our stream render with if `TurboStreamResponse::STREAM_FORMAT` equals `$request->getPreferredFormat()`.

Tip

In `symfony/ux-turbo` 2.1 and higher, this code has changed:

```
if (TurboBundle::STREAM_FORMAT === $request->getPreferredFormat()) {
    $request->setRequestFormat(TurboBundle::STREAM_FORMAT);

    return $this->render('product/reviews.stream.html.twig', [
        'product' => $product,
    ]);
}
```

```
src/Controller/ProductController.php
// ... lines 1 - 18
class ProductController extends AbstractController
// ... lines 20 - 69
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager)
    {
// ... lines 75 - 80
        if ($request->isMethod('POST')) {
// ... lines 82 - 85
            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
// ... lines 87 - 89
                if (TurboStreamResponse::STREAM_FORMAT === $request->getPreferredFormat()) {
                    return $this->render('product/reviews.stream.html.twig', [
                        'product' => $product,
                    ], new TurboStreamResponse());
                }

                $this->addFlash('review_success', 'Thanks for your review! I like you!');

                return $this->redirectToRoute('app_product_reviews', [
                    'id' => $product->getId(),
                ]);
            }
        }
// ... lines 103 - 109
    }
// ... lines 111 - 120
```

That's it. This preferred format thing basically looks at the `Accept` request header to see if the request supports turbo streams. All Ajax requests made through Turbo *send* this header.

If the request *does* support streams, then... we return a stream! If it doesn't, we do our normal behavior: redirect the page. So once again, this will work fine without JavaScript. Also, even though I've not done *any* work with it yet, Turbo can also be used to build Native iOS or Android apps: you can read about it in their docs. Streams don't really make sense in that context, so coding like this also makes sure your code supports native apps... if you ever choose to go in that direction.

Next: let's have some fun with Turbo Streams! I want to see if we can create and process them manually in JavaScript. Apart from being cool, this will give us a better understanding of how streams work and a better appreciation for the next big part of streams that we'll discuss after.

Chapter 46: Processing Streams by Hand for Fun & Profit

As we've learned, each time Turbo makes an Ajax call, it listens to see if the response has a content type of `text/vnd.turbo-stream.html`. If it *does*, then the HTML is passed to the Turbo Stream system... and it works its magic. But in theory, you could grab some Turbo Stream HTML from *anywhere* and tell the Stream system to process it. And... it's kind of fun!

Head to the homepage. This counter area is fueled by a Stimulus controller: the one at `assets/controllers/counter_controller.js`. It's pretty simple: click it and then it increments a variable and updates the text with the new number. In addition to doing this, I want to *invent* a Turbo Stream that adds a flash message to the top of the page.

Adding the Stream Target to the Page

First, we need to be able to target the flash area so that we can put stuff into it. In `templates/base.html.twig`, find the section - I'll search for flash - and surround it: a `<div>` with id set to, how about, `flash-container`. Pop the closing tag on the other side.

```
templates/base.html.twig
<!DOCTYPE html>
<html lang="en-US">
    <head>
// ... lines 4 - 13
    </head>
    <body>
// ... lines 16 - 69
            <div id="flash-container">
            {% for flash in app.session.flashBag.get('success') %}
                <div class="alert alert-success alert-dismissible fade show" role="alert">
                    {{ flash }}
                    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close">
                      <span aria-hidden="true"></span>
                    </button>
                </div>

            {% endfor %}
            </div>
// ... lines 81 - 100
    </body>
</html>
```

Manually Creating a Stream

Back in `counter_controller.js`, right after we update the count on the page, let's add a new variable: `const streamMessage` set to some ticks so we can easily create a multi-line string. Inside, we're literally going to invent a new `<turbo-stream>` with `action="update"` and `target="flash-container"`. Add the `template` element and, inside of that, an alert success div:

> *“thanks for clicking `${this.count}` times.”*

This variable is a plain, boring string... but a string that has the `<turbo-stream>` format.

So... could we tell the Turbo Stream system to read this and follow its instructions? And if so, how?

At the top of this file, we're already importing the `visit` function from `@hotwired/turbo`. This library exports a *bunch* of other things, including a function called `renderStreamMessage`.

Copy that. Down below, it's as simple as this: `renderStreamMessage()` passing `streamMessage`.

```
assets/controllers/counter_controller.js
// ... line 1
import { visit, renderStreamMessage } from '@hotwired/turbo';
// ... line 3
export default class extends Controller {
// ... lines 5 - 7
    increment() {
        this.count++;
        this.countTarget.innerText = this.count;

        const streamMessage = `
<turbo-stream action="update" target="flash-container">
    <template>
        <div class="alert alert-success">
            Thanks for clicking ${this.count} times!
        </div>
    </template>
</turbo-stream>
        `;
        renderStreamMessage(streamMessage);

        if (this.count === 10) {
            visit('/you-won');
        }
    }
}
```

Done! Let's try this thing. Head back over, refresh and click. Oh! That's *so* cool. We now have a dead-simple way to mutate different elements on your page from JavaScript. And more importantly, this shows off the fact that stream handling is a standalone system inside of Turbo. And so, in theory, we could get this stream HTML from *any* source, not just from an Ajax call. That will be important in the next chapter.

Removing an Element from the Snapshot Cache

Go back to the page and click this 10 times. Woo! The Stimulus controller navigates us to the winning page! Click back to the homepage... but watch closely. Did you see the flash message?

It was there for *just* a moment and then disappeared.

That is *totally* unrelated to streams. This would happen with *any* flash message in Turbo Drive, thanks to its preview system. But even though this has nothing to do with streams, while we're here, let's fix it... *and* learn something new along the way!

One solution to this would be to go into `assets/turbo/turbo-helper.js` and *remove* any flash messages before the snapshot is taken. We already have logic for that: we listen to `turbo:before-cache` and clean up several elements.

But starting in Turbo 7 Beta 8, there's a *new* attribute that you can add to any HTML element that you do *not* want to include in your snapshot. If you think about it, we *never* want a flash message to be in a snapshot: we want the flash message to show once... but *not* be there if we navigate away and then back again.

So, in `base.html.twig`, it's really simple: on `flash-container` - which will contain all flash messages - add a new `data-turbo-cache="false"`.

templates/base.html.twig

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
// ... lines 4 - 13
    </head>
    <body>
// ... lines 16 - 69
            <div id="flash-container" data-turbo-cache="false">
            {% for flash in app.session.flashBag.get('success') %}
                <div class="alert alert-success alert-dismissible fade show" role="alert">
                    {{ flash }}
                    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close">
                        <span aria-hidden="true"></span>
                    </button>
                </div>

            {% endfor %}
            </div>
// ... lines 81 - 100
    </body>
</html>
```

That's it! Thanks to this, the entire element - and anything inside - will *not* be included in the snapshot. Check it out: refresh the homepage... click 10 times and go back. Beautiful! No flash message.

Next: we know we can return Turbo Streams as the response from a controller. That's what we've been doing so far in the reviews action. But there's also *another* - more *powerful* - way to send streams to your users.

Chapter 47: Mercure: Pushing Stream Updates Async

Turbo streams would be *much* more interesting if we could *subscribe* to something that could send us stream updates in real time.

The Use-Case: Pushing Streams Directly to Users

Like, imagine we're viewing a page... and generally minding our own business. At the *same* moment, someone else - on the other side of the world - adds a new review to this same product. What if that review instantly popped onto *our* page *and* the quick stats updated? That would be... incredible!

Or imagine if, in `ProductController`, inside of the reviews action, after a successful form submit, we could *still* return a redirect like we were doing before... but we could *also* push a stream to the user that updates some *other* parts of the page, like the quick stats area. I said earlier that returning a redirect *and* a stream isn't possible. But... that's not entirely true.

The truthiest truth is that both of these scenarios are *totally* possible. How? Turbo Streams comes with built-in support to listen to a web socket the returns Turbo Stream HTML. It also supports doing that same thing with server-sent events, which are kind of a modern web socket: it's a way for a web server to *push* information to a browser without *us* needing to make an Ajax call to ask for it.

Hello Mercure!

And fortunately, in the Symfony world, we have *great* support for a technology that enables server-sent events: Mercure. Mercure could... probably be its own tutorial, so we'll just cover the basics.

Mercure is a "service" that you run, kind of like your database service, Elasticsearch or Redis. It allows, in JavaScript for example, to *subscribe* to messages. Then, in PHP, we can *publish* messages to Mercure. Anything that has *subscribed* will *instantly* receive those messages and can do something with them. If you're familiar with WebSockets, it has a similar feel.

Installing the Mercure Libraries

We're going to get Mercure rocking... and it's going to *really* make things fun. To start, install a package that makes it easy to work with Mercure and Turbo. At the command line, run:

```
composer require "symfony/ux-turbo-mercure:^1.3"
```

Tip

The `symfony/ux-turbo-mercure` is deprecated in favor of `symfony/ux-turbo` which already contains the cool Mercure stuff. Just install `symfony/mercure-bundle` to get it working:

```
composer require symfony/mercure-bundle
```

Or to get the version used in the tutorial, continue with:

```
composer require "symfony/ux-turbo-mercure:^1.3"
```

This installs several things. First, a PHP library called mercure that helps *talk* to the Mercure service in PHP. Second, a MercureBundle that makes that *even* easier in Symfony. And third, a `symfony/ux-turbo-mercure` library that gives us a special Stimulus controller that helps Mercure and Turbo Streams work together. Go team!

This executed a recipe... so run `git status` to see what it did.

```
git status
```

Ok cool. Let's look at `.env` first. At the bottom, we have three new environment variables that will help us talk to Mercure. More about these in a few minutes. The recipe also modified `controllers.json`. Remember: this means that a new Stimulus controller is now available that lives inside this bundle. We'll use that 2 chapters from now.

Tip

Instead of a new section in this file, find the existing `` `@symfony/ux-turbo ``` section. It will have a key called ```mercure-turbo-stream``. Change its ``enabled`` key to ``true` to activate the Stimulus controller we'll be using.

This also enabled a bundle... and added a new library to our `package.json` file. We've seen this several times before with UX packages: this adds a new package to our project... but instead of downloading the code, it already lives in the `vendor/` directory.

To get that part properly set up, near the bottom of the terminal output, it tells us to stop Encore and run `yarn install --force`.

In the other tab, hit Ctrl+C to stop Encore and run:

```
yarn install --force
```

When that finishes, restart Encore:

```
yarn watch
```

Ok, we just installed some PHP and JavaScript code that's going to help us communicate with Mercure. But... we don't actually have a Mercure service running yet! That's like installing Doctrine... but without MySQL or Postgresql running!

So next, let's get the Mercure service running. There are a bunch of ways to do this. But if you're using the Symfony binary web server like we are... then... it's already done!

Chapter 48: Running the Mercure Service in the symfony Binary

Mercure itself is a "service" or "server" - kind of like MySQL or Elasticsearch. The Mercure server is called the "hub"... and there are several good ways to get it running. First, they have a managed version where they handle it all for you. This is great for production: it keeps things simple *and* you can help support the project.

Or, you can download Mercure and set it up locally. *Or* you can set up Mercure with Docker - that's totally supported. Or the *final* or is... *if* you're using the Symfony binary as your local web server then... well... it's already running!

The Embedded Mercure Hub

Head to your open terminal tab, clear the screen and run:

```
symfony server:status
```

As a reminder, *way* back at the start of this tutorial, we used the Symfony binary to run a local web server for us. Back at the browser, open a new tab and go to https://127.0.0.1:8000 - the URL to our site - then `/.well-known/mercure`.

Tip

The latest `symfony` binary no longer embeds Mercure. But it's still easy to set up. First, add a `mercure` service to your `docker-compose.yaml` file:

docker-compose.yaml
```yaml
// ... line 1
services:
// ... lines 3 - 13
    mercure:
        image: dunglas/mercure
        command: caddy run -config /etc/caddy/Caddyfile.dev
        ports: ['80']
        environment:
            SERVER_NAME: ':80'
            MERCURE_PUBLISHER_JWT_KEY: '!ChangeMe!'
            MERCURE_SUBSCRIBER_JWT_KEY: '!ChangeMe!'
```

You can also copy the code block from the script below the video. Start the container by running:

```
docker-compose up -d
```

That's it! But instead of being accessible at the URL you see in the tutorial, the Mercure hub will be exposed on a random port. To find it, run:

```
symfony var:export --multiline
```

And look for the `MERCURE_URL` value - it should equal something similar to `http://127.0.0.1:64150/.well-known/mercure`. Put *this* into your address bar to see your Mercure Hub (you'll see the same error as in the video).

If everything is working... yes! You should see this error:

> *“Missing "topic" parameter.”*

This *is* a Mercure hub. Yup, the Symfony binary comes with Mercure *already* running at this URL. We get that for free.

The Environment Variables

To communicate with this, head back over to your editor and open the `.env` file.

```
.env
```

```
// ... lines 1 - 29
###> symfony/mercure-bundle ###
# See https://symfony.com/doc/current/mercure.html#configuration
# The URL of the Mercure hub, used by the app to publish updates (can be a local URL)
MERCURE_URL=https://127.0.0.1:8000/.well-known/mercure
# The public URL of the Mercure hub, used by the browser to connect
MERCURE_PUBLIC_URL=https://127.0.0.1:8000/.well-known/mercure
# The secret used to sign the JWTs
MERCURE_JWT_SECRET="!ChangeMe!"
###
```

These three environment variables define values that are used in a new config file: `config/packages/mercure.yaml`. `MERCURE_PUBLIC_URL` is the *public* URL to the Mercure hub that our JavaScript will use to *subscribe* to messages and `MERCURE_URL` is the URL that our PHP code will use to *publish* messages. These are usually the same. `MERCURE_SECRET` is basically a password that will allow us to publish: more on that later.

```
config/packages/mercure.yaml
```

```yaml
mercure:
    hubs:
        default:
            url: '%env(MERCURE_URL)%'
            public_url: '%env(MERCURE_PUBLIC_URL)%'
            jwt:
                secret: '%env(MERCURE_JWT_SECRET)%'
                publish: '*'
```

In our case, both URL variables *already*, by chance, point to the correct URL! Yay! But actually, if you're using the latest version of the Symfony binary... we don't even *need* these variables in this file! Why? Well, in addition to setting up Mercure for us, the Symfony binary *also* sets these environment variables automatically to their correct values.

Check it out. Back over in our editor, open `public/index.php`. Let me close a few things... then open it. Cool. Right after the runtime load, I'll paste in some code.

public/index.php

```php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

dd(array_filter($_SERVER, function($item) {
    return str_contains($item, 'MERCURE');
}, ARRAY_FILTER_USE_KEY));

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

This looks fancy, but I'm basically dumping the `$_SERVER` variable... except only the keys that contain `MERCURE`. The `$_SERVER` variable - among other things - will contain all environment variables. I'm filtering for `MERCURE` basically... because I don't want to accidentally publish any secret keys from my computer to the internet... as much fun as that would be.

Anyways, this will run *before* the `.env` file is loaded, so it will only print *real* environment variables. Back over on our site, refresh!

Tip

If you're using the `docker-compose.yaml` setup described earlier, you will only see 2 environment variables here... which are the only 2 you need anyways.

Yay! We see 4 environment variables including 2 we need! The first one is just a flag that tells us that the Symfony binary is running Mercure... and that last one is there for legacy reasons: we don't need it.

This means that our app is already configured and *ready* to talk to our Mercure Hub! In production, you'll need to run a *real* Mercure Hub and set these environment variables manually, however you do that in your hosting environment.

So... we have a Mercure hub running! What does that... mean? Well, it's a central place where some things can *listen* for messages and other things can *publish* messages. Next, let's do both of these things: listen to a Mercure "topic" in JavaScript and publish messages to it, both from the command line - just to see how it works - and from PHP, which is our *real* goal.

Chapter 49: Listening & Publishing

The purpose of Mercure is to have a hub where we can subscribe - or *listen* - to messages and also *publish* messages.

Here's our high-level goal, it's three steps. First, set up some JavaScript that listens to a "topic" in Mercure - a topic is like a message key or category. Second, in PHP, publish a message *to* that topic containing Turbo Stream HTML. And finally, when our JavaScript receives a message, make it pass the Turbo Stream HTML to the stream-processing system. The result? The power to update *any* part of *anyone's* page *whenever* we want to right from PHP. If this doesn't make sense yet, don't worry: we're going to put this into action right now.

But before we jump in, open `index.php` and remove the dump... so that our site is no longer dead. Excellent.

Listening in JavaScript via the Stimulus Controller

Ok, step 1: open `templates/product/reviews.html.twig`, which is the template that holds the entire reviews turbo frame. At the top, or really anywhere, add a `div`. Where its attributes live, render a new Twig function from the UX library we installed a few minutes ago - `turbo_stream_listen()` - and pass this the name of a "topic"... which could be anything. How about `product-reviews`. Then, close the `div`.

```
templates/product/_reviews.html.twig
<div {{ turbo_stream_listen('product-reviews') }}></div>
// ... lines 2 - 43
```

I know, that looks kind of weird. To see what it does, go refresh a product page... and inspect the reviews area to find this `div`. Here it is.

Ok: this `div` is a *dummy* element. What I mean is: it won't *ever* contain content *or* be visible to the user in any way. Its *real* job is to activate a Stimulus controller that listens for messages in the `product-reviews` topic. You can see the `data-controller` attribute pointing to the controller we installed earlier as well as an attribute for the `product-reviews` topic and the public URL to our Mercure hub.

Viewing a Mercure Topic in your Browser

Go to your network tools and make sure you're viewing fetch or XHR requests. Scroll up. Woh! There was a request to our Mercure hub *with* `?topic=product-reviews`. The Stimulus controller did this.

But the *really* interesting thing about this request is the "type": it's not fetch or XHR, it's `eventsource`. Right Click and open this URL in a new tab. Yup, it just spins forever. But not because it's broken: this is working perfectly. Our browser is *waiting* for messages to be published to this topic.

Publishing Messages via curl

We are now listening to the `product-reviews` topic both in this browser tab and, apparently, from some JavaScript on this page thanks to the Stimulus controller we just activated. So... how can we publish messages *to* that topic?

Tip

A *cooler* way to debug with Mercure is to go to `http://127.0.0.1:<random_port>/.well-known/mercure/ui/` to see an interactive, debugging Mercure dashboard where you can listen and publish messages.

Basically... by sending a POST request to our Mercure hub. Over in its documentation, go to the "Get Started" page and scroll down a bit down. Here we go: publishing. This shows an example of how you can publish a basic message to Mercure. Copy the `curl` command version. Then, over my editor, I'll go to File -> "New Scratch File" to create a plaintext scratch file. I'm doing this so we have a convenient spot to play with this long command.

In fact, it's so long that I'll add a few `\` so that I can organize it onto multiple lines. This makes it a *bit* easier to read... but I know, it's still pretty ugly.

Before we try this, change the topic: the example is a URL, but a topic can be any string. Use `product-reviews`. And at the end, update the URL that we're POSTing to so that it matches our server: 127.0.0.1:8000.

We'll talk about the other parts of this request in minute. For now, copy this, find your terminal, paste and... hit enter! Okay: we got a response... some `uuid` thing. Did that work?

Spin back over to your browser tab. Holy cats, Batman! It showed up! Our message contained this JSON data... which *also* appears in our tab.

The Parts of a Publish Request

Even if you're not super comfortable using `curl` at the command line - honestly, I do this *pretty* rarely - most of what's happening is pretty simple. First: we're sending a `topic` POST parameter set to `product-reviews` *and* a `data` POST parameter set to... well, whatever we want! For the moment, we're sending some JSON data, which is passed to anyone listening to this topic.

At the end of the command, we're making this a POST request to our Mercure Hub URL. But what about this `Authorization: Bearer` part... with this super long key? What's that? It's a JSON web token. Let's learn more about what it is, how it works and where it came from next. It's the *key* to convincing the Mercure Hub that we're allowed to *publish* messages to this topic.

Chapter 50: Mercure Hub's JWT Authorization

Can *anyone* publish a message to *any* topic on a Mercure hub? Definitely not. So how does the Mercure Hub know that *we* are allowed to publish this message? It's entirely thanks to this *long* string that we're passing to the `Authorization` header.

Where does this come from? It turns out, it's a JSON web token. Copy that *huge* string... then head over to jwt.io: a *lovely* site for working with JSON web tokens - or JWT's. If you're familiar with how JWT's work, *awesome*. If not, here's a little primer.

A JWT + Mercure Primer

Scroll down a bit to find a JWT editor. Paste in the encoded token. So this weird string here can actually be decoded to this *JSON*. And... you don't need a secret key to do it: the long string is basically just a base64 encoded version of this JSON. *Anyone* can turn this string into this JSON.

So when we send this long string to the server, what we're *really* sending is this JSON data. For us, the `subscribe` part isn't important... and neither is the `payload`. But the `publish` part *is* important. This basically says:

> *“Hi Mercure Hub! Guess what! I have permission to publish to any topic. Cool, huh?”*

Ok... but why does the Mercure server trust this? Can't *anyone* create a JSON web token that *claims* that *they* can publish to all topics? Yea! But those wouldn't be *signed* correctly unless they have the "secret".

When you run a Mercure Hub, you give it a "secret" value... which, by default - and for *our* Mercure Hub - is `!ChangeMe!`. This is the value that you see in our `.env` file.

Back over on `jwt.io`, look at the bottom. It says "invalid signature". When a JWT is created, it's *signed* by a secret key. When someone *uses* a JWT, after decoding it - which anyone can do - they are then supposed to verify the *signature* of the token. Right now, it's trying to verify the signature of our JWT... but using the wrong secret. If we paste in our *real* secret instead... it's verified!

This can... be a bit technical. The point is this: in order to generate a JWT that will have a *valid* signature, you need the secret. And while anyone can *read* a JWT, if you mess around with its contents, the signature will fail. *That's* why the Mercure Hub *trusts* us when we send a JWT that says we can publish to *any* topic: the signature of our message is valid. That means it was generated by someone who has the secret key.

Check this out: let's regenerate this JWT using the *same* "payload" but signed using the *wrong* secret... something a bad user might try to do. Copy the new JWT... update the `curl` command in our scratchpad... copy the whole command... and paste it into the terminal. Hit enter. Unauthorized! The Mercure Hub can *totally* read the JSON in this message, but it sees that the signature failed and does *not* publish the message.

Change back to our old key in the scratch pad. And at the browser, use the correct secret: `!ChangeMe!`.

Simplifying the Payload

To simplify things, change the payload to *just* the part we need. So remove the `subscribe` part - we're not trying to get access to subscribe to anything - and also remove `payload`. *This* is all we *really* need: some JSON that claims that we can publish to *any* topic *signed* with the correct secret. If you ever need to create a JWT by hand, this is how you do it: create the JSON you want and have something - like this site - *sign* it with your secret.

Copy the new, shorter JWT... and paste it in our scratchpad. Copy the entire command, paste it at your terminal and... yes! It works! In our browser, the listening tab shows a *second* message.

Publishing a Turbo Stream

Enough about authorization & JWT. In the real world, as long as we have the correct `MERCURE_SECRET` configured in our app, all of this will be handled automatically thanks to the Mercure PHP library. Internally, *it* will use the secret to generate the signed JWT *for* us.

But before we start publishing messages from our code, let's look closer at the `data` POST parameter. So far, we've been sending JSON. And, in theory, we could write some JavaScript that listens to this topic and *does* something with that JSON. But remember: the `turbo_stream_listen()` function activates a Stimulus controller that is already listening to this topic. It's listening and waiting for a message whose data isn't JSON, but `<turbo-stream>` HTML.

Check it out: over in our scratch pad, instead of setting the `data` to JSON, I'll paste in a turbo stream. It's a little ugly because it's all on one line, but it's valid: `action="update`, `target="product-quick-stats"` with some dummy content inside.

Let's *first* see if this message shows up inside our browser tab. Oh! It actually stopped listening. It probably hit a listening timeout - that's something you can configure or disable in Mercure. I'll refresh.

Now, go copy the command... find your terminal, paste, hit enter... and head back to the browser. No surprise: here's our message *with* the Turbo Stream HTML. But the *really* cool thing is back on our site. Scroll up. Yes! It updated the quick stats area! As *soon* as we published the message, the JavaScript from the Stimulus controller *saw* the message and passed the turbo-stream HTML to the stream-processing system. That's *so* cool.

Of course, we aren't normally going to publish via the command line & curl: we're going to publish messages via PHP... which is *way* easier. Let's do that next.

Chapter 51: Publishing Mercure Updates in PHP

We *now* know that we can *easily* subscribe to a Mercure topic in JavaScript. *And*, if we publish a message *to* that topic with `<turbo-stream>` HTML in it, our JavaScript will instantly notice & process it. Sweet!

So far, we've published messages to Mercure via curl at the command line... but that was just to get a feel for how it works. In reality, we're going to publish message from PHP... which is a heck of a lot simpler anyways.

Copy the `<turbo-stream>`... then go find `ProductController`... and the reviews action.

Publishing a Message from PHP

To publish updates to a Mercure Hub, we need to autowire a new service. Use `HubInterface` and I'll call it `$mercureHub`.

Down below, to start, let's publish an update when we submit the form... but not necessarily when it's successful. I'm lazy: this will let us test without filling out the form successfully. Add a variable - `$update` - set to `new Update()` - a handy class for creating messages. We need to pass this two arguments. The first is the topic or topics that we want to publish to. Use `product-reviews`. The second is the *data* that we want to send. Paste in the `<turbo-stream>` string.

Below, to *publish* this, all we need is `$mercureHub->publish($update)`.

```
src/Controller/ProductController.php
// ... lines 1 - 20
class ProductController extends AbstractController
{
// ... lines 23 - 71
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager, HubInterface $mercureHub)
    {
// ... lines 77 - 82
        if ($request->isMethod('POST')) {
// ... lines 84 - 85
            $update = new Update(
                'product-reviews',
                '<turbo-stream action="update" target="product-quick-stats"><template>QUICK STATS CHANGED!</template></turbo-stream>'
            );
            $mercureHub->publish($update);
// ... lines 91 - 109
        }
// ... lines 111 - 117
    }
// ... lines 119 - 126
}
```

Kind of... beautiful, isn't it?

Let's try this! Find your browser and refresh so the quick stats area is restored. Scroll down and submit the form empty. Uh... 500 error? Open the profiler for that request. Hmm:

> *“Failed to send an update”*

Setting verify_peer to False in dev for Macs

Not... very explanatory. But notice that there were *four* exceptions. When this happens, it's often one of the *other* exceptions that has more details. Ah:

> *“SSL peer certificate or SSH remote key was not okay”*

This... is a problem specific to the Symfony binary web server, https and... Macs. You can learn more about it on this issue for the Symfony CLI. If you're not using a Mac, good for you! That hopefully just worked.

If you *are*, the easiest way to fix this is to disable "peer verification" in the `dev` environment.

To do this, open `config/packages/framework.yaml`. At the bottom, use `when@dev` to set config specific to the `dev` environment - that's a feature that's new to Symfony 5.3. Under this, set `framework`, `http_client`, `default_options` then `verify_peer: false`.

config/packages/framework.yaml

```yaml
// ... lines 1 - 25
when@dev:
    framework:
        http_client:
            default_options:
                verify_peer: false
```

That's *not* something you want to set in production... and it's a bummer we need to set it in the `dev` environment. But it *should* fix our issue.

Close this... then refresh the page again. Scroll down... and submit the review form. Ok! We get the normal validation error - that's expected. But scroll up. Yes! We just updated the page with our stream through Mercure! That's awesome!

So next: let's use this new superpower to simplify our reviews action. We can now redirect on success like we originally were... *and* publish a stream to update the quick stats area.

Chapter 52: Turbo Stream for Instant Review Update

When we submit a new review, we update two different parts of the page. First, the review list and review form. And second, the quick stats area up here.

Over in `ProductController`, in the reviews action, we do this by returning a turbo stream: `reviews.stream.html.twig` is responsible for updating both spots.

Cool, but remember that the reviews list and review form live inside of a turbo frame. And so, before we started messing around and doing crazy stuff with Turbo Streams, we updated that section simply by returning a redirect to the reviews page on success. The Turbo Frame followed that redirect, grabbed the matching `<turbo-frame>` from that page and updated it here.

Unfortunately... as soon as we wanted to *also* update the quick stats area, we had to change *completely* to rely on turbo streams. The problem is that we can't return a turbo stream *and* a redirect from the controller.... so we chose to return a stream... which means that the stream needs to update *both* sections of the page.

Returning a Redirect And Publishing a Stream

Okay. So why are we talking about all of this again? Because now that we have Mercure running, we *can*, in a sense, return two things from our controller. Check it out: copy this dummy Mercure update code, remove it... and paste it down in the success area.

We're updating the `product-reviews` stream, which is the stream that we're listening to thanks to our code in `_reviews.html.twig`. Back in the controller, instead of *returning* a stream, copy the render line, delete that section, paste inside the update... and fix the formatting. Oh, also change this to `renderView()`: `render()` returns a `Response` object... but all we need is the *string* from this template. That's what `renderView()` gives us.

```
src/Controller/ProductController.php
// ... lines 1 - 20
class ProductController extends AbstractController
{
// ... lines 23 - 71
    /**
     * @Route("/product/{id}/reviews", name="app_product_reviews")
     */
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager, HubInterface $mercureHub)
    {
// ... lines 77 - 82
        if ($request->isMethod('POST')) {
            $this->denyAccessUnlessGranted('ROLE_USER');

            $reviewForm->handleRequest($request);

            if ($reviewForm->isSubmitted() && $reviewForm->isValid()) {
                $entityManager->persist($reviewForm->getData());
                $entityManager->flush();

                $update = new Update(
                    'product-reviews',
                    $this->renderView('product/reviews.stream.html.twig', [
                        'product' => $product,
                    ]),
                );
                $mercureHub->publish($update);

                $this->addFlash('review_success', 'Thanks for your review! I like you!');

                return $this->redirectToRoute('app_product_reviews', [
                    'id' => $product->getId(),
                ]);
            }
        }
// ... lines 107 - 113
    }
// ... lines 115 - 122
}
```

Thanks to this, our controller will now redirect like it did before... but it will *also* publish a stream to Mercure along the way.

Let's try it. Refresh the page... and scroll all the way down to the bottom. I want to trigger the weather widget Ajax call just so that we can cleanly see what happens with the network requests when we submit. Clear out the Ajax requests... then add a new review.

Cool! It looks like that worked! Check out the network requests. The first is the POST form submit. This returned a redirect, the frame system *followed* that redirect, found the frame on the next page, and updated this area. The normal Turbo Frames behavior. *Then* our *stream* caused the quick stats area to update... and it also re-updated the reviews area... because, right now, our stream template is still updating *both* things.

Only Streaming the Quick Stats

So probably we could stop streaming the `_reviews.html.twig` template... since the turbo-frame is taking care of that part of the page. *We* only need to focus on updating the quick stats.

```
templates/product/reviews.stream.html.twig
```

```
<turbo-stream action="update" target="product-quick-stats">
    <template>
        {{ include('product/_quickStats.html.twig') }}
    </template>
</turbo-stream>
```

Let's try this again. Right now we have 16 reviews. Head down and add the 17th. Ah! Silly validation! Type a bit more and submit. Yes! It still works! The behavior *is* slightly different than before: it renders a new review form... because that's what's rendered inside the `<turbo-frame>` on the redirected page. And... up above, the quick stats area *did* update.

So this is a really pure example of a turbo-stream in action. Inside of our `ProductController`, we can just redirect like normal, which powers the turbo-frame. Then, the minute that we realize that we need to update a *different* part of the page - something *outside* of the frame - we can do that through Mercure.

Updating the Page of Every User

But this is even cooler than it looks at first. In `reviews.stream.html.twig`, temporarily put *back* the `product-review` stream.

```
templates/product/reviews.stream.html.twig
<turbo-stream action="update" target="product-quick-stats">
    <template>
        {{ include('product/_quickStats.html.twig') }}
    </template>
</turbo-stream>

<turbo-stream action="replace" target="product-review">
    <template>
        {{ include('product/_reviews.html.twig') }}
    </template>
</turbo-stream>
```

Back at your browser, copy the URL and open this page in a second tab. Make sure both pages are refreshed. Ok: both show 17 reviews. In the original tab, scroll down and submit review number 18. It *does* show up here: no surprises.

Now check out the other tab. The quick stats *also* update here! And, down below, yup! There's review number 18! That's amazing! Sure, I'm sitting on one computer with two tabs open. But if two people - on opposite sides of the planet - were both viewing this page at the same time, the *same* thing would happen. When we post a new review, *everyone's* page is updated!

This opens up a new possibility for turbo streams. We already know that we can use streams to update any part of our page, like something that's outside of the frame that we're currently working in. But we can *also* use streams to update *any* part of *any* user's page... so that when a user in Belgium adds a new review, a different user in Japan - who was already on that page - will instantly see it.

Making Update Ids Specific to the Product

But now, in the second tab, navigate to a different product. Back in the first, post review number 19. When I submit, this, of course, works. But check out the second tab. Woh! This product should *not* have 19 reviews... and all of these reviews are for the *other* product, not this one! Refresh. Yup! This product has *way* less reviews. Our stream update is affecting *every* product page!

And... this makes sense. If you're on a product page - *any* product page - then you're listening to the `product-reviews` Mercure topic. When we publish an update, we target the `product-quick-stats` and `product-review` elements... both of which exist on *every* product page!

Fortunately, this is simple to fix. In `_reviews.html.twig`, we need to make sure that every element that we target with a turbo stream has a dynamic part in it so that it's *specific* to that product. In the `id` attribute, change it to `product-{{ product.id }}-review`.

```
templates/product/_reviews.html.twig
// ... lines 1 - 2
<turbo-frame id="product-{{ product.id }}-review">
// ... lines 4 - 41
</turbo-frame>
```

In `reviews.stream.html.twig`, do the same thing so they match. Repeat this for the quick stats, which lives in `show.html.twig`... here it is. Add `{{ product.id }}` inside the id. Copy that... and in the stream template, add it here too.

```
templates/product/show.html.twig
// ... lines 1 - 32
                <div id="product-{{ product.id }}-quick-stats">
// ... line 34
                </div>
// ... lines 36 - 49
```

```
templates/product/reviews.stream.html.twig
<turbo-stream action="update" target="product-{{ product.id }}-quick-
stats">
// ... lines 2 - 4
</turbo-stream>
// ... line 6
<turbo-stream action="replace" target="product-{{ product.id }}-review">
// ... lines 8 - 10
</turbo-stream>
```

Perfect. If two users are viewing two different products, they will *still* both be listening to the same Mercure topic. When a review is posted to the first product, the second user *will* receive the update... but they won't have any elements matching those ids on their page. So, it will do nothing.

Click to post another review. Ah! That killed the frame! Of course: we just changed the id of the frame... so we need to refresh. Post one more review. It shows up here... but it did *not* affect the other page.

Ok: thanks to the new system, we can simplify our turbo stream even more to deliver *exactly* the updates we want to every user. That's next.

Chapter 53: Smartly Updating Elements for all Users

With the power to return a normal redirect from our controller *and* publish a Mercure update to modify any part of *any* user's page, we can now *really* clean up our review system. After a successful form submit, we redirect to a page that renders `_reviews.html.twig`... which includes the reviews list on top and also the review form down here. Then... we send this *same* thing to the user via the stream update. The only reason we're doing this is so that the review list updates for *all* users, not just the user that submitted the form.

So... you can see that there's some duplicated work going on. But worse, there's a bug! Copy the URL and open this same page in an incognito window. Notice that we are *not* logged in. Let's pretend that this tab represents a user in Argentina... and the other tab is a user in Ukraine.

Let's refresh and have our Ukrainian friend submit a new review... this will be review number 21. When we submit, it looks good here. On the *other* user's page, the review shows up... but oh! It also shows a success message! So when our Ukrainian user submitted a new review, our Argentinian friend suddenly saw a success message!

That's... ya know... *not* what we want. But I can already see the problem: in the turbo stream, we're sending the *entire* `_reviews.html.twig` template to *all* users... which includes the reviews list... but also the flash message and the form.

Splitting the Reviews Frame and Stream

No worries: we just need to be a bit more careful. The entire `_reviews.html.twig` template is surrounded by a `<turbo-frame>`. But we really only need the frame to surround the *form*... because we can update the reviews *list* via the stream.

Check it out: at the bottom of the reviews list, close that `<turbo-frame>`. Now, create a *new* `<turbo-frame>` with `id=""`, how about, `product-reviews-form`. We don't need a closing tag... because we already have one.

```
templates/product/_reviews.html.twig
// ... lines 1 - 2
<turbo-frame id="product-{{ product.id }}-review">
// ... lines 4 - 13
</turbo-frame>

<hr>

<turbo-frame id="product-reviews-form">
// ... lines 19 - 44
</turbo-frame>
```

Oh, and in this case, we *don't* need to make the `id` dynamic for each product because we're not going to update this with a Turbo Stream. So there's no risk of affecting the wrong page.

With *just* this change, the form now lives in a different frame. And so, if we were to refresh the page and submit the form... it now only affects this part of the page.

The next step is to make sure that our stream update sends back the *list*, not the list *and* the form. To do that, we need to isolate the list into its own template. Copy that turbo frame and, inside `templates/product/`, create a new file called, how about, `_reviews_list.html.twig`. Paste the frame here.

```
templates/product/_reviews_list.html.twig
<turbo-frame id="product-{{ product.id }}-review">
{% for review in product.reviews %}
    <div class="component-light my-3 p-3">
        <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }}
<i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
        <div>
            {{ review.content }}
        </div>
    </div>
{% else %}
    <p>This product has not been reviewed yet!</p>
{% endfor %}
</turbo-frame>
```

Back in the other template, include this.

```
templates/product/_reviews.html.twig
<div {{ turbo_stream_listen('product-reviews') }}></div>

{{ include('product/_reviews_list.html.twig') }}
// ... lines 4 - 6
<turbo-frame id="product-reviews-form">
// ... lines 8 - 33
</turbo-frame>
```

Nice. Oh, but in the new template, we don't actually need this to be a Turbo frame anymore. Change this to be a `div`. Think about it: we're not using any Turbo frame features with this... we just need an element that we can target from our turbo stream. A `turbo-frame` *would* have worked... it just wasn't necessary.

```
templates/product/_reviews_list.html.twig
<div id="product-{{ product.id }}-review">
// ... lines 2 - 11
</div>
```

Anyways, stream this template instead: `_reviews_list.html.twig`.

Sweet! Testing time! Refresh both tabs... and let's post review number 22.

When I submit here... perfect! The review form area updated thanks to the frame. Then the *stream* took care of adding the review here *and* updating the quick stats area. In the other browser, the quick stats updated, we see the new review, but it did *not* mess with the form area.

Appending the New Review

Look back at `reviews.stream.html.twig`. Right now we're streaming and replacing the *entire* reviews list. That's probably fine... but because we know that a single new review was just added, we could, instead, send only the *new* review in the stream instead of *everything*. We don't *have* to do this, but let's try it.

First, over in `_reviews.html.twig`, on the `id`, I'm going to add a `-list` to the end. I'm doing this *just* to make its meaning more obvious: it's a *list*, not a single review. Repeat this in the stream template.

templates/product/_reviews_list.html.twig

```
<div id="product-{{ product.id }}-review-list">
// ... lines 2 - 11
</div>
```

templates/product/reviews.stream.html.twig

```
// ... lines 1 - 6
<turbo-stream action="replace" target="product-{{ product.id }}-review-list">
    <template>
        {{ include('product/_reviews_list.html.twig') }}
    </template>
</turbo-stream>
```

Now over in `_reviews_list.html.twig`, copy the `div` for a single review and isolate it into *its* own template: `_review.html.twig`. Back in the list, include that.

templates/product/_review.html.twig

```
<div class="component-light my-3 p-3">
    <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }} <i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
    <div>
        {{ review.content }}
    </div>
</div>
```

templates/product/_reviews_list.html.twig

```
<div id="product-{{ product.id }}-review-list">
{% for review in product.reviews %}
    {{ include('product/_review.html.twig') }}
{% else %}
    <p>This product has not been reviewed yet!</p>
{% endfor %}
</div>
```

So no changes yet, just some reorganization. But now, in the stream, change the action to `append`... and include the single review template.

That's nice! In `_review.html.twig`, this needs a `review` variable. In `ProductController`... let's see: we're only passing a `product` variable right now. Also pass a `newReview` variable set to the review... which is `$form->getData()`.

Back in the stream, pass in a `review` variable set to `newReview`.

```
src/Controller/ProductController.php
// ... lines 1 - 20
class ProductController extends AbstractController
{
// ... lines 23 - 74
    public function productReviews(Product $product, CategoryRepository $categoryRepository, Request $request, EntityManagerInterface $entityManager, HubInterface $mercureHub)
    {
// ... lines 77 - 91
                $update = new Update(
                    'product-reviews',
                    $this->renderView('product/reviews.stream.html.twig', [
                        'product' => $product,
                        'newReview' => $reviewForm->getData(),
                    ]),
                );
// ... lines 99 - 114
    }
// ... lines 116 - 123
}
```

```
templates/product/reviews.stream.html.twig
// ... lines 1 - 6
<turbo-stream action="append" target="product-{{ product.id }}-review-list">
    <template>
        {{ include('product/_review.html.twig', {
            review: newReview,
        }) }}
    </template>
</turbo-stream>
```

Let's try the *whole* flow. Refresh both tabs. We're filling in review number 23. Submit and... sweet! Three things just happened. First the form area updated thanks to the Turbo frame system. Second, the new review was appended to the list thanks to the stream. And finally, the quick stats area was updated *also* thanks to the stream.

Over in the incognito tab, it's almost the same. The reviews list has the new review and the quick stats area updated... all *without* affecting the form area.

Next: let's celebrate by visually highlighting the new review the moment it pops onto the page.

Chapter 54: Visually Highlighting new Items that Pop onto the Page

Our review system is super cool: if *any* user submits a review, that review will pop onto the page of anyone *else* that's currently viewing this product.

To make this a bit more obvious, I want to highlight the new review as soon as it appears. And this is pretty easy. Start over in `assets/styles/app.css`. Add a `.streamed-new-item` style with a `background-color` set to `lightgreen`.

```
assets/styles/app.css
// ... lines 1 - 181
.streamed-new-item {
    background-color: lightgreen;
}
```

Adding a Green Background to New Items

Let's *add* this class to a new review *if* it's added via a stream. We can do this in `reviews.stream.html.twig`: pass a new variable into the template called `isNew` set to `true`.

```
templates/product/reviews.stream.html.twig
// ... lines 1 - 6
<turbo-stream action="append" target="product-{{ product.id }}-review-list">
    <template>
        {{ include('product/_review.html.twig', {
            review: newReview,
            isNew: true
        }) }}
    </template>
</turbo-stream>
```

Now, over in *that* template - `_review.html.twig` - at the end of the class list, use the ternary syntax: if `isNew` - and `default` this to `false` if the variable is *not* passed in - then print `streamed-new-item`.

```
templates/product/_review.html.twig
<div class="component-light my-3 p-3{{ isNew|default(false) ? ' streamed-new-item' }}">
    <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }} <i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
    <div>
        {{ review.content }}
    </div>
</div>
```

That's it. The "else" is automatic: if `isNew` is false, this will print nothing.

Let's check it out! Refresh both of the pages to get the new CSS... and then submit a new review. Yay! The green background shows up here... *and* on the page of *everyone* on the planet that happens to be viewing this page.

So... this is cool. But... we need more fancy! What if we show this background for only five seconds and then fade it out. Start again in `app.css` to set up the fading out part: we need a new class that describes this transition. Add a `fade-background` class that declares that we want any `background-color` changes to happen gradually over 2000 milliseconds.

```
assets/styles/app.css
// ... lines 1 - 181
.streamed-new-item {
    background-color: lightgreen;
}
.fade-background {
    transition: background-color 2000ms;
}
```

A Stimulus Controller to Fade Out

Before we try to use this somewhere directly, let's stop and think. If the goal is to remove this background after 5 seconds, then the only way to accomplish that is by writing some custom JavaScript. In other words, we need a Stimulus controller! In the `assets/controllers/` directory, create a new file called, how about, `streamed-item_controller.js`. I'll paste in the normal structure, which imports turbo, exports the controller and creates a `connect()` method.

```
assets/controllers/streamed-item_controller.js
import { Controller } from 'stimulus';

export default class extends Controller {
// ... lines 4 - 6

    connect() {
// ... line 9
    }
}
```

Before we fill this in, go over to `_review.html.twig` and use this. I'll break this onto multiple lines.. cause it's getting kind of ugly. Copy the class name, but delete the custom logic. Replace it with a normal if statement: if `isNew|default(false)`, then we want to *activate* that new Stimulus controller. Do that with `{{ stimulus_controller('streamed-item') }}`. Oh, and pass a second argument, I want to pass a variable *into* the controller called `className` set to `streamed-new-item`.

```
templates/product/_review.html.twig
<div
    class="component-light my-3 p-3"
    {% if isNew|default(false) %}
        {{ stimulus_controller('streamed-item', {
            className: 'streamed-new-item'
        }) }}
    {% endif %}
>
    <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }} <i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
    <div>
        {{ review.content }}
    </div>
</div>
```

I'm doing this for two reasons. First, it will now be the responsibility of the *controller* to add this class to the element. We'll do that in a minute. And second, while we don't need it now, making this class name dynamic will help us reuse this controller later.

Anyways, head back to the controller and define the value: `static values = {}` an object with `className` which will be a `String`.

Cool. Down in `connect()`, add that class to the element: `this.element.classList.add()` and pass `this.classNameValue`.

```
assets/controllers/streamed-item_controller.js
import { Controller } from 'stimulus';

export default class extends Controller {
    static values = {
        className: String
    }

    connect() {
        this.element.classList.add(this.classNameValue);
    }
}
```

If we stopped right now... this would just be a really fancy way to add the `streamed-new-item` class to the element as soon as it pops onto the page.

So let's do our *real* work. Use `setTimeout()` to wait 5 seconds... and then... if I steal some code... *remove* `this.classNameValue`.

If we just did this, after five seconds, the green background would suddenly disappear. To activate the transition when the background is removed, add *another* class: `fade-background`.

```
assets/controllers/streamed-item_controller.js
import { Controller } from 'stimulus';

export default class extends Controller {
    static values = {
        className: String
    }

    connect() {
        this.element.classList.add(this.classNameValue);
        setTimeout(() => {
            this.element.classList.add('fade-background');
            this.element.classList.remove(this.classNameValue);
        }, 5000);
    }
}
```

If you wanted to be really fancy, you could wait until the transition finishes and then remove this class to clean things up. But this will work fine.

Let's try it! Refresh both tabs so that we get that new CSS... then go fill in another review. When we submit... good! A green background here... *and* in the other browser. If we wait... beautiful! It faded out! How nice is that?

Ok team, we're currently publishing updates to Mercure from inside of our controller. But the Mercure Turbo UX package that we installed earlier makes it possible to publish updates *automatically* whenever an entity is updated, added or removed. It's pretty incredible, and it's our next topic.

Chapter 55: Entity Broadcast

There's one super cool feature of the Turbo Mercure UX package that we installed earlier that we have *not* talked about. And it's this: the ability to publish a Mercure update automatically whenever an entity is created, updated or removed. It's a *powerful* idea.

For example, instead of publishing this update from inside of our controller, what if we published this update *whenever* a review is added to the system, regardless of how or *where* it's added? Or what if we could publish a Mercure update whenever a review is changed... like if we changed a review in an admin area, that review would automatically re-render on *anyone's* page that was currently viewing it!

The Broadcast Attribute/Annotation

That is *totally* possible. Go into the entity where you want to activate this behavior. For us that's `src/Entity/Review.php`. Above the class, add `@Broadcast`.

```
src/Entity/Review.php
<?php
// ... line 2
namespace App\Entity;
// ... lines 4 - 9
/**
 * @ORM\Entity(repositoryClass=ReviewRepository::class)
 * @Broadcast()
 */
class Review
{
// ... lines 16 - 109
}
```

If you're using PHP 8, you can also use `Broadcast` as an attribute. Next, open `templates/products/_reviews.html.twig`. This is where we originally used `turbo_stream_listen()` to *listen* to the `product-reviews` Mercure topic.

Copy that and, temporarily, *also* listen to a topic called `App\Entity\Review`. We need the double slashes to avoid escaping problems. Oh, and not *reviews*, just `Review`: the name of the class.

```
templates/product/_reviews.html.twig
<div {{ turbo_stream_listen('product-reviews') }}></div>
<div {{ turbo_stream_listen('App\\Entity\\Review') }}></div>
// ... lines 3 - 36
```

Okay: whenever a `Review` is created, change or removed, an update will be sent to the `App\Entity\Review` topic on our Mercure hub. And now we're *listening* to that topic.

If this doesn't all make sense yet, don't worry: we're missing one important piece. To find out what it is, let's fearlessly forge ahead and try this! Refresh the page and check out the network tools. Let's see... here it is! We're listening to a new stream URL. Open this in a new tab. Like with the other topic, our browser spins and waits for updates.

The Broadcast Template

Ok! Try to submit a new `Review`. And oh! A 500 error. Open the profiler for that request to see what happened:

> *"Unable to find template `broadcast/Review.stream.html.twig`."*

Okay. So here's the *whole* flow that we activated by adding the `@Broadcast` annotation above the entity. First, we create, change or remove a `Review` from the database. Second, the Turbo Mercure library *notices* this and tries to render a template called `Review.stream.html.twig`. We will create this in a moment. And third, whatever this template renders is published to Mercure... in a specific way.

Let's go create that template. In the templates directory, add the `broadcast` directory... and inside, the new file: `Review.stream.html.twig`.

These "broadcast" templates always look the same, and I'm going to paste in a skeleton to show you. It's... kind of a cool use of blocks. If a `Review` is created, the content in the `create` block is sent to Mercure. If a `Review` is updated, the `update` block is used. And if we delete a `Review`, the contents of the `remove` block are published to Mercure as an update.

templates/broadcast/Review.stream.html.twig

```
{% block create %}
    CREATE!
{% endblock %}

{% block update %}
    UPDATE!
{% endblock %}

{% block remove %}
    REMOVE!
{% endblock %}
```

We can see this immediately. Close the profiler tab, refresh this page... and add another review. When we submit, nothing looks different *here* yet. But check out the tab that's listening to Mercure. Yes! There it is! This published an update and passed the contents from our `create` block as that update's *data*!

Publishing turbo-stream Updates

Now we're dangerous. Go into the original `reviews.stream.html.twig` template, copy both streams and paste them into our `create` block.

templates/broadcast/Review.stream.html.twig

```
{% block create %}
    <turbo-stream action="update" target="product-{{ product.id }}-quick-stats">
        <template>
            {{ include('product/_quickStats.html.twig') }}
        </template>
    </turbo-stream>

    <turbo-stream action="append" target="product-{{ product.id }}-review-list">
        <template>
            {{ include('product/_review.html.twig', {
                review: newReview,
                isNew: true
            }) }}
        </template>
    </turbo-stream>
{% endblock %}
// ... lines 17 - 25
```

Boom, done! We can now completely delete `reviews.stream.html.twig`. And inside of `ProductController`, we don't need to dispatch this update at *all* anymore. It will happen automatically when we create the `Review`. So I'll delete the `Update`, the `$mercureHub` argument... and if you want to get really crazy, you can clean up the unused `use` statements on top.

Finally, in `_reviews.html.twig`, we no longer need to listen to our original `product-reviews` topic.

The "entity" Variable

Testing time! go back, refresh... and publish a new review. Ah! Another 500 error? Let's check out what happened:

> *“Variable `product` does not exist coming from `Review.stream.html.twig`.”*

Ah! So Apparently there is *not* a product variable that's passed to this template... which begs the question: what variables *are* passed to this template? When the Mercure Turbo library renders this template, it passes several variables. The most important - by far - is a variable called `entity`... which in this case will be set to the `Review` object. We can use that to fix our template.

So instead of `product.id`, we need `entity.product.id`. Do that in both places. And this template also needs a `product` variable... so pass that in set to `entity.product`. And down here, `review` is now `entity`.

```
templates/broadcast/Review.stream.html.twig
{% block create %}
    <turbo-stream action="update" target="product-{{ entity.product.id }}-quick-stats">
        <template>
            {{ include('product/_quickStats.html.twig', {
                product: entity.product
            }) }}
        </template>
    </turbo-stream>

    <turbo-stream action="append" target="product-{{ entity.product.id }}-review-list">
        <template>
            {{ include('product/_review.html.twig', {
                review: entity,
                isNew: true
            }) }}
        </template>
    </turbo-stream>
{% endblock %}
// ... lines 19 - 27
```

Hopefully that's everything. Close the error, refresh the page... and add a new review. If all goes well, this will have the *same* behavior as before. Submit. We got it! The new review loaded onto the page thanks to the stream! The quick stats area *also* updated. In the other tab, yup! The new review streamed here too!

The *big* difference now is that the stream update will be published no matter *how* a review was created.

Next, let's also instantly update every user's page whenever a review is changed or removed. I'll show you a review admin area that's been hiding on our site where we can watch this in real time.

Chapter 56: Broadcasting Frontend Changes on Entity Update/Remove

In `Review.stream.html.twig`, we have the ability to publish turbo streams automatically whenever a review is created, updated or removed. That's pretty cool. *Unrelated* to this, I haven't mentioned it yet, but our site has a review admin area! You can get it by going to `/admin/review`. Here we can create, update or delete reviews. Do you... see where this is going? Sometimes an admin user will "tweak" a review to make it... um... more *encouraging*. Wouldn't it be cool if, when an admin user did this, that review was instantly updated on the frontend for all users?

Uh, yea! That *would* be cool! So let's do it.

Publishing an "update" Update

Start in `_review.html.twig`. This is the template that renders a single `Review`. Give this element an id so that we can target it from a turbo stream, how about `id="product-review-{{ review.id }}"`.

templates/product/_review.html.twig

```
<div
    id="product-review-{{ review.id }}"
    class="component-light my-3 p-3"
    {% if isNew|default(false) %}
        {{ stimulus_controller('streamed-item', {
            className: 'streamed-new-item'
        }) }}
    {% endif %}
>
    <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }} <i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
    <div>
        {{ review.content }}
    </div>
</div>
```

Copy that value and, in `Review.stream.html.twig`, when the review is updated, let's add a new turbo stream: `<turbo-stream>` with `action="replace"` and `target=""` set to `product-review-{{ review.id }}`. Except that in *this* template, the variable is called `entity`.

Inside, add the boring - but required - `template` element and inside of *that*, include `product/_review.html.twig`. This template needs a `review` variable... so make sure to pass that in: `review` set to `entity`.

```
templates/broadcast/Review.stream.html.twig
// ... lines 1 - 19
{% block update %}
    <turbo-stream action="replace" target="product-review-{{ entity.id }}">
        <template>
            {{ include('product/_review.html.twig', {
                review: entity
            }) }}
        </template>
    </turbo-stream>
{% endblock %}
// ... lines 29 - 33
```

That's it! When a review is updated, it will *replace* this review element with the updated content.

Let's see it in action! Refresh the frontend. This is the review we'll update. Over in the admin area, add a very important dinosaur emoji... and save. Okay. I think that worked. Let me double check. Yep! Review updated.

Now check out the front end. That's amazing! This review just updated for *every* user in the world that is currently viewing a page where this is rendered. We could also update the quick stats area... but I'll leave that to you.

Removing a Review on Delete

What about *removing* a review? In the admin area, we can actually *delete* a review. Could we automatically *remove* this element from the frontend when that happens? Absolutely!

Inside the `remove` block, create a `<turbo-stream>`. This will have a new action - `action="remove"` - and will `target` the same element as our update. Now, you *might* expect me to say `entity.id`. But... by the time this template is rendered, the entity has *already* been deleted from the database. And so, `entity.id` is empty.

Fortunately, the library *also* passes us an `id` variable that we can use instead. Oh, and because we have `action="remove"`, the `turbo-stream` element won't have anything inside: it's just an instruction to find this element and *remove* it.

templates/broadcast/Review.stream.html.twig

```
// ... lines 1 - 29
{% block remove %}
    <turbo-stream action="remove" target="product-review-{{ id }}"></turbo-stream>
{% endblock %}
```

Ok: refresh the frontend just to be safe... and in the admin area, delete this. Now... deep breath... switch to the frontend. It's gone! Ok, this is getting fun.

Fading out on Remove

So let's get fancier. What if, when a review is deleted, instead of instantly disappearing, the element turned red, then faded out. OoooOOOoo.

Start in `styles/app.css`. Add a new `streamed-removed-item` class that sets the `background-color` to `coral`.

assets/styles/app.css

```
// ... lines 1 - 184
.streamed-removed-item {
    background-color: lightcoral;
}
// ... lines 188 - 191
```

Back in `Review.stream.html.twig`, this will be a bit trickier. We don't *actually* want to *remove* the element anymore... we want to *keep* it... but trigger some JavaScript that will fade it out.

To do this, change the action to `replace`... and then copy the entire `template` from `update`. But *this* time, pass in a new variable: `isRemoved` set to `true`. We can use that in the template to do something special.

```
templates/broadcast/Review.stream.html.twig
// ... lines 1 - 29
{% block remove %}
    <turbo-stream action="replace" target="product-review-{{ id }}">
        <template>
            {{ include('product/_review.html.twig', {
                review: entity,
                isRemoved: true
            }) }}
        </template>
    </turbo-stream>
{% endblock %}
```

Go open it up: `_review.html.twig`. If we pass in an `isNew` variable, we already have code to activate a Stimulus controller that causes the item to get a green background that fades out. We're going to do something similar.

If `isRemoved`, then initialize that *same* Stimulus controller. But this time pass `className` set to `streamed-removed-item`. *This* is why we made that controller dynamic. Also pass in *another* value called `removeElement` set to `true`.

```
templates/product/_review.html.twig
<div
    id="product-review-{{ review.id }}"
    class="component-light my-3 p-3"
    {% if isNew|default(false) %}
        {{ stimulus_controller('streamed-item', {
            className: 'streamed-new-item'
        }) }}
    {% endif %}
    {% if isRemoved|default(false) %}
        {{ stimulus_controller('streamed-item', {
            className: 'streamed-removed-item',
            removeElement: true
        }) }}
    {% endif %}
>
    <p><i class="fas fa-user-circle me-2"></i>{{ review.owner.email }} <i class="fas fa-star ms-4"></i> {{ review.stars }}/5</p>
    <div>
        {{ review.content }}
    </div>
</div>
```

This will signal to the controller that we want to fade out the element entirely.

Let's get to work in that file: `streamed-item_controller.js`.

Start by setting up the `removeElement` value, which will be a `Boolean`.

Then, import a helper function called `addFadeTransition`. This is a utility that we created in the first tutorial to help us fade in or fade out an element.

To activate it, inside `connect()`, call `addFadeTransition()` and pass it `this` object, `this.element` - the element that we're going to fade - and also an options object with `transitioned` set to `true`. That's needed because our element will *start* visible and then we want it to fade *out*. If you want to know more about how this all works, check out our Stimulus tutorial.

Inside `setTimeout()`, check to see if `this.removeElementValue` is `true`. If it is *not*, then keep the original code: this is where we fade out the background color. But if it *is* true, call `this.leave()`. That will trigger the entire element to fade out.

assets/controllers/streamed-item_controller.js

```
import { Controller } from 'stimulus';
import { addFadeTransition } from '../util/add-transition';

export default class extends Controller {
    static values = {
        className: String,
        removeElement: Boolean
    }

    connect() {
        addFadeTransition(this, this.element, {
            transitioned: true
        });

        this.element.classList.add(this.classNameValue);
        setTimeout(() => {
            if (this.removeElementValue) {
                this.leave();
            } else {
                this.element.classList.add('fade-background');
                this.element.classList.remove(this.classNameValue);
            }
        }, 5000);
    }
}
```

Phew! Let's see this thing in action! Go back and find this review... here it is. Refresh the frontend to get the fresh CSS... delete the review... and go to the frontend! Yes! It's there but with a red background! And then... woohoo! It faded out!

The big takeaway here? By combining Turbo Streams with Stimulus, you can do *much* more than simply "update the HTML of an element". You can do... anything.

Okay team: there's just *one* more thing that I want to try: using Turbo Streams to pop up "toast" notifications on the frontend, like after we do something awesome. That's next.

Chapter 57: Toast Notifications

We've made it to the *last* topic of the tutorial... so let's do something fun, like making it *super* easy to open "toast" notifications.

Toast notifications are those little messages that "pop up" like toast on the bottom - or top - of your screen. And Bootstrap has support for them. Our goal is simple but bold! I want to be able to trigger a toast notification from *any* template or from a Turbo Stream.

Creating the toast.html.twig Template

Start by creating a new template partial: `_toast.html.twig`. I'll paste in a structure that's from Bootstrap's documentation. Then let's make a few parts of this dynamic like `{{ title }}` - that's a variable we'll pass in... `{{ when }}` that defaults to `just now` and... for the body, `{{ body }}`.

templates/_toast.html.twig

```
<div class="toast" role="alert" aria-live="assertive" aria-atomic="true">
    <div class="toast-header">
        <svg class="rounded me-2" width="20" height="20"
xmlns="http://www.w3.org/2000/svg" aria-hidden="true"
preserveAspectRatio="xMidYMid slice" focusable="false"><rect width="100%"
height="100%" fill="#007aff"></rect></svg>
        <strong class="me-auto">Bootstrap</strong>
        <small>11 mins ago</small>
        <button type="button" class="btn-close" data-bs-dismiss="toast"
                aria-label="Close"></button>
    </div>
    <div class="toast-body">
        Hello, world! This is a toast message.
    </div>
</div>
```

Next, open up `product/_reviews.html.twig`. After submitting a new review, we render a flash message. *Now* I want this to be a toast notification! Cool! Include *that* template instead... and pass in a couple of variables like `title` set to `Success` and `body` set to the actual flash message content.

templates/product/_reviews.html.twig

```
// ... lines 1 - 6
<turbo-frame id="product-reviews-form">
// ... line 8
{% for flash in app.flashes('review_success') %}
    {{ include('_toast.html.twig', {
        title: 'Success!',
        body: flash
    }) }}
{% endfor %}
// ... lines 15 - 36
</turbo-frame>
```

The Toast Stimulus Controller

If we stopped now... congratulations! Absolutely nothing would happen. These toast elements are *invisible* until you execute some JavaScript that opens them on the page. To do *that*, we need a Stimulus controller!

Up in the `assets/controllers/` directory, create a new file called, how about, `toast_controller.js`. Inside, give this the normal structure where we import `Controller` from `stimulus`, export *our* controller... and have a `connect()` method that, of course, logs a loaf of bread.

assets/controllers/toast_controller.js

```
import { Controller } from 'stimulus';

export default class extends Controller {
    connect() {
        console.log('?');
    }
}
```

Over in `_toast.html.twig`, I want to activate this controller *whenever* this toast element appears on the page. No problemo: on the outer element, add `{{ stimulus_controller('toast') }}`.

templates/_toast.html.twig

```
<div class="toast" role="alert" aria-live="assertive" aria-atomic="true"
{{ stimulus_controller('toast') }}>
// ... lines 2 - 11
</div>
```

Our controller doesn't do anything yet, but let's at *least* make sure that it's connected. Head over to our site, refresh the page... make sure that your console is open... and then go fill out a new review. When we submit... got it! As soon as the toast HTML was rendered onto the page, our controller was initialized. Though... like I mentioned, you can't actually *see* the toast element yet. It's taking up some space... but it's invisible.

Let's fix that! Back in the controller, import `Toast` from `bootstrap`. Below add `const toast = new Toast()` and pass it `this.element`. To *open* the toast, say `toast.show()`.

```
assets/controllers/toast_controller.js
```

```js
import { Controller } from 'stimulus';
import { Toast } from 'bootstrap';

export default class extends Controller {
    connect() {
        const toast = new Toast(this.element);
        toast.show();
    }
}
```

That's it! Refresh again and add another review. This time... that's super cool! And it means that we can, from *anywhere*, render the `_toast.html.twig` template and it will activate this behavior.

Grouping all the Toasts into One Container

Though... the positioning isn't what I was imagining. Before it disappeared, it was open... right in the middle of the page. I was hoping to put it in the top right corner of the screen.

To do that, we just need to add a few classes to the toast element. Except... there's one other minor problem. If you think about it, it's possible that a user could see *multiple* toast notifications at the same time. The toast system *totally* supports this.... it stacks them on top of each other. But for that to work, we need to have a single global "toast container" element on our page that all individual toasts live *inside* of.

This might be easier to show. Open up `templates/base.html.twig`. Really, anywhere, but I'll go to the bottom, add a `<div>` with `id="toast-container`. That could be anything: we'll use this `id` to find this element in JavaScript.

Also add `class="toast-container"` and a few other classes. `toast-container` helps Bootstrap *stack* any toasts inside of this... and everything else puts the toast in the upper right corner of the screen.

templates/base.html.twig

```
<!DOCTYPE html>
<html lang="en-US">
// ... lines 3 - 14
    <body>
// ... lines 16 - 101
        <div
            id="toast-container"
            class="toast-container position-fixed top-0 end-0 p-3"
        ></div>
    </body>
</html>
```

Now, in order for this to work, we need all the toast notifications to physically live *inside* of this `toast-container` element. So basically, we need to render `_toast.html.twig`... and somehow get that HTML *inside* of the container.

But... I don't want to do that! I want to keep the flexibility of being able to render `_toast.html.twig` from... *wherever* and have it work. And we can *still* have this with a little help from our Stimulus controller.

Check it out: at the top of `connect()`, add `const toastContainer = document.getElementById()` and pass it `toast-container` to find the element that lives at the bottom of the page. Then... let's move *ourselves into* that: `toastContainer.appendChild(this.element)`.

And now that it lives inside the container, we open it like normal!

Though... there is one subtle "catch". When the toast HTML initially loads, it will live here in the middle of the page. Naturally, Stimulus *notices* this element, instantiates a new controller instance and calls `connect()`. Yay! But when we move `this.element` into `toast-container`, Stimulus destroys the original controller instance, creates a new one, and calls `connect()` a *second* time.

In other words, the `connect()` method will be called twice: once when we originally render our toast element onto the page and again after we move into `toast-container`. Right now, that's going to cause an infinite loop where we call `appendChild()` over and over again.

To avoid that, add, if `this.element.parentNode` does not equal `toastContainer`. So only if the element has *not* been moved yet, move it... and then return. The first time this executes, it will move the element and exit. The second time it executes, it will skip all of this and pop open the toast.

```
assets/controllers/toast_controller.js
```

```javascript
import { Controller } from 'stimulus';
import { Toast } from 'bootstrap';

export default class extends Controller {
    connect() {
        const toastContainer = document.getElementById('toast-container');
        if (this.element.parentNode !== toastContainer) {
            toastContainer.appendChild(this.element);

            return;
        }

        const toast = new Toast(this.element);
        toast.show();
    }
}
```

Let's try this thing! Refresh the page, add another review and... beautiful! If you quickly inspect the toast element... yup! It lives down inside of `toast-container`.

Publishing a Toast through Mercure to All Users

Ok, I have one last micro-challenge: whenever a new review is added to a product, I want to open a toast notification on *every* user's screen that's currently viewing the product. Something that says:

> *“Hey! This product has a new review!”*

Over in `Review.stream.html.twig`, in the `create` block, add another turbo stream with `action="append"` and `target=""`... well... leave that empty for a minute. Give this the `template` element, include `_toast.html.twig` and pass in a few variables: `title` set to `New Review` and `body` set to

> *“A new review was just posted for this product.”*

```
templates/broadcast/Review.stream.html.twig
{% block create %}
// ... lines 2 - 18
    <turbo-stream action="append" target="product-{{ entity.product.id }}-toasts">
        <template>
            {{ include('_toast.html.twig', {
                title: 'New Review!',
                body: 'A new review was just posted for this product'
            }) }}
        </template>
    </turbo-stream>
{% endblock %}
// ... lines 28 - 49
```

Very nice! But... what should the `target` be? We could use `toast-container`. That would append it to this element. But... then the message would show up on *every* page. We only want this message to show up if you're viewing *this* specific product.

To do that, we need to target an element that *only* exists on *this* specific product's page. Open `show.html.twig`. Right inside of the `product_body` block, let's add an empty `div` with `id="product-{{ product.id }}-toasts"`:

```
templates/product/show.html.twig
{% extends 'product/productBase.html.twig' %}
// ... line 2
{% block productBody %}
    <div id="product-{{ product.id }}-toasts"></div>
// ... lines 5 - 49
{% endblock %}
```

A little empty element *just* for our toasts to go into. Copy this and, in `Review.stream.html.twig`, target it. Except that we need `entity.product.id`.

Let's check it out! Refresh the page... and then open the same product in another tab to "mimic" what a *different* user would see. Scroll down, fill in a review and... submit. Awesome! We have two toasts over here and... the other user sees the *one* toast! The *two* toast notifications in our first tab *is* a bit weird, but I'll leave it for now.

And... we're done! Woh! Congrats to you! You deserve a nice crisp high five... and maybe a short vacation for making it through this *huge* tutorial. It was huge because... well... Turbo has a lot to offer. I hope you're as excited about the possibilities of Stimulus and Turbo as I am.

Let us know what you're building. And, as always, if you have any questions, we're here for you down in the comments section.

All right, friends. See you next time!

With <3 from SymfonyCasts