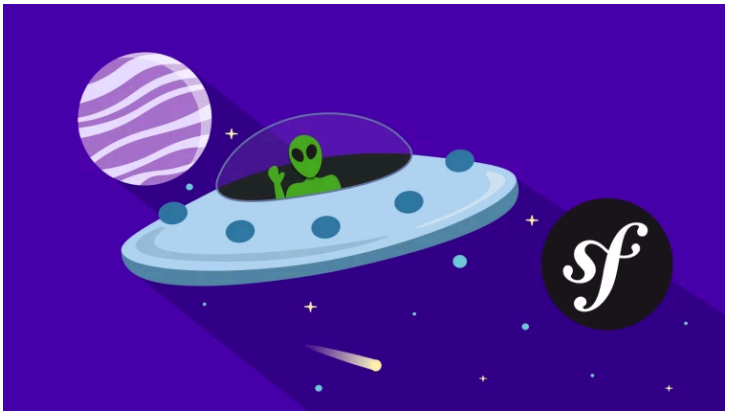
30 Days with LAST Stack



Chapter 1: Hello LAST Stack!

Hey everyone! Welcome to 30 days of LAST Stack! I gotta say, this might be my favorite tutorial ever. I had an absolute blast building this, because unlike usual around here, I'm going to go a little bit less into teaching deep concepts and instead focus on making a rich, polished, beautiful product. And I think you're going to love it.

But first, LAST Stack, what the heck is that? It's an acronym that... I made up. I wanted something fun to match a whole new paradigm. It stands for Live Components, AssetMapper, Stimulus, and Turbo. It's a front-end stack that'll let us build a *truly* rich user interface - like a single-page application, with modals and AJAX everywhere - but entirely with Symfony, Twig... and just a bit of JavaScript. Oh, and this will require *no* build step and no Node.js. Woo!

By the end of this tutorial, we're going to have reusable patterns that we can leverage in on our projects to get things done really quickly but that *work* and *feel* incredible.

At the core of this whole system is Hotwire: a collection of libraries that include Turbo, Stimulus and Strada. Strada is the new kid on the block and it looks *cool*. We won't have time to talk about it, but it promises to let you take the same project that we're about to build and use it to power a mobile app. Woh.

One other cool things about Hotwire is that... it's *not* unique to Symfony. It's used, for example, by the Ruby on Rails community. And many of the things that we're going to build come from patterns I learned from people in that community. The fact that we're all using the same tool means we get to share libraries, share ideas and build on top of each other's shoulders. That's massive.

Project Setup

So let's get into this! Because it's fun to make pretty things that pop onto the screen, you should absolutely download the course code and code along with me. When you unzip the file, you'll find a `start/` directory, which has the same files that you see here, including the all-important README.md! This tells you all about how to get the project set up.

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

```
symfony serve -d
```

To start a local web server at ... oh, in my case, `127.0.0.1:8001`. I must already have something running on port 8000. I'll click the link to see a big, ugly page of... nothing! That's on purpose!

What we're starting with is a Symfony 6.4 project. I've pre-installed Twig and we have two different entities - `Planet` and `Voyage` - because we're going to build a trip-planning site for aliens. I also have some data fixtures and I used MakerBundle to generate a CRUD for each entity. This `PlanetController`, `VoyageController` and these templates come from MakerBundle, with just a few styling adjustments.

But basically... there's nothing special going on! We do have a `MainController`, which powers this homepage:

src/Controller/MainController.php

```
// ... lines 1 - 12
class MainController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(
        VoyageRepository $voyageRepository,
        PlanetRepository $planetRepository,
        #[MapQueryParameter('query')] string $query = null,
        #[MapQueryParameter('planets', \FILTER_VALIDATE_INT)] array $searchPlanets = [],
    ): Response
    {
        $voyages = $voyageRepository->findBySearch($query, $searchPlanets);

        return $this->render('main/homepage.html.twig', [
            'voyages' => $voyages,
            'planets' => $planetRepository->findAll(),
            'searchPlanets' => $searchPlanets,
        ]);
    }
}
```

It contains a query that will help us later... but the template, right now, is doing a whole lot of nothing:

templates/main/homepage.html.twig

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

{% block title %}Space Inviters!{% endblock %}

{% block body %}
    <h1>Space Inviters: Plan your voyage and come in peace!</h1>
{% endblock %}
```

No CSS, no JavaScript, no assets of any kind... and the site doesn't *do* anything. But in 30 short lessons, we'll transform this into a small digital masterpiece.

That's it for day 1. Tomorrow, we'll install AssetMapper: a system for handling CSS, JavaScript and other frontend assets with batteries include... but absolutely *no* build step.

Chapter 2: Asset Mapper

Okay, so how are we going to bring CSS and JavaScript into our app? Are we going to add a build system like Vite or Webpack? Heck no! That's one of the fun things about all of this! We're going to create something *amazing* with *zero* build system. To do that, let's install a new Symfony component called AssetMapper.

Installing AssetMapper

Spin over to our terminal and run:

```
composer require symfony/asset-mapper
```

This is the new alternative to Webpack Encore. It can do pretty much everything that Encore can do and more... but it's *way* simpler. You should definitely use it on new projects.

When I run:

```
git status
```

We see that its Flex recipe made a number of changes. For example, `.gitignore` is ignoring a `public/assets/` directory and `assets/vendor/`:

.gitignore

```
// ... lines 1 - 11
###> symfony/asset-mapper ###
/public/assets/
/assets/vendor
###
```

We'll talk more about those later. But on production, this is where your assets will be written to and, when we install third-party JavaScript libraries, they'll live in that `vendor/` directory.

It also updated `base.html.twig` and added an `importmap.php` file. But put those on the back burner for now: we'll talk about them tomorrow.

The "Mapped Paths"

For today's adventure, pretend that, when we installed this, all it gave us was a new `asset_mapper.yaml` file and an `assets/` directory. Let's go check out that config file: `config/packages/asset_mapper.yaml`:

config/packages/asset_mapper.yaml

```yaml
framework:
    asset_mapper:
        # The paths to make available to the asset mapper.
        paths:
            - assets/
```

The idea behind AssetMapper couldn't be simpler: you define paths - like the `assets/` directory - and AssetMapper makes every file inside available publicly... as *if* they lived in the `public/` directory.

Referencing an Asset File

Let's see it in action you. If you downloaded the course code, you should have a `tutorial/` directory, which I added so we can copy a few things out of it. Copy `logo.png`. Inside `assets/`, we can make this look however we want. So let's create a new directory called `images/` and paste that in.

Since this new files lives inside the `assets/` directory, we should be able to reference it publicly. Let's do that in our base layout: `templates/base.html.twig`. Anywhere, say `<img src="">`, `{{` and then use the normal `asset()` function. For the argument, pass the path *relative* to the `assets/` directory. This is called the logical path: `images/logo.png`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 14
    <body>
        <img src="{{ asset('images/logo.png') }}" alt="Space Inviters Logo" />
// ... lines 17 - 18
    </body>
</html>
```

Before we try this, an easy way to see *every* asset that's available is via:

```
php bin/console debug:asset
```

Very simply: this looks through all of your mapped paths - just `assets/` for us - finds every file then lists them with their logical path. So I can be lazy and copy that, paste it here.... and done.

Now, when we try this, it doesn't work! The `asset()` function is still its *own* component, so let's get that installed:

```
composer require symfony/asset
```

And now.... cool logo!

Instant Asset Versioning

To see the *really* neat thing, inspect the image and look at the filename. It's `/assets/images/logo-` and then this long hash. This hash comes from the file's *contents*. If we updated `logo.png`, it would automatically generate a new hash. And that is *super* important for two, related, reasons. First, because when we deploy, the new filename will bust the browser cache for our users so that they see the new file immediately. And second, because of this, we can configure our production web server to serve all the assets with long-lived Expiration headers. That *maximizes* that caching & performance.

Serving Assets in Dev vs Prod

Now in the `dev` environment, there is *no* physical file with this filename. Instead, the request for this asset is processed through Symfony and intercepted by a core listener. That listener looks at the URL, finds the matching `logo.png` inside the `assets/images/` directory and returns it.

But on production, that's not fast *enough*. So, when you deploy, you'll run:

```
php bin/console asset-map:compile
```

Very simply: this writes all the files into the `public/assets/` directory. Look: in `public/assets/`, we now have real, physical files! So when I go over and refresh, this file isn't being processed by Symfony, it's loading one of those real files.

Now, if you ever run this command locally, make sure to delete that directory after... so it stops using the compiled versions:

```
rm -rf public/assets/
```

Wow! Day 2 is already done! We now have a way to serve images, CSS or *any* file publicly with automatic file versioning. The second part of `AssetMapper` is all about JavaScript modules. And that's tomorrow's topic.

Chapter 3: JavaScript Modules

Inspect element on this page and head over to the browser console. Ah, we've got a console log that says it comes from `assets/app.js`. And sure enough, if we spin over and open that file... there it is!

assets/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css'

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉')
```

But how is this file being loaded? To answer that, view the page source. There's some interesting stuff going on here, but I want to zoom in on one part: `<script type="module">`, `import 'app';`.

ECMAScript Modules

It turns out that all modern browsers - basically everything except for IE 11... and you should *not* be supporting IE 11 anymore - ahem all modern browsers support JavaScript modules, also known as ECMAScript modules or ESM. But they're nothing fancy: a JavaScript module is any JavaScript file that uses the `import` or `export` statements that you probably grew accustomed to in Webpack Encore.

The big news is that: browsers understand `import` and `export` all by themselves! No build step needed. If you open any HTML page and say `<script type="module">`, the code inside is allowed to use `import` and `export` statements.

Importmaps

So... the second question is: what the heck is `app`? How does `app` ultimately refer to `assets/app.js`? This is *also* a new trick of browsers called *importmaps*. And this has *nothing* to do with Symfony or AssetMapper. If, on your page, you have a `<script type="importmap">`, this becomes a key value map that's used by your browser when it loads modules. So if we say `import 'app'`, it looks inside of this list, sees `app` and ultimately loads *this* file... which is served by AssetMapper. It's a nice bit of teamwork!

Importmaps are supported by all modern browsers... though it has slightly less support than JavaScript modules. Fortunately, there's a shim or polyfill so that if your user *happens* to use a browser that *doesn't* support importmaps, that shim will *add* it and everything will work.

The importmap() Function

The final question on my mind is: where the heck is this all coming from? To answer that, open `templates/base.html.twig`. It's entirely coming from this one line right here: `{{ importmap('app') }}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 10
        {% block javascripts %}
            {{ importmap('app') }}
        {% endblock %}
    </head>
// ... lines 15 - 19
</html>
```

Because we passed `app`, this will generate a `<script type="module">` with `import 'app'` inside. But this also dumps the polyfill, some preloads - those are good for performance, but not required - and, of course, the importmap itself. The importmap is *primarily*, though not entirely (we'll get to that), generated from this `importmap.php` file:

importmap.php

```
// ... lines 1 - 2
/**
 * Returns the import map for this application.
 *
 * - "path" is a path inside the asset mapper system. Use the
 *     "debug:asset-map" command to see the full list of paths.
 *
 *  - "entrypoint" (JavaScript only) set to true for any module that will
 *      be used as an "entrypoint" (and passed to the importmap() Twig
function).
 *
 * The "importmap:require" command can be used to add new entries to this
file.
 *
 * This file has been auto-generated by the importmap commands.
 */
return [
    'app' => [
        'path' => './assets/app.js',
        'entrypoint' => true,
    ],
];
```

The importmap.php File

When we installed AssetMapper, its recipe gave us this file. And *this* is the reason that the `importmap` in our HTML has an `app` key that points to `assets/app.js`.

Writing Some JavaScript Modules

So I want to play a bit with this new system. Inside the `assets/` directory - we can organize this however we want - create a `lib/` directory with an `alien-greeting.js` file. Inside, I'm going to write some awesome, modern JavaScript: `export default` a function, give it `message` and `inPeace` arguments... then I'll log a message using a template literal - the fancy backticks - and some emojis:

assets/lib/alien-greeting.js

```
export default function (message, inPeace = false) {
    console.log(`${message}! ${inPeace ? '👽' : '👾'}`);
}
```

Cool! This new file lives inside `assets/` so, technically, it's publicly available. But... nobody is *using* it yet.

Let's try something non-traditional, but fun to start. Go into the base layout and, anywhere, say `<script type="module">`. Inside, `import alienGreeting`... and I'll hit tab:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 10
        {% block javascripts %}
            {{ importmap('app') }}

            <script type="module">
                import alienGreeting from '{{ asset('lib/alien-greeting.js') }}';
// ... lines 16 - 17
            </script>
        {% endblock %}
    </head>
// ... lines 21 - 25
</html>
```

Hmm: PhpStorm used `../assets` for the path. That's not going to work. Instead, we can use the `asset()` function and the logical path: `lib/alien-greeting.js`. Then below, use that: `alienGreeting()`, a message and we will *not* come in peace!

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 10
        {% block javascripts %}
// ... lines 12 - 13
            <script type="module">
                import alienGreeting from '{{ asset('lib/alien-greeting.js') }}';

                alienGreeting('Give us all your candy!', false);
            </script>
        {% endblock %}
    </head>
// ... lines 21 - 25
</html>
```

Let's see if it works! Close that, and... it *doesn't*? I actually thought it *would*! We get a 404 for `lib/alien-greeing.js` - with no "t"...! Boop!

templates/base.html.twig

```
↕ // ... lines 1 - 14
15                 import alienGreeting from '{{ asset('lib/alien-
greeting.js') }}';
↕ // ... lines 16 - 27
```

Now it works! No build, nice code, nothing special.

If you view the page source, we, of course, have this nice versioned filename in the `import`. So you can import simple things like `app` and rely on the `importmap` to point to the true filename, or you can include full paths.

Importing from JS Files

As fun as it was to hack this into the HTML, in reality, we're not usually going to write in-line code like this. Copy this, get rid of the `<script type="module">`:

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
3      <head>
↕  // ... lines 4 - 10
11         {% block javascripts %}
12             {{ importmap('app') }}
13         {% endblock %}
14     </head>
↕  // ... lines 15 - 19
20 </html>
```

Then go into `app.js`. Paste the code here:

assets/app.js

```
↕ // ... lines 1 - 6
7  import './styles/app.css'
8  import alienGreeting from './lib/alien-greeting.js';
9
10 alienGreeting('Give us all your candy!', false);
```

And now that we're inside JavaScript, when we refer to a path, we can write it with normal, relative paths: `./alien-greeting.js`:

assets/app.js

```
// ... lines 1 - 7
import alienGreeting from './lib/alien-greeting.js';
// ... lines 9 - 11
```

This is the exact code that we would have in Webpack Encore, with one small difference. In Webpack, you don't need to have the `.js` on the end. It turns out that leaving *off* the extension is a Node-specific thing. In real JavaScript, you *do* need to have the extension. So you *do* need to add the `.js`.

And... it works!

PhpStorm: Auto-add Extension

By the way, if you let PhpStorm auto-complete the path to the imported JavaScript file, by default, it will *not* include the `.js` on the end. To fix that, open the settings... and search for "extensions". There we go: "Editor"=>"Code Style"=> "JavaScript". Right here, change this "use file extension" to "always".

Ok, day 3 is in the books! Tomorrow, we'll make our JavaScript set up much more powerful by learning how to install 3rd-party packages!

Chapter 4: 3rd Party JavaScript Packages

Welcome to the fabulous day 4! Where we're already creating JavaScript modules... a fancy term that means we're writing import statements and export statements. And we're pulling this off entirely without a build system. Time for a happy dance!

But what about third-party packages? Head over to https://npmjs.com and search for a very important package called `js-confetti`. This package is all about celebrating, which... is exactly what we're doing during these 30 days! In the README, it says to use Yarn to install it. We are *not* going to do that. Instead, skip right down to the usage example. Copy that, head over to our `app.js`... and paste that in:

assets/app.js

```
↕ // ... lines 1 - 8
9  import JSConfetti from 'js-confetti';
10
11 const jsConfetti = new JSConfetti();
12 jsConfetti.addConfetti();
↕ // ... lines 13 - 15
```

Side note: `import` statements always go at the *top* of your file. If you *don't* do that - if you do something weird like this, well, you *can*, but your browser will move this up to the top when it executes the code anyway. So we'll avoid being troublemakers.

Missing JavaScript Module Error

Ok: is this going to work? I mean... probably *not* because we haven't *installed* anything. But let's live recklessly and try it anyway! Error! A very important error:

> *"Failed to resolve module specifier `js-confetti`. Relative references must start with either `/`, `./` or `../`."*

So what this is saying is that your browser found an `import` statement... and has no idea how to load that file. If an import statement starts with `./` or `../`, your browser knows how to handle that: it looks for a file *relative* to this file. Easy peasy.

But if there is *no* `./` or `../`, it's called a bare module. In that case, your browser looks for a match in the importmap. Right now, our importmap looks like it did before. Notably, we do *not* have a `js-confetti` key. And *that's* why we get this error.

This is one of the *most* important errors you'll see when coding with modules. And it'll look a bit different based on which browser you're using. Firefox, for example, phrases this error differently.

But regardless of the wording, this error almost always means the same thing: you're trying to use a third party package, but it's not *installed*.

Installing Packages with `importmap:require`

How *do* we install it? Glad you asked! Copy the package name, spin over and run:

```
php bin/console importmap:require js-confetti
```

That's it! Spin *back* over and... celebration! It works! Mad refreshing!

How does that work? Karma? Well, not surprisingly, if you view the page source, we have a new entry inside our `importmap` with a `js-confetti` key. And it points to a file in an `assets/vendor/` directory. Interesting.

When we ran that command, it really did just one thing. It updated our `importmap.php` file. It added this entry right here:

importmap.php

```
↕ // ... lines 1 - 15
16 return [
↕ // ... lines 17 - 20
21     'js-confetti' => [
22         'version' => '0.11.0',
23     ],
24 ];
```

Behind the scenes, it went out and found what the latest version was and put that here. And because we have a `js-confetti` item in `importmap.php`, it means that we're going to have a matching `js-confetti` key inside of the importmap on the page.

The assets/vendor/ Directory

Where does that file actually live? Up here in a new `assets/vendor/` directory. If you dig, here is the actual file that's being loaded.

Two juicy details about this `vendor/` directory. The first is: it's ignored from Git: you can see `/assets/vendor/`:

```
.gitignore
↕ // ... lines 1 - 11
12 ###> symfony/asset-mapper ###
↕ // ... line 13
14 /assets/vendor
15 ###
```

Just like the composer `vendor/` directory, this is *not* something that you should commit to your repository.

The second is more of a question: how do I get these files if I clone or update a project and some or all of the files are missing?

To find out, get crazy and destroy that directory. Muwahahaha. And now, to increase our reckless streak, try to refresh the page. Error! Awesome error!

> *"The `js-confetti` vendor asset is missing: try running the `importmap:install` command."*

Lovely idea! Spin over and try that:

```
php bin/console importmap:install
```

Just like `composer install`, that downloads whatever you need into `assets/vendor/`... and now it just works.

That's it! By day 4, we've already solved 3rd party packages! We don't even need a giant `node_modules/` directory! I'm going to have to find some other way to fill my hard drive. Maybe, more Docker containers?

Ok, for tomorrow's adventure, we'll jazz up our site with some CSS. Stay tuned!

Chapter 5: CSS

Day 5 already? We're flying! It's time to add some CSS to our site. So how does that work inside AssetMapper?

Including a Manual link Tag?

Well, we already have an `assets/styles/app.css` file. And... there's nothing stopping us from going into `base.html.twig`, and adding a link tag: `link`, `rel="stylesheet"`, `href` then `asset()` and the logical path: `styles/app.css`.

Swell! When we refresh... and look at the page source, there it is! It works great and it's super boring. The kind of boring I like.

However, if we remove this line... and go and refresh the page. Huh, we *still* have this `blue` background: a blue background that's coming from `app.css`:

assets/styles/app.css

```
1 body {
2     background-color: skyblue;
3 }
```

Take another peek at the page source. There is *still* a `link` tag pointing to that file? Back over in `base.html.twig`, hmm, nothing here. Where is that coming from?

The answer - I bet you guessed - is the `importmap()` function:

templates/base.html.twig

```
↕ // ... line 1
2 <html>
3     <head>
↕ // ... lines 4 - 10
11         {% block javascripts %}
12             {{ importmap('app') }}
13         {% endblock %}
14     </head>
↕ // ... lines 15 - 19
20 </html>
```

And it's because it's being imported from `app.js`:

assets/app.js

```
↕ // ... lines 1 - 6
7 import './styles/app.css';
↕ // ... lines 8 - 15
```

How CSS Works

Importing a CSS file from JavaScript is probably something you got used to with Webpack Encore. You import a CSS file... and ultimately, it's rendered on the page as a `link` tag. *However*, this is *not* something that ECMAScript modules actually support. The *only* thing you can import are *JavaScript* files. So this *should* fail spectacularly: like it should download the CSS file and try to parse it as JavaScript.

However, as you may have noticed, it doesn't fail! I love mysteries!

This is a *totally* extra feature that we added to AssetMapper. Here's how it works. In `base.html.twig`, we say `importmap('app')`. The `app` is known as the entrypoint: the *one* file the browser will execute directly. And we know that refers to `assets/app.js`.

So what AssetMapper does is, it goes into this file and finds all the `import` statements for JavaScript and CSS files. For every CSS import it finds, it adds that as a `link` tag. It's... basically just that simple.

The CSS Importmap Trick

Well, there *is* one little, fascinating complication. Go to the network tab in your browser and search for `app`. This is the `app.js` file that's being executed by the browser. Notice: it *does* still have the import statement to the CSS file! If you think about it, when our browser executes this line, it should fail! It should download the CSS file, try to parse it as JavaScript & hit a syntax error. But it doesn't.

The reason is a trick inside AssetMapper. When you import a CSS file, AssetMapper adds an importmap entry for it. So even though this starts with `./`, our browser *does* look to see if there's a matching path inside the importmap. And there *is*. Because of that, it downloads this file.... which is *not* a real file. It's a fake file that does.... absolutely nothing. So it makes that line not error out and... not *do* anything.

Importing CSS from Other JavaScript Files

To see how powerful this is, let's create a second CSS file to support our alien greeting. Call it `alien-greeting.css` and make the body background `darkgreen`. Though, personally, I'm hoping aliens are rainbow colored:

assets/styles/alien-greeting.css

```
1 body {
2     background: darkgreen;
3 }
```

Over in `alien-greeting.js`, import that: `../styles/alien-greeting.css`:

assets/lib/alien-greeting.js

```
1 import '../styles/alien-greeting.css';
↕ // ... lines 2 - 6
```

Will this work? Try it! Refresh and... green background! In the source, we have a second `link` tag and a second new item in the `importmap`. So that's awesome! Because `app.js` imports `alien-greeting.js`, AssetMapper *also* finds any CSS files that *it* imports.

Lazy-Loading CSS

Here's where things get *really* spooky-cool. JavaScript modules have a dynamic import syntax that allows you to import modules asynchronously. That lets us load a file *later* when we need it, instead of on page load. And we can use this trick with CSS.

Copy this. Pretend that we only want to load that CSS file when `inPeace` equals false. So I'll say, if not `inPeace`, then use `setTimeout()` to wait for 4 seconds. After 4 seconds, import the CSS file. Except, as soon as you need an import to *not* live at the top of your file, you need to call it like a function:

assets/lib/alien-greeting.js

```
1  export default function (message, inPeace = false) {
2      if (!inPeace) {
3          setTimeout(() => {
4              import('../styles/alien-greeting.css');
5          }, 4000);
6      }
↕  // ... lines 7 - 8
9  }
```

That's pretty cool. Try it. At first, blue background! 2, 3, 4, green background! The CSS file loaded *lazily*. How? Well, there's no `alien-greeting.css` link tag in the page source anymore. Instead, we wait for the browser to execute this JavaScript line. When it does, it looks for this in the importmap, finds it and downloads this fake file. But this time, instead of it being a line that does *nothing*, this fake file adds a new `link` tag to the `head` section with `rel="stylesheet"` and `href` set to `alien-greeting.css`.

Heck, we can watch this in real time! Over here, under the `head` tag, we see the stylesheet. If I refresh and quickly open that, it's not there. And... *then* it gets added. *So* stinkin' cool.

Now that we've conquered *how* CSS works, tomorrow, we'll use it to bring our site to life! But I want to do it with an extra fun angle: I want to use Tailwind CSS.

Chapter 6: Tailwind CSS

I love using Tailwind for CSS. If you've never used it before, or maybe only heard of it, you might... hate it at first. That's because you use classes inside of HTML to define *everything*. And so your HTML can end up looking, well, a bit crazy. But give it a chance. I've absolutely fallen in love with it. And, instead of this looking ugly to me, it looks descriptive.

Tailwind Requires Building!

Tailwind isn't your traditional CSS behemoth where you download a giant CSS file and include it. Instead, Tailwind has a binary that parses all of your templates, finds the classes you're using, and then dumps a final CSS containing *just* those classes. So it keeps your final CSS as small as possible.

But to do this, duh duh duh! Tailwind requires a *build* step. And that's okay. Just because we don't have a build step for our *entire* JavaScript system doesn't mean we can't opt *into* a small one for a specific purpose.

Installing symfonycasts/tailwind-bundle

And there's a super-easy bundle to help us do this with AssetMapper. It's called `symfonycasts/tailwind-bundle`. Hey, I've heard of them!

Head down here, go to the documentation... and I'll copy the `composer require` line. Spin over and run that:

```
composer require symfonycasts/tailwind-bundle
```

This bundle will help us run the build command in the background and serve up the final file. We point it at a real CSS file, then it'll sneak in the dynamic content. You'll see.

While we're here, run:

```
php bin/console debug:config symfonycasts_tailwind
```

to see the default configuration for the bundle. By default, the file that it "builds" is `assets/styles/app.css`... which is *great* because we already have an `assets/styles/app.css` file!

To get things set up, run a command from the bundle:

```
php bin/console tailwind:init
```

Tip

If using the Symfony CLI, you can add a build command as a worker to be started whenever you run the Symfony web server:

```
# .symfony.local.yaml
workers:
    tailwind:
        cmd: ['symfony', 'console', 'tailwind:build', '--watch']
```

See the docs for more information.

This downloads the Tailwind binary in the background, which is awesome. That binary is standalone and doesn't require Node. It just works. The command also did two other things. First: it added the three lines needed inside of `app.css`:

assets/styles/app.css

```
1  @tailwind base;
2  @tailwind components;
3  @tailwind utilities;
↕  // ... lines 4 - 8
```

When this file is built, these will be replaced by the actual CSS we need. Second, this created a `tailwind.config.js` file:

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    "./assets/**/*.js",
    "./templates/**/*.html.twig",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

This tells Tailwind *where* to look for all the classes we'll use. This will find any classes in our JavaScript files or our templates.

To *execute* Tailwind, run:

```
php bin/console tailwind:build -w
```

For *watch*. That builds... then hangs out, waiting for future changes.

So... what did that do? Remember: we're already including `app.css` on our page. When we refresh, woh! It looks a bit different! The reason is, if you view the page source, and click to open the `app.css` file, it's full of Tailwind code! Our `app.css` file is no longer this exact source file! Behind the scenes, the Tailwind binary parses our templates, dumps a final version of this file, which is then returned. This file already contains a bunch of code because I filled the CRUD templates with Tailwind classes before we started the tutorial.

Using Tailwind

But let's see this in action for real. If we refresh the page, this is my `h1`. It's small and sad. Open `templates/main/homepage.html.twig`. On the `h1`, add `class="text-3xl"`:

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <h1 class="text-3xl">Space Inviters: Plan your voyage and come in peace!</h1>
{% endblock %}
```

Now, refresh. It works! If that `text-3xl` wasn't in the `app.css` file before, Tailwind just added it because it's running in the background.

Pasting in The Layout

So Tailwind is working! To celebrate, let's paste in a proper layout. If you downloaded the course code, you should have a `tutorial/` directory with a couple of files. Move `base.html.twig` into templates:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>{% block title %}Space Inviters!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size=%2296%22>⚫</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
            {{ importmap('app') }}
        {% endblock %}
    </head>
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
                    <div class="flex items-center">
                        <a href="{{ path('app_homepage') }}">
                            <img src="{{ asset('images/logo.png') }}" width="50" alt="Space Inviters Logo" >
                        </a>
                        <a href="{{ path('app_homepage') }}" class="text-xl ml-3">Space Inviters</a>
                        <a href="{{ path('app_voyage_index') }}" class="ml-6 hover:text-gray-400">Voyages</a>
                        <a href="{{ path('app_planet_index') }}" class="ml-4 hover:text-gray-400">Planets</a>
                    </div>
                    <div
                        class="hidden md:flex pr-10 items-center space-x-2 border-2 border-gray-900 rounded-lg p-2 bg-gray-800 text-white cursor-pointer"
                    >
                        <svg xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 text-gray-500" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M10 10m-7 0a7 7 0 1 0 14 0a7 7 0 1 0 -14 0"/><path d="M21 21l-6 -6"/></svg>
                        <span class="pl-2 pr-10 text-gray-500">Search Cmd+K</span>
                    </div>
                </nav>
```

```
            </header>

            <!-- Make sure the main tag takes up the remaining height -->
            <main class="flex-grow">{% block body %}{% endblock %}</main>

            <!-- Footer -->
            <footer class="py-4 mt-6 bg-gray-800 text-center">
                <div class="text-sm">
                    With <svg class="inline-block w-4 h-4 text-red-600 fill-current" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"><path d="M10 3.22l-.61-.6a5.5 5.5 0 00-7.78 7.78l7.39 7.4 7.39-7.4a5.5 5.5 0 00-7.78-7.78l-.61.61z"/></svg> from Symfonycasts.
                </div>
            </footer>
        </div>
    </body>
</html>
```

And these other two go into the `main/` directory:

templates/main/homepage.html.twig

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

{% block title %}Space Inviters!{% endblock %}

{% block body %}
    <div class="flex">
        <aside class="hidden md:block md:w-64 bg-gray-900 px-2 py-6">
            <h2 class="text-xl text-white font-semibold mb-6 px-2">Planets</h2>
            <div>
                {{ include('main/_planet_list.html.twig') }}
            </div>
        </aside>

        <section class="flex-1 ml-10">
            <form
                method="GET"
                action="{{ path('app_homepage') }}"
                class="mb-6 flex justify-between"
            >
                <input
                    type="search"
                    name="query"
                    value="{{ app.request.query.get('query') }}"
                    aria-label="Search voyages"
                    placeholder="Search voyages"
                    class="w-1/3 px-4 py-2 rounded bg-gray-800 text-white placeholder-gray-400"
                >
                <div class="whitespace-nowrap m-2 mr-4">{{ voyages|length }} results</div>
            </form>
            <div class="bg-gray-800 p-4 rounded">
                <table class="w-full text-white">
                    <thead>
                        <tr>
                            <th class="text-left py-2">Purpose</th>
                            <th class="text-left py-2 pr-4">Planet</th>
                            <th class="text-left py-2">Departing</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for voyage in voyages %}
                        <tr class="border-b border-gray-700 {% if loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
                            <td class="p-4">{{ voyage.purpose }}</td>
```

```
                            <td class="px-2 whitespace-nowrap">
                                <img
                                    src="{{ asset('images/'~voyage.planet.imageFilename) }}"
                                    alt="Image of {{ voyage.planet.name }}"
                                    class="inline-block w-8 h-8 rounded-full bg-gray-600 ml-2"
                                >
                            </td>
                            <td class="px-2 whitespace-nowrap">{{ voyage.leaveAt|date('Y-m-d') }}</td>
                        </tr>
                        {% endfor %}
                    </tbody>
                </table>
            </div>
            <div class="flex items-center mt-6 space-x-4">
                <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Previous</a>
                <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Next</a>
            </div>
        </section>
    </div>
{% endblock %}
```

templates/main/_planet_list.html.twig

```twig
<ul>
    {% for planet in planets %}
        <li class="mb-4 group">
            <a href="{{ path('app_planet_show', {
                'id': planet.id,
            }) }}" class="block transform transition duration-300 ease-in-out hover:translate-x-1 hover:bg-gray-700 rounded">
                <div class="flex justify-between items-start p-2">
                    <div class="pr-3">
                        <span class="text-white">{{ planet.name }}</span>
                        <span class="block text-gray-400 text-sm">{{ planet.lightYearsFromEarth|round|number_format }} light years</span>
                    </div>
                    <img
                        class="flex-shrink-0 w-8 h-8 bg-gray-600 rounded-full group-hover:bg-gray-500 transition duration-300 ease-in-out"
                        src="#"
                        alt="Image of {{ planet.name }}"
                    >
                </div>
            </a>
        </li>
    {% endfor %}
</ul>
```

Refresh. Huh, no difference. That's because, at least on a Mac, because I moved and overwrote those files, Twig didn't notice that they were updated... so its cache is out-of-date.

Open a fresh terminal tab and run:

```
php bin/console cache:clear
```

Then... woo! Welcome to Space Inviters! Styled up and ready to go! But there's nothing special about the new templates. We *do* have a list of voyages... but it's all boring, normal Twig code with Tailwind classes.

Referencing Assets Dynamically

We do have some broken planet images though. To fix those, go into the `tutorial/assets/` directory... and move all of those planets into `assets/images/`. Delete the `tutorial/`

folder.

That broken `img` tag comes from the `_planet_list.html.twig` partial. Here it is:

templates/main/_planet_list.html.twig

```
<ul>
    {% for planet in planets %}
        <li class="mb-4 group">
            <a href="{{ path('app_planet_show', {
                'id': planet.id,
            }) }}" class="block transform transition duration-300 ease-in-out hover:translate-x-1 hover:bg-gray-700 rounded">
                <div class="flex justify-between items-start p-2">
// ... lines 8 - 11
                    <img
// ... line 13
                        src="#"
// ... line 15
                    >
                </div>
            </a>
        </li>
    {% endfor %}
</ul>
```

I left it for us to finish! How nice of me! We know that we can do `{{ assets() }}` then something like `images/planets-1.png`. That would work. But this time, the `planet-1.png` part is a dynamic property on the `Planet` entity. So, instead say `~` then `planet.imageFilename`:

templates/main/_planet_list.html.twig

```
<ul>
    {% for planet in planets %}
        <li class="mb-4 group">
            <a href="{{ path('app_planet_show', {
                'id': planet.id,
            }) }}" class="block transform transition duration-300 ease-in-out hover:translate-x-1 hover:bg-gray-700 rounded">
                <div class="flex justify-between items-start p-2">
// ... lines 8 - 11
                    <img
// ... line 13
                        src="{{ asset('images/'~planet.imageFilename) }}"
// ... line 15
                    >
                </div>
            </a>
        </li>
    {% endfor %}
</ul>
```

And... pretty! Yea, I know that's not what Earth and Saturn look like - I've got some randomness in my fixtures - but they're fun to look at!

Using KnpTimeBundle

Since day 6 is the "making everything look awesome day", let's do two more things. To start, I don't love this date. It's boring! I want a cool looking date.

So, install one of my favorite bundles:

```
composer require knplabs/knp-time-bundle
```

This gives us a handy `ago` filter. So as soon as this finishes, spin over and open `homepage.html.twig`. Search for `leaveAt`, here we go. Replace that `date` filter with `ago`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 29
30             <div class="bg-gray-800 p-4 rounded">
31                 <table class="w-full text-white">
↕ // ... lines 32 - 38
39                     <tbody>
40                         {% for voyage in voyages %}
41                         <tr class="border-b border-gray-700 {% if loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
↕ // ... lines 42 - 49
50                             <td class="px-2 whitespace-nowrap">{{ voyage.leaveAt|ago }}</td>
51                         </tr>
52                         {% endfor %}
53                     </tbody>
54                 </table>
55             </div>
↕ // ... lines 56 - 59
60         </section>
61     </div>
62 {% endblock %}
```

And... *much* cooler!

What else? Go check out the CRUD areas. These were generated via MakerBundle... but... I *did* preload them with Tailwind classes so they look good. Wow, there is a *lot* of celebrating right now. I'm not complaining.

But... if you go to a form, it looks terrible! Why? The form markup comes from Symfony's form theme... which outputs the fields *without* Tailwind classes.

Flowbite for Tailwind Examples

So what do we do? Do we need to create a form theme? Fortunately, no. One of the great things about Tailwind is there's an entire ecosystem set up around it. For example, Flowbite is a site with a mixture of open source examples and pro, paid features. On the open source side of things, you can, for example, find an "Alert" page with different markup to make great-looking

alerts. And, you don't need to install anything with Flowbite. These classes are all native Tailwind. You can copy this markup into your project and refresh. Nothing special. And I *love* it.

> **Tip**
>
> Flowbite *does* also come with some JavaScript & a Tailwind Plugin. Check out the bonus chapter for the details!

This also has a forms section where it shows how we can make forms look really nice. In theory, if we could make our forms output like this, they would look great.

Adding a Tailwind Form Theme

And fortunately, there's a bundle that can help us. It's called `tales-from-a-dev/flowbite-bundle`. Click "Installation" and copy the `composer require` line. Then run it:

```
composer require tales-from-a-dev/flowbite-bundle
```

We're getting a lot of stuff installed today! This asks if we want to install the contrib recipe. Let's say yes, permanently. Cool!

Back on the installation instructions, we don't need to register the bundle - that happens automatically - but we *do* need to copy this line for the tailwind configuration file.

Open up `tailwind.config.js`, and paste that:

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
// ... lines 4 - 5
    "./vendor/tales-from-a-dev/flowbite-bundle/templates/**/*.html.twig",
  ],
// ... lines 8 - 11
}
```

This bundle comes with its own form theme template with its own Tailwind classes. So we want to make sure that Tailwind *sees* those and outputs the CSS for them.

The last step over on the docs is to tell our system to *use* this form theme by default. This happens over in `config/packages/twig.yaml`. I'll paste... then fix the indentation:

> **Tip**
>
> Starting in version 0.4.0, use `@TalesFromADevFlowbite/form/default.html.twig`.

config/packages/twig.yaml

```yaml
twig:
// ... line 2
    form_themes: ['@Flowbite/form/default.html.twig']
// ... lines 4 - 8
```

That's it. Go back, refresh and eureka! In just over 10 minutes, we installed Tailwind and started using it *everywhere*.

Tomorrow, we'll turn back to JavaScript and leverage Stimulus to write reliable JavaScript code that we can love.

Chapter 7: Stimulus

Welcome to lucky day number 7. Today we're talking about Stimulus: a small, easy-to-understand JavaScript library that lets us create super-organized code that... just always works. It is one of my favorite reasons to use the Internet.

Installing StimulusBundle

But even though Stimulus is a JavaScript library... Symfony has a bundle to help us load it, get it set up, and use it. So, find your terminal and run:

```
composer require symfony/stimulus-bundle
```

One of the most important things about StimulusBundle is its *recipe*. After it finishes, run:

```
git status
```

The Recipe Changes

Oooh. It made a number of changes. The first is over here in `assets/app.js`. On top - I'll remove that comment - we're now importing a new `bootstrap.js`:

assets/app.js

```
1 import './bootstrap.js';
↕ // ... lines 2 - 16
```

That file starts the Stimulus application.

Notice that this imports an `@symfony/stimulus-bundle` module:

assets/bootstrap.js

```
1  import { startStimulusApp } from '@symfony/stimulus-bundle';
↕  // ... lines 2 - 6
```

The `@` symbol isn't important: that's just a character namespaced JavaScript packages use. The important thing is that this is a *bare* import, which means the browser will try to find this package by looking at our importmap.

Ok! Open up `importmap.php`. The recipe added *two* new entries here:

importmap.php

```
↕  // ... lines 1 - 15
16 return [
↕  // ... lines 17 - 23
24     '@hotwired/stimulus' => [
25         'version' => '3.2.2',
26     ],
27     '@symfony/stimulus-bundle' => [
28         'path' => './vendor/symfony/stimulus-
   bundle/assets/dist/loader.js',
29     ],
30 ];
```

The first is for Stimulus itself - that now lives in the `assets/vendor/` directory. The second is... a kind of "fake" 3rd party package. It says that `@symfony/stimulus-bundle` should resolve to a file in our `vendor/` directory. This is a bit of fanciness: we say `import '@symfony/stimulus-bundle'`... and that will ultimately import this `loader.js` file from `vendor/`.

The recipe also added a `controllers/` directory - the home for our custom Stimulus controllers - and a `controllers.json` file, which we'll talk about tomorrow.

Oh, and in `base.html.twig`, it added this `ux_controller_link_tags()` line:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 7
        {% block stylesheets %}
            {{ ux_controller_link_tags() }}
        {% endblock %}
// ... lines 11 - 14
    </head>
// ... lines 16 - 47
</html>
```

Delete it! That was needed with AssetMapper 6.3, but not anymore. We'll talk about what that did tomorrow anyway.

Using Stimulus

Ok: so, all we've done is `composer require` this new bundle. And *yet*, when we refresh the page and look at the console, Stimulus is already working! This `application #starting` and `application #start` come from Stimulus. That's awesome.

With StimulusBundle, anything we put into the `controllers/` directory will automatically be available as a Stimulus controller. So, the fact that we have a `hello_controller.js` means that we can use a controller named `hello`:

assets/controllers/hello_controller.js

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

/*
 * This is an example Stimulus controller!
 *
 * Any element with a data-controller="hello" attribute will cause
 * this controller to be executed. The name "hello" comes from the
filename:
 * hello_controller.js -> "hello"
 *
 * Delete this file or adapt it for your use!
 */
export default class extends Controller {
    connect() {
        this.element.textContent = 'Hello Stimulus! Edit me in
assets/controllers/hello_controller.js';
    }
}
```

In fact, we can see it right now. When this controller is activated, it replaces the text of the element it's attached to. To prove Stimulus is working, inspect any element on the page... and hack in a `data-controller="hello"`.

When I hit enter, boom! It activates the controller.

Creating a Custom Controller

That was fun, but let's create our own, *real* controller. Copy `hello_controller.js` and create a new file called `celebrate_controller.js`. I'll remove the comments and the connect method:

assets/controllers/celebrate_controller.js

```
import { Controller } from '@hotwired/stimulus';
↕ // ... lines 2 - 3
export default class extends Controller {
↕ // ... lines 5 - 8
}
```

Here's the goal: when we hover over the logo, I want to call a method on the controller that triggers the `js-confetti` library. Start by creating the method. It could be called anything, but `poof()` sure is a fun name!

Head over to `app.js`, copy the `js-confetti` code and delete it:

assets/app.js

```
↕ // ... lines 1 - 9
10 import JSConfetti from 'js-confetti';
11
12 const jsConfetti = new JSConfetti();
13 jsConfetti.addConfetti();
↕ // ... lines 14 - 16
```

Pop that into `celebrate` controller... and move the import statement to the top:

assets/controllers/celebrate_controller.js

```
1 import { Controller } from '@hotwired/stimulus';
2 import JSConfetti from 'js-confetti';
3
4 export default class extends Controller {
5     poof() {
6         const jsConfetti = new JSConfetti();
7         jsConfetti.addConfetti();
8     }
9 }
```

Cool! The last step is to activate this on an element. Do that in `base.html.twig`. Let's see... here's the logo. Add `data-controller="celebrate"`. And to trigger the action on hover, say `data-action=""`... and the suggestion is *almost* correct. The format is, first: the JavaScript *event* that we want to listen to. Instead of `click`, we want `mouseover`. Then `->`, the name of our controller, `#` and the method name: `poof`:

templates/base.html.twig

```
// ... line 1
<html>
// ... lines 3 - 14
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
                    <div class="flex items-center">
                        <a
                            href="{{ path('app_homepage') }}"
                            data-controller="celebrate"
                            data-action="mouseover->celebrate#poof"
                        >
                            <img src="{{ asset('images/logo.png') }}" width="50" alt="Space Inviters Logo" >
                        </a>
// ... lines 27 - 29
                    </div>
// ... lines 31 - 36
                </nav>
            </header>
// ... lines 39 - 48
        </div>
    </body>
</html>
```

That's it! Refresh and celebrate!!! Each time we `mouseover`, it calls the method. You can see this liberally in the console.

Wow, so, as soon as we add a controller to the `controllers/` directory, it's loaded and ready to go. Remember, all with no build.

Lazy-Loading Controllers

But sometimes you might have a controller that's only used on *certain* pages... so you don't want to force your user to download it immediately on *every* page. If you have that situation, you can make your controller lazy. It's the best.

To do that, add this special comment above it: `stimulusFetch: 'lazy'`:

assets/controllers/celebrate_controller.js

```
// ... lines 1 - 3
/* stimulusFetch: 'lazy' */
export default class extends Controller {
// ... lines 6 - 9
}
```

Yes, that *is* pretty crazy. But as soon as we do that, instead of downloading this file on page load, it will wait until it sees an element on the page with `data-controller"celebrate"`.

Watch: delete the `data-controller` temporarily. Then go over, refresh, and on the network tools, if I search for `celebrate`, there's nothing there. If I search for `confetti` - since our controller imports - `js-confetti`, that's *also* not there. Those have *not* been downloaded.

Clear out your network tools. Then go up to the logo and hack in that `data-controller`. We're imitating what would happen if we loaded some fresh HTML via AJAX and... that fresh HTML includes an element with `data-controller"celebrate"`.

As soon as that appears on the page, go back to the network tools. Two new items showed up! It noticed the `data-controller` and downloaded the controller *and* `js-confetti`... since that's imported *from* the controller. And it works brilliantly. I absolutely love this.

Keep this controller lazy, but back in `base.html.twig`, re-add `data-controller`.

One of the great things about Stimulus is that it's used by people all over the Interwebs! And there are many pre-made Stimulus controllers out there to help us solve problems. One group of them is called Symfony-UX. We'll dive into one of its packages tomorrow.

Chapter 8: Symfony UX Packages

Head over to https://ux.symfony.com. This is the site for the Symfony UX Initiative: a group of PHP and JavaScript packages that give us free Stimulus controllers. There's a Stimulus controller that can render chart.js, one that can add an image cropper, and so on.

Today we're going to focus on grabbing a *free* Stimulus controller that will give us a fancy autocomplete `select` element. You can search, select - it's all very nice.

On our site, head to the voyages section and hit edit. The form has a planet dropdown, which is fine... but I want to give it more awesomeness!

Installing UX Autocomplete

So let's get this package installed. The UX Autocomplete library is a mixture of PHP with a Stimulus controller inside. Copy the `composer require` line and paste:

```
composer require symfony/ux-autocomplete
```

When that finishes... run:

```
git status
```

Oooh: the recipe modified two interesting things: `controllers.json` and `importmap.php` We know that everything in the `assets/controllers/` directory will be available as a Stimulus controller. In *addition*, anything in `controllers.json` will *also* be registered as a Stimulus controller:

assets/controllers.json

```
{
    "controllers": {
        "@symfony/ux-autocomplete": {
            "autocomplete": {
                "enabled": true,
                "fetch": "eager",
                "autoimport": {
                    "tom-select/dist/css/tom-select.default.css": true,
                    "tom-select/dist/css/tom-select.bootstrap5.css": false
                }
            }
        }
    },
    "entrypoints": []
}
```

It's a way for third-party PHP packages to add more controllers. The recipe added this entry, which basically means that it'll grab some code from the package we just installed and register it as a Stimulus controller.

The point is, we now have a *third* Stimulus controller in our app! The other change the recipe made is in `importmap.php`: it added a new entry for `tom-select`:

importmap.php

```
↕ // ... lines 1 - 15
return [
↕ // ... lines 17 - 29
    'tom-select' => [
        'version' => '2.3.1',
    ],
];
```

`tom-select` is a JavaScript package... and it's actually what does the heavy lifting for the autocomplete functionality. This stimulus controller is just a small wrapper *around* `tom-select`. And so, since that controller needs `tom-select`, it was added!

UX "autoimport" CSS

But when we refresh the page, we are greeted with a *lovely* error. It says

> *"The `autoimport` `tom-select.default.css` could not be found in `importmap.php`. Try running `importmap:require` and then that path."*

Look back into `controllers.json`. Sometimes, these controllers have an extra section called `autoimport`:

assets/controllers.json

```
{
    "controllers": {
        "@symfony/ux-autocomplete": {
            "autocomplete": {
// ... lines 5 - 6
                "autoimport": {
                    "tom-select/dist/css/tom-select.default.css": true,
                    "tom-select/dist/css/tom-select.bootstrap5.css": false
                }
            }
        }
    },
// ... line 14
}
```

The idea is that a Stimulus controller might have a CSS file that goes along with it. This section allows you to activate or deactivate those CSS files. For example, with `tom-select`, there's a default CSS file. Or if you're using Bootstrap, you can use the Bootstrap 5 CSS file. We could set this to `false` and this to `true`.

One difference between using JavaScript modules in a browser versus Node & Webpack is how *much* of the package you get. With Node, when you `npm add tom-select`, that downloads the *entire* package: the JavaScript files, CSS files and anything else. But with AssetMapper & the browser environment in general, when you `importmap:require tom-select`, that downloads a *single* file: just the JavaScript file. The CSS files are *not* there.

However, with `importmap:require`, we can, of course, grab a package with its name, like this:

```
php bin/console importmap:require tom-select
```

Cool. But we can *also* import a specific file path *within* that package. And, because AssetMapper support CSS files, that path can be to a CSS file.

In other words, if we need this vendor CSS file, we can get it with:

```
php bin/console importmap:require tom-select/dist/css/tom-select.default.css
```

Got it! Over in the `assets/vendor/` directory... there it is! And in `importmap.php`, it's there too. This means it's available for our Stimulus controller to import.

The end result? Error gone! And in the page source, there's the CSS file.

Applying Autocomplete to a Field

Ok, after one `composer require` call, one `importmap:require` call and a ton of *me* yapping, we have a new autocomplete Stimulus controller ready to go.

We could add a `data-controller` to the `select` element. But remember: UX packages are usually a mixture of Stimulus controllers and PHP code. In this case, the PHP code allows us to activate the controller directly in our form. Open up `src/Form/VoyageType.php`. The `planet` field is an `EntityType`:

src/Form/VoyageType.php

```
// ... lines 1 - 10
class VoyageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
// ... lines 16 - 19
            ->add('planet', null, [
                'choice_label' => 'name',
                'placeholder' => 'Choose a planet',
            ])
        ;
    }
// ... lines 26 - 32
}
```

And, thanks to the new package, any `EntityType` or `ChoiceType` now has an `autocomplete` option. Set it to `true`:

src/Form/VoyageType.php

```
// ... lines 1 - 10
class VoyageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
// ... lines 16 - 19
            ->add('planet', null, [
// ... lines 21 - 22
                'autocomplete' => true,
            ])
        ;
    }
// ... lines 27 - 33
}
```

And now... Ta-da! Well, the fashion police might not love this, but it works! That option activated the Stimulus controller: you can even see it on the page. Here's the `select` now with a `data-controller` followed by that controller's long name.

Customizing the CSS

How can we make this look better? Thanks to the `autoimport`, the `tom-select.default.css` at least makes it look okay. If we were using Bootstrap, I'd change this to `true`, this to `false`, `importmap:require` the Bootstrap file and we'd be good.

Right now, there's no official support for Tailwind, so we'll style it manually. Over in `assets/styles/app.css`, I'll remove the `body`. In addition to the Tailwind stuff, you can paste in whatever custom styling you need. These override some of the default styles to look nice in our space-themed, dark mode:

assets/styles/app.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

body {
    background-color: skyblue;
}

/* tom-select custom styling */
/* Base Styles for Dark Mode */
.ts-wrapper {
  @apply border-gray-600;
}
.ts-wrapper .ts-control,
.ts-wrapper.single .ts-control,
.ts-wrapper.single.input-active .ts-control,
.full .ts-control,
.ts-dropdown {
  @apply bg-gray-800 text-white !important;
  box-shadow: none !important;
  background-image: none !important;
  border: none !important;
}

/* Specific Style for the Input Field */
.ts-wrapper .ts-control > input,
.ts-wrapper.single .ts-control > input {
  @apply bg-transparent text-white;
}

.ts-wrapper .ts-dropdown .option {
  @apply bg-gray-800 text-white;
}

/* Active and Hover States for Dropdown Items */
.ts-wrapper .ts-dropdown .active,
.ts-wrapper .ts-dropdown [data-selectable]:hover {
  @apply bg-gray-700 text-white;
}

/* Disabled and Focus States */
.ts-wrapper.disabled .ts-control,
.ts-wrapper.focus .ts-control {
  @apply bg-gray-700 text-gray-400 border-gray-500;
}

```

```
/* Multi-select Tags Style */
.ts-wrapper.multi .ts-control > div {
  @apply bg-gray-600 text-white;
}

/* Border Radius Adjustments */
.ts-wrapper .ts-control,
.ts-wrapper .ts-dropdown,
.ts-wrapper .ts-control > div {
  @apply rounded-md;
}

/* Dropdown Box Shadow */
.ts-wrapper .ts-dropdown {
  @apply shadow-md;
}
```

And now... love it!

Making UX Controllers Lazy

Oh, and remember how we can make *our* controllers lazy by adding a special comment? We can do the same thing with controllers loaded in `controllers.json` by setting `fetch` to `lazy`:

assets/controllers.json

```
{
    "controllers": {
        "@symfony/ux-autocomplete": {
            "autocomplete": {
// ... line 5
                "fetch": "lazy",
// ... lines 7 - 10
            }
        }
    },
// ... line 14
}
```

Check it out. Go to the voyages page. I'll go to my network tools, refresh and search for "autocomplete"... and "TomSelect". Nothing! But as *soon* as we go to the edit page where that's being used: search for autocomplete. There it is! "TomSelect" and the CSS file were also loaded lazily, only when we needed them.

We're now done with day 8! A full week and day into LAST stack! Tomorrow, we're going to crank it up to eleven and transform our app into a sleek, single-page wonder with Turbo! Over the next 7 days... things wil start to get crazy.

Chapter 9: Turbo Drive

It's day 9! Beautiful day 9 where we start to make our app shine. All the fundamentals are in place - AssetMapper, Tailwind & Stimulus - so today is... almost a victory lap. We're about to get a huge bang for our buck thanks to a library called Turbo.

Right now, our site, of course, has full page refreshes. Keep an eye on the logo in the address bar. When I click, everything is done with a full page refresh. That's fine. Never mind, that's not fine! I want our site to have a *devastatingly* great user experience.

Luckily, we have Turbo on our team: a JavaScript library forged from the depths of the internet, bent on destroying all full page refreshes. Watch on their site: you won't see any full page reloads as we navigate. And check out how *fast* that feels. It feels like a single page application, because, well, it *is*, it's just not one that we need to build with a frontend framework like React.

Installing Turbo

Like Stimulus, Symfony has a package that helps us work with this Turbo. Find your terminal, and run:

```
composer require symfony/ux-turbo
```

When that finishes, do:

```
git status
```

Like the other UX package, this modified `controllers.json` and `importmap.php`. In `assets/controllers.json`, it added *two* new controllers:

assets/controllers.json

```
{
    "controllers": {
// ... lines 3 - 12
        "@symfony/ux-turbo": {
            "turbo-core": {
                "enabled": true,
                "fetch": "eager"
            },
            "mercure-turbo-stream": {
                "enabled": false,
                "fetch": "eager"
            }
        }
    },
// ... line 24
}
```

The first is... kind of a fake controller. It loads and activates Turbo - you'll see what that does in a moment - but it's not a Stimulus controller that we'll ever use directly. The second controller is optional - we're not going to talk about it, and it's disabled by default.

The other change, in `importmap.php` is, no surprise: it added `@hotwired/turbo`:

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 36
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
];
```

The result of this single command is *amazing*. When I refresh, watch the address bar: we're not going to see *any* more full page reloads! And everything feels super-duper fast. Uh, I love it. Even the forms! Click edit. Watch: this submits via AJAX. Or, if I go and create a new one, hit enter, *that* submits via AJAX. Our site just got transformed into a single page app with one command!

Turbo: What's the Catch?

You might be thinking:

> *“This is too good to be true, Ryan. What's the catch?”*

Ok, there *is* a catch, but minor for new projects: your JavaScript must be written to work *without* full page refreshes. Historically, we've written our JavaScript to execute on page load... or run on `document.ready`. And those things just don't happen after the first page load. But as long as you have everything written in Stimulus, you're good.

For example: our `celebrate` controller: it doesn't matter how many pages I click around to, that just keeps on rolling.

If your app *isn't* ready for Turbo yet - because of the JavaScript problem - you can disable it. In `app.js`, `import * as Turbo from '@hotwired/turbo'`. Below, say `Turbo.session.drive = false`. I'm not going to do that... so I'll comment it out:

assets/app.js

```
1  import * as Turbo from '@hotwired/turbo';
↕  // ... lines 2 - 5
6  //Turbo.session.drive = false;
↕  // ... lines 7 - 8
```

But why would I install Turbo... just to disable it? Because Turbo is actually *three* parts. The first is called *Turbo Drive*. That's the part that gives us free AJAX navigation on all link clicks and form submits. And *that's* what this disables.

But even if you're not ready for Drive, you can still use the two other parts: *Turbo Frames* and *Turbo Streams*. These are *powerful* and we'll spend a lot of time in this tutorial doing some wild things with them.

Preloading Links

Turbo Drive itself is pretty simple, but it *does* have a few other tricks up its sleeve. And they're constantly adding new things. For example, one feature is called preloading. To show this off, go into `templates/base.html.twig`. If you're ever on a page... and you're *really* sure that you know what link the user is going to click next, you can *preload* that.

For example, on the "voyages" link, add `data-turbo-preload`:

templates/base.html.twig

```
↕ // ... lines 1 - 14
15     <body class="bg-black text-white font-mono">
16         <div class="container mx-auto min-h-screen flex flex-col">
17             <header class="my-8 px-4">
18                 <nav class="flex items-center justify-between mb-4">
19                     <div class="flex items-center">
↕ // ... lines 20 - 27
28                         <a href="{{ path('app_voyage_index') }}" data-turbo-preload class="ml-6 hover:text-gray-400">Voyages</a>
↕ // ... line 29
30                     </div>
↕ // ... lines 31 - 36
37                 </nav>
38             </header>
↕ // ... lines 39 - 48
49         </div>
50     </body>
51 </html>
```

Refresh, inspect element, then go to network tools, XHR... and clear the filter. When I refresh, we immediately see an AJAX request made for the voyages page! Because of this, when we click this link, watch: it's going to be instant. Boom!

Use this only when you're *quite* sure what the next page will be. We don't want to trigger a bunch of unnecessary traffic to your site that won't be used.

Oh, and see these JavaScript errors? These come from Symfony's web debug toolbar and profiler. I'm not sure why... but it doesn't like the preloading. That's something we need to fix, but the preloading itself works fine. You can ignore these.

Back in the template, remove the `data-turbo-preload`... because we don't *really* know what page the user will click to next.

Today was *great*. With one library, we eliminated all full page reloads. What could be next? Tomorrow we'll talk about Turbo Frames: a way for us to create Ajax-loading "portions" of our page, without writing a single line of JavaScript.

Chapter 10: Turbo Frames

On this, day 10: we're going to talk about an ancient concept: frames. If you're old enough on the Internet, like me, you might remember iframes. They were these weird things where you could separate your site into different pieces. And when you clicked a link inside a frame, the navigation *stayed* inside that frame. It was like having separate web pages that you cobbled together into one.

The second part of Turbo is Turbo Frames... which is a *not* weird, modern way to break your page down into parts... kinda similar to iframes.

See this left sidebar? When we click a planet, it takes us to the show page for that planet. Cool. But not cool enough! Instead, when I click a planet, I want that content to load right inside of this sidebar *without* changing pages.

Adding the `<turbo-frame>`

To do that, find the sidebar: it's over in `templates/main/homepage.html.twig`... up near the top. This partial renders that planet list. To make this a frame, find the element that surrounds it and change it to `<turbo-frame>`. And the one rule of frames is that each needs to have an `id` attribute. It should be something unique that describes what it holds. How about `planet-info`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
7         <aside class="hidden md:block md:w-64 bg-gray-900 px-2 py-6">
↕ // ... line 8
9             <turbo-frame id="planet-info">
10                {{ include('main/_planet_list.html.twig') }}
11            </turbo-frame>
12        </aside>
↕ // ... lines 13 - 60
61    </div>
62 {% endblock %}
```

Ok: what does that do? At first, nothing. A `<turbo-frame>` is just an HTML element like a `div`, and so it renders normally. Though, for styling purpose, `turbo-frame` is an *inline* element by default.

However, when we click a link... it's busted! It says "Content missing". And in the console:

> *"The response did not contain the expected `<turbo-frame id="planet-info">`."*

When we click this link, it makes an Ajax request to the show page... like it normally would with Turbo. But because the link is inside a `<turbo-frame>`, it grabs the HTML and looks for a *matching* `<turbo-frame>` with `id="planet-info"`. If it finds that, it grabs the content inside and puts *just* that in the `<turbo-frame>` over here.

Adding the Matching `<turbo-frame>`

This means that each link inside a `<turbo-frame>` - whatever page it goes to - that page *must* have a matching `<turbo-frame>`.

Copy this `<turbo-frame id="planet-info">` and then open `planet/show.html.twig`. Put this around the content that we want to load into the sidebar. I don't really want the `h1`... so put it around this table. Add the closing `</turbo-frame>` at the bottom:

templates/planet/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
↕ // ... lines 7 - 8
9     <turbo-frame id="planet-info">
10         <table class="min-w-full bg-gray-800 text-white">
↕ // ... lines 11 - 33
34         </table>
35     </turbo-frame>
↕ // ... lines 36 - 47
48 </div>
49 {% endblock %}
```

Refresh! And click. How cool is that? It makes an AJAX request to this page, grabs *just* the `<turbo-frame>` content and puts it here.

You know what else would be great? A "back" link! To add that, still inside the `<turbo-frame>`, add a `<div class="mt-2">` then an `a`, `href` set to `{{ path() }}`. Link to the homepage:

templates/planet/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
↕ // ... lines 7 - 8
9     <turbo-frame id="planet-info">
10         <table class="min-w-full bg-gray-800 text-white">
↕ // ... lines 11 - 33
34         </table>
35
36         <div class="mt-2">
37             <a href="{{ path('app_homepage') }}">&lt;-- Back</a>
38         </div>
39     </turbo-frame>
↕ // ... lines 40 - 51
52 </div>
53 {% endblock %}
```

Try it! We don't even need to refresh. Behold! A back link! Whoops, let's make that more of an arrow. When we click it... it goes back! That made an AJAX request to the homepage and looked for a matching `<turbo-frame id="planet-info">`. And guess what that holds? This list of planets.

We're on a roll! Before we finish today, add one more link: an edit link. The route is `app_planet_edit`... with `id` set to `planet.id`:

templates/planet/show.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="m-4 p-4 bg-gray-800 rounded-lg">
// ... lines 7 - 8
    <turbo-frame id="planet-info">
// ... lines 10 - 35
        <div class="mt-2">
            <a href="{{ path('app_homepage') }}">&lt;-- Back</a>

            <a href="{{ path('app_planet_edit', {'id': planet.id}) }}">Edit</a>
        </div>
    </turbo-frame>
// ... lines 42 - 53
</div>
{% endblock %}
```

Cool! this time, if we click a planet... then edit... it doesn't work! And I bet you can guess why. It made an AJAX request to the *edit* page.... but there is *no* matching `<turbo-frame>` on that page. And so, we get this error.

But... I *don't* want to add a `<turbo-frame>` to the edit page. The form wouldn't fit into the sidebar anyway. Nope, when I click this link, I want it to result in a "full page" Turbo navigation.

As soon as you add a `<turbo-frame>`, you need to keep track of the links that you have inside of it and either make sure that each goes to a page that has a matching `<turbo-frame>`.... *or* that you target the link or form to do a *full* visit.

Targeting Links to the Full Page

How do you do that? Find the link, and add `data-turbo-frame` - that's a typo Ryan - set to `_top`:

templates/planet/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
↕ // ... lines 7 - 8
9     <turbo-frame id="planet-info">
↕ // ... lines 10 - 35
36         <div class="mt-2">
↕ // ... lines 37 - 38
39             <a data-turbo-frame="_top" href="{{ path('app_planet_edit', {'id': planet.id}) }}">Edit</a>
40         </div>
41     </turbo-frame>
↕ // ... lines 42 - 53
54 </div>
55 {% endblock %}
```

Now, without refreshing, hit edit. It still breaks. I did the wrong thing. It's `data-turbo-frame="_top"`. There we go.

Now hit edit. Full page navigation! It's still Ajax-powered, but the whole page changes.

The other way to target links or forms to the full page is on the `<turbo-frame>` itself. You might say:

> *"Hey! I want all links in this `<turbo-frame>` to be full page navigation by default."*

To do that, set `target` to `_top`. Then, if you have a *specific* link that you want to load in this frame, add `data-turbo-frame` equals and then the id: `planet-info`.

Both approaches are fine: your call. But adding `target="_top"` to each frame is a bit safer.

Hiding Content Not in a Frame

So this is working *super* well! Except for the fact that if the user *does* ever get to the planet show page, we have an extra set of links. We really only want to show those when we're inside the `<turbo-frame>`. How can we do that?

When Turbo sends an Ajax request for a `<turbo-frame>`, it does add a request header that *tells* your app that this is a Turbo Frame request. You can use that inside Symfony to conditionally do different things... like conditionally render these links.

We *are* going to do that one time later in the tutorial. However, I try to minimize this: it adds complexity. Another option is to hide extra stuff with CSS! For example, we could add a class onto the sidebar... then only show these links if we're *inside* that class.

However, Tailwind doesn't really work like that. In Tailwind, you can't change styling conditionally based on your parent. At least not out-of-the-box. But we *can* do this with a trick called a variant.

The first thing to notice is that a `<turbo-frame>`, by default, looks like this: exactly like we have in our template. But as soon as we click a link, it has a `src` attribute. We can take advantage of that by adding a way inside of Tailwind to style elements *conditionally* based on whether they are inside of a `<turbo-frame>` that has a `src` attribute. Because, it *will* have a `src` over here... but won't have a `src` inside of this `<turbo-frame>`... because it never navigates. In fact, it would be a good idea to add a `target="_top"` to *this* frame, since we don't need fancy frame navigation on this page.

Anyway, Tailwind variants are a bit more advanced, but simple enough. Import this `plugin` module, then go down to `plugins`. I'll paste in some code:

tailwind.config.js

```
const plugin = require('tailwindcss/plugin');

/** @type {import('tailwindcss').Config} */
module.exports = {
// ... lines 5 - 12
  plugins: [
    plugin(function({ addVariant }) {
      addVariant('turbo-frame', 'turbo-frame[src] &')
    }),
  ],
}
```

This adds a variant called `turbo-frame`: you'll see how we use that in a second. It basically applies to an element that's inside a `<turbo-frame>` that has a `src` attribute.

Because we called this `turbo-frame`, copy that. Now, in `show.html.twig`, add a `hidden` class to hide this `div` by default.

When we refresh, it's gone. But then, if we match our `turbo-frame` variant, change to display `block`:

templates/planet/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
↕ // ... lines 7 - 8
9     <turbo-frame id="planet-info">
↕ // ... lines 10 - 35
36         <div class="mt-2 hidden turbo-frame:block" >
37             <a href="{{ path('app_homepage') }}">&lt;-- Back</a>
38 
39             <a data-turbo-frame="_top" href="{{ path('app_planet_edit', {'id': planet.id}) }}">Edit</a>
40         </div>
41     </turbo-frame>
↕ // ... lines 42 - 53
54 </div>
55 {% endblock %}
```

Check it out. When we refresh, those links are still hidden. But over here... we've got them! Because we're inside a `turbo-frame` with a `src` attribute, our variant activates and the display `block` takes over.

Turbo Frames *do* add some complexity, but we've only started to scratch the surface on what they make possible.

Tomorrow, when I hover over each planet, I want to add a cool popover with more planet info. To make that happen, we're going to install *another* third-party Stimulus controller.

Chapter 11: Popover!

On the menu for day 11 is our first big, beautiful, fully-functional feature: a popover. But, like a gorgeous, reusable, lazy-loading popover!

Open source Stimulus controllers already exist to solve lots of different problems. And one of the best sources for them is Stimulus Components: a rich collection of controllers. We're going to work with the one called popover.

If you don't know, a popover is a friendly box that pops over to say hello when you hover on an element. It's like a tooltip, except they're usually bigger and you can hover over the box itself.

Installing & Setting up stimulus-popover

This is a *pure* JavaScript library. But we're not going to install it with `yarn` or `npm`. Instead, you know, run:

```
php bin/console importmap:require stimulus-popover
```

Since we're dealing with a pure JavaScript package, there's no Flex recipe. The only change this made was to `importmap.php`:

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 39
    'stimulus-popover' => [
        'version' => '6.2.0',
    ],
];
```

So we have *access* to the code, but this time, we need to register the controller manually.

That's okay! Copy these lines from the documentation... then open `assets/bootstrap.js`. Paste this on top. We don't need `Application.start()`... and move `application.register()` after... and call it `app`:

```
assets/bootstrap.js
↕ // ... line 1
2 import Popover from 'stimulus-popover';
3
4 const app = startStimulusApp();
5 app.register('popover', Popover);
```

Congrats! We have a shiny new controller named `popover`.

Using the Controller

The goal is to hover over this planet and show a popover with extra info. To get that working, head down on the docs. There's some Rails documentation for server-side stuff.... we don't need that. Here we go. So we need an element with `data-controller"popover"` and, inside, a link that, on `mouseenter` calls a `show` method and, on `mouseleave` calls `hide`. Below, this is the content that will be shown in the popover.

Copy this then, head to `homepage.html.twig`, and search for planets. Here's the `td` and here's the planet image. Paste... then I'll move my `img` inside.

Lovely! Then we need to put this `data-action` somewhere. It's tempting to put it on the `img` itself. But, the library adds the popover content *inside* the element that triggers it... and since you can't put content inside an `img`, it's a no-go. Instead, put it directly on the parent `div`:

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="flex">
// ... lines 7 - 13
        <section class="flex-1 ml-10">
// ... lines 15 - 29
            <div class="bg-gray-800 p-4 rounded">
                <table class="w-full text-white">
// ... lines 32 - 38
                    <tbody>
                        {% for voyage in voyages %}
                        <tr class="border-b border-gray-700 {% if loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
// ... line 42
                            <td class="px-2 whitespace-nowrap">
                                <div
                                    data-controller="popover"
                                    data-action="mouseenter->popover#show mouseleave->popover#hide"
                                >
                                    <img
                                        src="{{ asset('images/'~voyage.planet.imageFilename) }}"
                                        alt="Image of {{ voyage.planet.name }}"
                                        class="inline-block w-8 h-8 rounded-full bg-gray-600 ml-2"
                                    >

                                    <template data-popover-target="content">
                                        <div data-popover-target="card">
                                            <p>This content is in a hidden
                                                template.</p>
                                        </div>
                                    </template>
                                </div>
                            </td>
// ... line 62
                        </tr>
                        {% endfor %}
                    </tbody>
                </table>
            </div>
// ... lines 68 - 71
        </section>
    </div>
```

```
74 {% endblock %}
```

So on `mouseenter` of this div, show the popover, on `mouseleave` of this div, *hide* the popover.

That ought to do the trick! It might look a bit wild at first... but hey, let's dive in and see what happens. And, it... works! It's weird and jumpy, but functional!

Styling the Popover

Time to make it look better. I'll select this entire `div` and paste:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 29
30             <div class="bg-gray-800 p-4 rounded">
31                 <table class="w-full text-white">
↕ // ... lines 32 - 38
39                     <tbody>
40                         {% for voyage in voyages %}
41                         <tr class="border-b border-gray-700 {% if
loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
↕ // ... line 42
43                             <td class="px-2 whitespace-nowrap">
44                                 <div
45                                     data-controller="popover"
46                                     data-action="mouseenter->popover#show
mouseleave->popover#hide"
47                                     class="relative"
48                                 >
49                                     <img
50                                         src="{{
asset('images/'~voyage.planet.imageFilename) }}"
51                                         alt="Image of {{
voyage.planet.name }}"
52                                         class="inline-block w-8 h-8
rounded-full bg-gray-600 ml-2"
53                                     >
54
55                                     <template data-popover-
target="content">
56                                         <div
57                                             data-popover-target="card"
58                                             class="max-w-sm rounded
shadow-lg bg-gray-900 absolute left-0 bottom-10"
59                                         >
60                                             <div class="px-6 py-4">
61                                                 <h4>
62                                                     <a class="hover:text-
blue-300 transition-colors duration-100" href="{{ path('app_planet_show',
{ id: voyage.planet.id }) }}">
63                                                         {{
voyage.planet.name }}
64                                                     </a>
65                                                 </h4>
```

```
66                                                  <small>{{
   voyage.planet.lightYearsFromEarth|round|number_format }} ly</small>
67                                                 </div>
68                                             </div>
69                                         </template>
70                                     </div>
71                                 </td>
↕  // ... line 72
73                             </tr>
74                             {% endfor %}
75                         </tbody>
76                     </table>
77                 </div>
↕  // ... lines 78 - 81
82             </section>
83         </div>
84 {% endblock %}
```

That didn't do anything too big: I added a `relative` class on the outer `div`... and down here, made the popover absolutely positioned and printed out some planet info.

Now... look at that! You know what would be cool? A little arrow! We can add one in pure CSS with an `:after` pseudo-element on the popover `card` target. This is a standard CSS strategy for adding arrows, and you can find it on the web, or you use AI to help generate it.

Open `app.css` and I'll paste in some code. You *can* also do this with Tailwind classes:

assets/styles/app.css

```
↕  // ... lines 1 - 63
64 [data-popover-target=card]:after {
65     content: "";
66     position: absolute;
67     top: 100%;
68     left: 1rem;
69     border-width: .75rem;
70     @apply border-t-white dark:border-t-gray-900 border-transparent;
71 }
```

Go check it out! Love it!

Lazy-Loading with a Turbo Frame

At this point, the popover works great and looks great. Are you up for a challenge? Instead of loading all of this markup on page load, I want to load it via Ajax only once the user hovers. The popover library *does* have a way to do this. But as an extra, extra challenge, I want to do it with regular `ol`' Turbo frames. Because, Frames are *really* good at loading stuff via AJAX! Plus, we'll see a couple of extra frames features that we haven't talked about yet.

To start, we need an endpoint that renders this planet info. In the homepage template, copy that code, then delete it:

templates/main/homepage.html.twig

```
// ... lines 1 - 59
                                        <div class="px-6 py-4">
                                            <h4>
                                                <a class="hover:text-blue-300 transition-colors duration-100" href="{{ path('app_planet_show', { id: voyage.planet.id }) }}">
                                                    {{ voyage.planet.name }}
                                                </a>
                                            </h4>
                                            <small>{{ voyage.planet.lightYearsFromEarth|round|number_format }} ly</small>
                                        </div>
// ... lines 68 - 85
```

In `templates/planet/`, create a new file called `_card.html.twig`, and paste:

templates/planet/_card.html.twig

```
// ... line 1
    <div class="px-6 py-4">
        <h4>
            <a class="hover:text-blue-300 transition-colors duration-100" href="{{ path('app_planet_show', { id: voyage.planet.id }) }}">
                {{ voyage.planet.name }}
            </a>
        </h4>
        <small>{{ voyage.planet.lightYearsFromEarth|round|number_format }} ly</small>
    </div>
// ... lines 10 - 11
```

Next, create an endpoint for this. In `src/Controller/PlanetController.php`, anywhere, I'll paste in a route and controller:

src/Controller/PlanetController.php

```
↕ // ... lines 1 - 14
15 class PlanetController extends AbstractController
16 {
↕ // ... lines 17 - 54
55     #[Route('/{id}/card', name: 'app_planet_show_card', methods: ['GET'])]
56     public function showCard(Planet $planet): Response
57     {
58         return $this->render('planet/_card.html.twig', [
59             'planet' => $planet,
60         ]);
61     }
↕ // ... lines 62 - 94
95 }
```

Nothing special: it queries for the `Planet` and passes it to the template. *In* that template, adjust the variables. Instead of `voyage.planet`, use `planet` in each spot:

templates/planet/_card.html.twig

```
↕ // ... line 1
2     <div class="px-6 py-4">
3         <h4>
4             <a class="hover:text-blue-300 transition-colors duration-100" href="{{ path('app_planet_show', { id: planet.id }) }}">
5                 {{ planet.name }}
6             </a>
7         </h4>
8         <small>{{ planet.lightYearsFromEarth|round|number_format }} ly</small>
9     </div>
↕ // ... lines 10 - 11
```

We now have an AJAX endpoint that returns the content. Here's the magic part. Over in `homepage.html.twig`, we want to load that content right here. Do that by adding a `turbo-frame` with `id` set to `planet-card-` then `{{ voyage.planet.id }}` so it's unique on the page:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 29
30             <div class="bg-gray-800 p-4 rounded">
31                 <table class="w-full text-white">
↕ // ... lines 32 - 38
39                     <tbody>
40                         {% for voyage in voyages %}
41                         <tr class="border-b border-gray-700 {% if
loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
↕ // ... line 42
43                             <td class="px-2 whitespace-nowrap">
44                                 <div
45                                     data-controller="popover"
46                                     data-action="mouseenter->popover#show
mouseleave->popover#hide"
47                                     class="relative"
48                                 >
↕ // ... lines 49 - 54
55                                     <template data-popover-
target="content">
56                                         <div
57                                             data-popover-target="card"
58                                             class="max-w-sm rounded
shadow-lg bg-gray-900 absolute left-0 bottom-10"
59                                         >
60                                             <turbo-frame id="planet-card-
{{ voyage.planet.id }}" src="{{ path('app_planet_show_card', {
61                                                 'id': voyage.planet.id,
62                                             }) }}"></turbo-frame>
63                                         </div>
64                                     </template>
65                                 </div>
66                             </td>
↕ // ... line 67
68                         </tr>
69                         {% endfor %}
70                     </tbody>
71                 </table>
72             </div>
↕ // ... lines 73 - 76
77         </section>
78     </div>
```

```
{% endblock %}
```

Add this same frame in `_card.html.twig`... with the closing tag:

templates/planet/_card.html.twig

```
<turbo-frame id="planet-card-{{ planet.id }}">
    <div class="px-6 py-4">
        <h4>
            <a class="hover:text-blue-300 transition-colors duration-100" href="{{ path('app_planet_show', { id: planet.id }) }}">
                {{ planet.name }}
            </a>
        </h4>
        <small>{{ planet.lightYearsFromEarth|round|number_format }} ly</small>
    </div>
</turbo-frame>
```

Usually, a `<turbo-frame>` will be one part of a whole page. But it's perfectly ok to make an endpoint that *just* returns a single frame.

Back over in `homepage.html.twig`, we have the basic setup. The trick is that... we're not waiting for somebody to click a link inside this frame that will *then* load the card page. Nope, we want it to load immediately.

To do that, add a `src` attribute set to `{{ path() }}`... and... that's almost correct. The route is `app_planet_show_card`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 29
30             <div class="bg-gray-800 p-4 rounded">
31                 <table class="w-full text-white">
↕ // ... lines 32 - 38
39                     <tbody>
40                         {% for voyage in voyages %}
41                         <tr class="border-b border-gray-700 {% if
loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
↕ // ... line 42
43                             <td class="px-2 whitespace-nowrap">
44                                 <div
45                                     data-controller="popover"
46                                     data-action="mouseenter->popover#show
mouseleave->popover#hide"
47                                     class="relative"
48                                 >
↕ // ... lines 49 - 54
55                                     <template data-popover-
target="content">
56                                         <div
57                                             data-popover-target="card"
58                                             class="max-w-sm rounded
shadow-lg bg-gray-900 absolute left-0 bottom-10"
59                                         >
60                                             <turbo-frame id="planet-card-
{{ voyage.planet.id }}" src="{{ path('app_planet_show_card', {
61                                                 'id': voyage.planet.id,
62                                             }) }}"></turbo-frame>
63                                         </div>
64                                     </template>
65                                 </div>
66                             </td>
↕ // ... line 67
68                         </tr>
69                         {% endfor %}
70                     </tbody>
71                 </table>
72             </div>
↕ // ... lines 73 - 76
77         </section>
78     </div>
```

```
79 {% endblock %}
```

Done! When a turbo frame appears that *already* has a `src` attribute, it will make the AJAX call to load that content immediately.

Try it. Refresh and... content missing. I mucked something up. That's ok - we can debug! The call failed with a 500 error. This is where the web debug toolbar comes in handy. We can't see the error easily... but via the Ajax toolbar element, we can click to see the big beautiful exception page. Ah:

> *"Variable `voyage` does not exist."*

Because I need to adjust that to `planet.id`:

templates/planet/_card.html.twig

```
1  <turbo-frame id="planet-card-{{ planet.id }}">
↕  // ... lines 2 - 9
10 </turbo-frame>
```

All right. And now... got it! There *is* a moment when the popover is empty... but we'll fix that soon.

Lazy-Loading Turbo Frames

By complete accident, our Turbo Frame is even being loaded *lazily*. What I mean is: when we have a `<turbo-frame>` with a `src` attribute, the AJAX call will be made *immediately*. With that in mind, shouldn't we see 30 Ajax calls happening all at once? Yea... but we *don't*! It only happens once we hover. Why?

Inspect that element. Ah. It's by accident thanks to the `template` element. The `template` element is special in your browser: anything inside it behaves... as if it's *not* on the page at all: almost like it's a string instead of an element. So, when we first load, the `<turbo-frame>` is *technically* not part of the page. But the moment we hover, it copies that onto the page, the `turbo-frame` comes alive and the Ajax call is made. Pretty cool!

But there *will* be other situations when we want a `turbo-frame` to load its content *only* once that frame becomes *visible*. And we can do that! To show this off, change the `template` to a `div` temporarily:

templates/main/homepage.html.twig

```
// ... lines 1 - 43
                                <div
                                    data-controller="popover"
                                    data-action="mouseenter->popover#show
mouseleave->popover#hide"
                                    class="relative"
                                >
// ... lines 49 - 54
                                    <div data-popover-target="content">
                                        <div
                                            data-popover-target="card"
                                            class="max-w-sm rounded
shadow-lg bg-gray-900 absolute left-0 bottom-10"
                                        >
                                            <turbo-frame id="planet-card-
{{ voyage.planet.id }}" target="_top" src="{{ path('app_planet_show_card',
{
                                                'id': voyage.planet.id,
                                            }) }}"></turbo-frame>
                                        </div>
                                    </div>
                                </div>
// ... lines 66 - 80
```

This is going to look awful... because every card will be visible all at once. Yup! They're all on the page *and* it made 30 Ajax calls immediately! Yikes! To tell these frames to not load until they become visible on the page, add `loading="lazy"`:

templates/main/homepage.html.twig

```
// ... lines 1 - 43
                                <div
                                    data-controller="popover"
                                    data-action="mouseenter->popover#show mouseleave->popover#hide"
                                    class="relative"
                                >
// ... lines 49 - 54
                                    <div data-popover-target="content">
                                        <div
                                            data-popover-target="card"
                                            class="max-w-sm rounded shadow-lg bg-gray-900 absolute left-0 bottom-10"
                                        >
                                            <turbo-frame loading="lazy" id="planet-card-{{ voyage.planet.id }}" target="_top" src="{{ path('app_planet_show_card', {
                                                'id': voyage.planet.id,
                                            }) }}"></turbo-frame>
                                        </div>
                                    </div>
                                </div>
// ... lines 66 - 80
```

Refresh now. 3 ajax calls... because only 3 frames are visible! The other elements *are* all on the page... but below the scroll. Watch this number as I scroll. See that? As they become visible, each makes its AJAX call. *So* cool.

Change the element back to a `template` so that things work nicely again:

templates/main/homepage.html.twig

```
// ... lines 1 - 43
                                <div
                                    data-controller="popover"
                                    data-action="mouseenter->popover#show
mouseleave->popover#hide"
                                    class="relative"
                                >
// ... lines 49 - 54
                                    <template data-popover-
target="content">
                                        <div
                                            data-popover-target="card"
                                            class="max-w-sm rounded
shadow-lg bg-gray-900 absolute left-0 bottom-10"
                                        >
                                            <turbo-frame loading="lazy"
id="planet-card-{{ voyage.planet.id }}" target="_top" src="{{
path('app_planet_show_card', {
                                                'id': voyage.planet.id,
                                            }) }}"></turbo-frame>
                                        </div>
                                    </template>
                                </div>
// ... lines 66 - 80
```

Adding Loading Content

Ok, I'm really happy. But I want to *perfect* this new feature. One thing I don't like is that it's empty before the Ajax call finishes. I'd like to add some loading content.

This is easy: when you have a `turbo-frame` with a `src` attribute, whatever content is inside will be shown by default while it loads. I'll paste in a `div` with an SVG:

templates/main/homepage.html.twig

```
// ... lines 1 - 59
                                            <turbo-frame loading="lazy" id="planet-card-{{ voyage.planet.id }}" target="_top" src="{{ path('app_planet_show_card', {
                                                'id': voyage.planet.id,
                                            }) }}">
                                                <div class="p-10">
                                                    <svg xmlns="http://www.w3.org/2000/svg" class="animate-spin" width="24" height="24" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round">
                                                        <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                                                        <path d="M12 3a9 9 0 1 0 9 9"></path>
                                                    </svg>
                                                </div>
                                            </turbo-frame>
// ... lines 70 - 87
```

This SVG comes from Tabler Icons. That's a *great* source to find an icon that you copy into your project. I've coupled that with an `animate-spin` class from Tailwind.

Let's check it. Quick, spinny and lovely!

Remembering the Ajax Call

Do we have time for one more thing? When we hover over the element, it makes the AJAX call. And... it repeats that *every* time we hover. It doesn't *remember* the content from the AJAX call.

That's due to how the popover controller works... and if I had been less stubborn and used *its* way of Ajax-loading content, it wouldn't be a problem. Anyway, each time we hover, it creates the `turbo-frame`, destroys it, creates a new one, destroys it, etc.

So: how can we make the controller *remember* the content? I have no idea! But let's go look inside the code. In `assets/vendor/stimulus-popover/`, open this file. The contents are minified... but a quick `Cmd`+`L` to reformat the code fixes that. How cool is this? We can now read this vendor file - and even add temporary debugging code if we needed to. And... I think I see a way that we can make this work.

Just like with Symfony controllers, we can override Stimulus controllers. Inside the `controllers/` directory, create our own `popover_controller.js`. Then I'll paste in some code:

assets/controllers/popover_controller.js

```
import Popover from 'stimulus-popover';

export default class extends Popover {
    async show(t) {
        if (this.hasCardTarget) {
            this.cardTarget.classList.remove('hidden');
            return;
        }

        super.show(t);
    }

    hide() {
        this.hasCardTarget && this.cardTarget.classList.add('hidden');
    }
}
```

Normally we import `Controller` from Stimulus and extend that. But in this case, I'm importing the popover controller directly and extending *that*. Then we override the `show` method and `hide` method to toggle a `hidden` class instead of fully destroying the element.

And now that we have our own controller named `popover`, in `bootstrap.js`, we don't need to register the one from Stimulus components. The `popover` controller will be *our* controller... then we leverage the Stimulus components controller via inheritance.

assets/bootstrap.js

```
// ... lines 1 - 3
// app.register('popover', Popover);
```

Moment of truth! It loads once... then *remembers* its content!

Not *only* did we create the perfect popover, we can now easily repeat this on other parts of our site. If you're wondering if we could reuse some of the popover markup... stay tuned for Day 23 when we talk about Twig Components.

That's a wrap for today! Get some well-deserved rest, because tomorrow we'll write a tiny, yet mighty, Stimulus controller called auto-submit.

Chapter 12: Auto-Submitting Forms

Day 12 already? Over the next 3 days, we're going to work on one, big goal: transforming this table into a rich data-table setup, with searching, column filtering, pagination, all happening with beautiful AJAX. This is one of the parts I'm *most* excited to dive into.

Our homepage *does* have a search. And there's nothing particularly special about it. I hit enter to submit the form, the query parameter is in the URL, and it filters the results. Naturally, thanks to Turbo Drive, it all happens via AJAX.

For our *first* trick, watch as we make the search update automatically as we type. So we type and, without hitting enter, the list should update.

To do this, we're going to borrow a controller from a 30 Days of Hotwire repository. This comes from a *fantastic* 30 Days of Hotwire challenge that someone from the Rails community did. I *love* this series and it has a ton of good stuff. I highly recommend checking it out.

The autosubmit Stimulus Controller

Anyway, I'm going to borrow this great "auto-submit" controller. It's dead-simple: it gives us a way to submit a form... with optional debouncing. If I type really quickly, I don't want to submit the form four times. I want it to wait for a slight pause... and *then* submit. That's called debouncing. This waits for a 300 millisecond pause.

So let's roll up our sleeves and get this into our app. Create a new file called `autosubmit_controller.js`... then paste:

assets/controllers/autosubmit_controller.js

```
import { Controller } from "@hotwired/stimulus"
import debounce from 'debounce'

// Connects to data-controller="autosubmit"
export default class extends Controller {
  initialize() {
    this.debouncedSubmit = debounce(this.debouncedSubmit.bind(this), 300)
  }

  submit(e) {
    this.element.requestSubmit()
  }

  debouncedSubmit() {
    this.submit()
  }
}
```

Then head to the homepage to use it. Near the top... here's our search form. On the form, add `data-controller"autosubmit"`:

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="flex">
// ... lines 7 - 13
        <section class="flex-1 ml-10">
            <form
// ... lines 16 - 18
                data-controller="autosubmit"
            >
// ... lines 21 - 30
            </form>
// ... lines 32 - 85
        </section>
    </div>
{% endblock %}
```

Notice I'm getting auto-complete on that. That's thanks to a Stimulus plugin I have for PhpStorm.

Next, down on the input, say `data-action` equals `autosubmit#debouncedSubmit`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
15             <form
↕ // ... lines 16 - 18
19                 data-controller="autosubmit"
20             >
21                 <input
↕ // ... lines 22 - 27
28                     data-action="autosubmit#debouncedSubmit"
29                 >
↕ // ... line 30
31             </form>
↕ // ... lines 32 - 85
86         </section>
87     </div>
88 {% endblock %}
```

In the controller, you can call `submit` to submit the form immediately or `debouncedSubmit()` to wait for the pause. And we don't need to include the event name this time - like `input->` to listen to the `input` event. When you apply a `data-action` to an `input`, a `button` or a `link`, Stimulus figures out which event you want to listen to. Most of the time, life will be simple like this.

Installing the Missing Package

Does it work? No! Because we have an error... an error that I hope will feel familiar!

> *"Failed to resolve module specifier `debounce`."*

This comes from our code! Our copied code is using a `debounce` package... and we don't have that installed! Cool! Copy `debounce`, spin over and run:

```
php bin/console importmap:require debounce
```

Now it's in our project... and now the error is gone. Ready for the magic? Hey, it's working! Just one request after I finished typing thanks to debounce!

The only bummer is that we're losing focus when it reloads the entire page. As a workaround - this is *not* going to be our final solution - we can try putting `autofocus`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6      <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
15             <form
↕ // ... lines 16 - 18
19                 data-controller="autosubmit"
20             >
21                 <input
↕ // ... lines 22 - 28
29                     autofocus
30                 >
↕ // ... line 31
32             </form>
↕ // ... lines 33 - 86
87         </section>
88     </div>
89 {% endblock %}
```

This... *almost* works... except we're losing the cursor location: it puts us back at the beginning. That's okay: we're going to solve this in a much better way soon. And when we do, we're not even going to need the autofocus.

Tomorrow, let's make this richer by adding pagination and column sorting.

Chapter 13: Pagination & Column Sorting

Welcome to Day 13! We're going to tap the breaks on Stimulus and Turbo and only work with Symfony and Twig today. Our goal is to add pagination and column sorting to this list.

Adding Pagination

I like to add pagination with Pagerfanta. I *love* this library, though I do get a bit lost in its documentation. But hey: it's open source, if you're not happy, go fix it!

To use Pagerfanta, we'll install *three* libraries:

```
composer require babdev/pagerfanta-bundle pagerfanta/doctrine-orm-adapter pager
```

Cool beans! Let's get the PHP side working first. Open `src/Controller/MainController.php`. The current page will be stored on the URL as `?page=1` or `?page=2`, so we need to *read* that `page` query parameter. I'll do that with a cool newish `#[MapQueryParameter]` attribute. And actually, before... I was doing too much work. If your query parameter matches your argument name, you don't need to specify it there. So, I'll remove it on those two. It *is* different for `searchPlanet`: a parameter we'll use later.

Anyway, this will read the `?page=` and we'll default it to 1. Oh, and the order of these doesn't matter:

src/Controller/MainController.php

```
↕ // ... lines 1 - 12
13 use Symfony\Component\Routing\Annotation\Route;
14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 20
21         #[MapQueryParameter] int $page = 1,
22         #[MapQueryParameter] string $query = null,
23         #[MapQueryParameter('planets', \FILTER_VALIDATE_INT)] array
   $searchPlanets = [],
24     ): Response
25     {
↕ // ... lines 26 - 37
38 }
```

Below, copy the `$voyageRepository->findBySearch()` line, and replace it with a Pager object: `$pager` equals `Pagerfanta::createForCurrentPageWithMaxPerPage()`:

src/Controller/MainController.php

```
↕ // ... lines 1 - 7
8 use Pagerfanta\Pagerfanta;
↕ // ... lines 9 - 14
15 class MainController extends AbstractController
16 {
↕ // ... line 17
18     public function homepage(
↕ // ... lines 19 - 23
24     ): Response
25     {
26         $pager = Pagerfanta::createForCurrentPageWithMaxPerPage(
↕ // ... lines 27 - 29
30         );
↕ // ... lines 31 - 36
37     }
38 }
```

The first argument is an adapter: new `QueryAdapter` then paste in the code from before. But, that's not quite right: this method returns an array of voyages:

src/Repository/VoyageRepository.php

```
// ... lines 1 - 17
class VoyageRepository extends ServiceEntityRepository
{
// ... lines 20 - 24
    /**
     * @return Voyage[]
     */
    public function findBySearch(?string $query, array $searchPlanets, int $limit = null): array
    {
        $qb =  $this->findBySearchQueryBuilder($query, $searchPlanets);

        if ($limit) {
            $qb->setMaxResults($limit);
        }

        return $qb
            ->getQuery()
            ->getResult();
    }
// ... lines 40 - 60
}
```

but we now need a `QueryBuilder`. Fortunately, I already set things up so that we can get this same result, but as a `QueryBuilder` via: `findBySearchQueryBuilder`:

src/Repository/VoyageRepository.php

```
↕ // ... lines 1 - 17
18 class VoyageRepository extends ServiceEntityRepository
19 {
↕ // ... lines 20 - 40
41     public function findBySearchQueryBuilder(?string $query, array $searchPlanets, ?string $sort = null, string $direction = 'DESC'): QueryBuilder
42     {
43         $qb = $this->createQueryBuilder('v');
44
45         if ($query) {
46             $qb->andWhere('v.purpose LIKE :query')
47                 ->setParameter('query', '%' . $query . '%');
48         }
49
50         if ($searchPlanets) {
51             $qb->andWhere('v.planet IN (:planets)')
52                 ->setParameter('planets', $searchPlanets);
53         }
54
55         if ($sort) {
56             $qb->orderBy('v.' . $sort, $direction);
57         }
58
59         return $qb;
60     }
61 }
```

Paste that method name in.

The next argument is the current page - `$page` - then max per page. How about 10?

src/Controller/MainController.php

```
↕ // ... lines 1 - 6
7  use Pagerfanta\Doctrine\ORM\QueryAdapter;
↕ // ... lines 8 - 14
15 class MainController extends AbstractController
16 {
↕ // ... line 17
18     public function homepage(
↕ // ... lines 19 - 23
24     ): Response
25     {
26         $pager = Pagerfanta::createForCurrentPageWithMaxPerPage(
27             new QueryAdapter($voyageRepository->findBySearchQueryBuilder($query, $searchPlanets)),
28             $page,
29             10
30         );
↕ // ... lines 31 - 36
37     }
38 }
```

Pass `$pager` to the template as the `voyages` variable:

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
↕ // ... line 17
18     public function homepage(
↕ // ... lines 19 - 23
24     ): Response
25     {
26         $pager = Pagerfanta::createForCurrentPageWithMaxPerPage(
27             new QueryAdapter($voyageRepository->findBySearchQueryBuilder($query, $searchPlanets)),
28             $page,
29             10
30         );
31
32         return $this->render('main/homepage.html.twig', [
33             'voyages' => $pager,
↕ // ... lines 34 - 35
36         ]);
37     }
38 }
```

That... should just work because we can loop over `$pager` to get the voyages.

Rendering the Pagination Links

Next up, in `homepage.html.twig`, we need pagination links! Down at the bottom, I already have a spot for this with hardcoded previous and next links:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 82
83             <div class="flex items-center mt-6 space-x-4">
84                 <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Previous</a>
85                 <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Next</a>
86             </div>
87         </section>
88     </div>
89 {% endblock %}
```

The way you're supposed to render Pagerfanta links is by saying `{{ pagerfanta() }}` and then passing `voyages`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 82
83             <div class="flex items-center mt-6 space-x-4">
84                 {{ pagerfanta(voyages) }}
85                 <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Previous</a>
86                 <a href="#" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Next</a>
87             </div>
88         </section>
89     </div>
↕ // ... lines 90 - 91
```

When we try this - let me clear my search out - the pagination looks awful... but it *is* working! As we click, the results are changing.

So... how can we change these pagination links from "blah" to "ah"? There *is* a built-in Tailwind template that you can tell Pagerfanta to use. That involves creating a `babdev_pagerfanta.yaml` file and a bit of configuration. I haven't used this before - so let me know how it goes!

```
babdev_pagerfanta:
    # The default Pagerfanta view to use in your application
    default_view: twig

    # The default Twig template to use when using the Twig Pagerfanta view
    default_twig_template: '@BabDevPagerfanta/tailwind.html.twig'
```

Because... I'm going to be stubborn. I want to *just* have previous & next buttons... and I want to style them *exactly* like this. So let's go rogue!

The first thing we need to do is render these links conditionally, only if there *is* a previous page. To do that, say if `voyages.hasPreviousPage`, then render. And, if we have a next page, render *that*:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 82
83             <div class="flex items-center mt-6 space-x-4">
84                 {% if voyages.hasPreviousPage %}
85                     <a href="#" class="block py-2 px-4 bg-gray-700 text-
   white rounded hover:bg-gray-600">Previous</a>
86                 {% endif %}
87                 {% if voyages.hasNextPage %}
88                     <a href="#" class="block py-2 px-4 bg-gray-700 text-
   white rounded hover:bg-gray-600">Next</a>
89                 {% endif %}
90             </div>
91         </section>
92     </div>
93 {% endblock %}
```

For the URLs, use a helper called `pagerfanta_page_url()`. Pass it the pager, `voyages`, then which page we want to go to: `voyages.previousPage`. Copy that, then repeat it below

with `voyages.nextPage`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="flex">
↕ // ... lines 7 - 13
14         <section class="flex-1 ml-10">
↕ // ... lines 15 - 82
83             <div class="flex items-center mt-6 space-x-4">
84                 {% if voyages.hasPreviousPage %}
85                     <a href="{{ pagerfanta_page_url(voyages, voyages.previousPage) }}" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Previous</a>
86                 {% endif %}
87                 {% if voyages.hasNextPage %}
88                     <a href="{{ pagerfanta_page_url(voyages, voyages.nextPage) }}" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Next</a>
89                 {% endif %}
↕ // ... lines 90 - 92
93             </div>
94         </section>
95     </div>
96 {% endblock %}
```

Lovely! Let's give that a try. Refresh. Love it! The previous page is missing, we click next, and it's there. Click next again. Page 3 is the last one. We got it!

For extra credit, let's even print the current page. Add a div... then print `voyages.currentPage`, a slash and `voyages.nbPages`:

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="flex">
// ... lines 7 - 13
        <section class="flex-1 ml-10">
// ... lines 15 - 82
            <div class="flex items-center mt-6 space-x-4">
                {% if voyages.hasPreviousPage %}
                    <a href="{{ pagerfanta_page_url(voyages, voyages.previousPage) }}" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Previous</a>
                {% endif %}
                {% if voyages.hasNextPage %}
                    <a href="{{ pagerfanta_page_url(voyages, voyages.nextPage) }}" class="block py-2 px-4 bg-gray-700 text-white rounded hover:bg-gray-600">Next</a>
                {% endif %}
                <div class="ml-4">
                    Page {{ voyages.currentPage }}/{{ voyages.nbPages }}
                </div>
            </div>
        </section>
    </div>
{% endblock %}
```

Good job, AI!

And... there we go. Page 1 of 3. Page 2 of 3.

Column Sorting

What about column sorting? I want to be able to click each column to sort by that. For this, we need two new query parameters. A `sort` column name and `sortDirection`. Back to PHP! Add `#[MapQueryParameter]` on a `string` argument called `$sort`. Default it to `leaveAt`. That's the property name for this departing column. Then, do `#[MapQueryParameter]` again to add a string `$sortDirection` that defaults to ascending:

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 21
22         #[MapQueryParameter] string $sort = 'leaveAt',
23         #[MapQueryParameter] string $sortDirection = 'ASC',
↕ // ... lines 24 - 25
26     ): Response
27     {
↕ // ... lines 28 - 42
43     }
44 }
```

Inside the method, I'll paste 2 boring lines that validate that `sort` is a real column:

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 25
26     ): Response
27     {
28         $validSorts = ['purpose', 'leaveAt'];
29         $sort = in_array($sort, $validSorts) ? $sort : 'leaveAt';
↕ // ... lines 30 - 42
43     }
44 }
```

We could probably do the same for `$sortDirection`, but I'll skip and go to `findBySearchQueryBuilder()`. This is already set up to expect the sort arguments. So pass `$sort` and `$sortDirection`... and it should be happy!

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 25
26     ): Response
27     {
28         $validSorts = ['purpose', 'leaveAt'];
29         $sort = in_array($sort, $validSorts) ? $sort : 'leaveAt';
30         $pager = Pagerfanta::createForCurrentPageWithMaxPerPage(
31             new QueryAdapter($voyageRepository-
>findBySearchQueryBuilder($query, $searchPlanets, $sort, $sortDirection)),
↕ // ... lines 32 - 33
34         );
↕ // ... lines 35 - 42
43     }
44 }
```

Finally, we're going to need this info in the template to help render the sort links. Pass `sort` set to `$sort` and `sortDirection` set to `$sortDirection`:

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 25
26     ): Response
27     {
↕ // ... lines 28 - 35
36         return $this->render('main/homepage.html.twig', [
↕ // ... lines 37 - 39
40             'sort' => $sort,
41             'sortDirection' => $sortDirection,
42         ]);
43     }
44 }
```

Adding the Column Sorting Links

The most tedious part is transforming each `th` into the proper sort link. Add an `a` tag and break it onto multiple lines. Set the `href` to this page - the homepage - with an extra `sort` set to `purpose`: the name of this column. For `sortDirection`, this is more complex: if `sort` equals `purpose` and `sortDirection` is `asc`, then we want `desc`. Otherwise, use `asc`.

Finally, in addition to the `sort` and `sortDirection` query parameters, we need to keep any *existing* query parameters that might be present - like the search query. And there's a cool way to do this: `...` then `app.request.query.all`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 55
56             <div class="bg-gray-800 p-4 rounded">
57                 <table class="w-full text-white">
58                     <thead>
59                         <tr>
60                             <th class="text-left py-2">
61                                 <a href="{{ path('app_homepage', {
62                                     ...app.request.query.all(),
63                                     sort: 'purpose',
64                                     sortDirection: sort == 'purpose' and
   sortDirection == 'asc' ? 'desc' : 'asc',
65                                 }) }}">
↕ // ... line 66
67                                 </a>
68                             </th>
↕ // ... lines 69 - 78
79                         </tr>
80                     </thead>
↕ // ... lines 81 - 119
120                 </table>
121             </div>
↕ // ... lines 122 - 132
133         </section>
134     </div>
135 {% endblock %}
```

Done! Oh, but after the word Purpose, let's add a nice down or up arrow. To help, I'll paste a Twig macro. I don't often use macros... but this will help us figure out the direction, then print the correct SVG: a down arrow, an up arrow, or an up and down arrow:

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% macro sortArrow(sortName, sort, sortDirection) %}
    {% if sort == sortName %}
        {% if sortDirection == 'asc' %}
            <svg xmlns="http://www.w3.org/2000/svg" class="inline-block w-4 h-4" width="24" height="24" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round">
                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                <path d="M6 15l6 -6l6 6"></path>
            </svg>
        {% else %}
            <svg xmlns="http://www.w3.org/2000/svg" class="inline-block w-4 h-4" width="24" height="24" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round">
                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                <path d="M6 9l6 6l6 -6"></path>
            </svg>
        {% endif %}
    {% else %}
        <!-- up and down arrow svg -->
        <svg xmlns="http://www.w3.org/2000/svg" class="inline-block w-4 h-4 text-slate-300" width="24" height="24" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round">
            <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
            <path d="M8 9l4 -4l4 4"></path>
            <path d="M16 15l-4 4l-4 -4"></path>
        </svg>
    {% endif %}
{% endmacro %}
// ... lines 27 - 136
```

Down here... use this with `{{ _self.sortArrow() }}` passing `'purpose'`, `sort` and `sortDirection`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 55
56             <div class="bg-gray-800 p-4 rounded">
57                 <table class="w-full text-white">
58                     <thead>
59                         <tr>
60                             <th class="text-left py-2">
61                                 <a href="{{ path('app_homepage', {
62                                     ...app.request.query.all(),
63                                     sort: 'purpose',
64                                     sortDirection: sort == 'purpose' and
sortDirection == 'asc' ? 'desc' : 'asc',
65                                 }) }}">
66                                     Purpose {{ _self.sortArrow('purpose',
sort, sortDirection) }}
67                                 </a>
68                             </th>
↕ // ... lines 69 - 78
79                         </tr>
80                     </thead>
↕ // ... lines 81 - 119
120                 </table>
121             </div>
↕ // ... lines 122 - 132
133         </section>
134     </div>
135 {% endblock %}
```

Phew! Let's repeat all of this for the departing column. Paste, change `purpose` to `leaveAt`, the text to `Departing`... then use `leaveAt` in the other two spots:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 55
56             <div class="bg-gray-800 p-4 rounded">
57                 <table class="w-full text-white">
58                     <thead>
59                         <tr>
↕ // ... lines 60 - 69
70                             <th class="text-left py-2">
71                                 <a href="{{ path('app_homepage', {
72                                     ...app.request.query.all(),
73                                     sort: 'leaveAt',
74                                     sortDirection: sort == 'leaveAt' and
    sortDirection == 'asc' ? 'desc' : 'asc',
75                                 }) }}">
76                                     Departing {{
    _self.sortArrow('leaveAt', sort, sortDirection) }}
77                                 </a>
78                             </th>
79                         </tr>
80                     </thead>
↕ // ... lines 81 - 119
120                 </table>
121             </div>
↕ // ... lines 122 - 132
133         </section>
134     </div>
135 {% endblock %}
```

So, all pretty boring code, though it *was* a bit of work to get this set up. Could we have some tools in the Symfony world to make this all easier to build? Probably. That would be a cool thing for someone to work on.

Moment of truth! Refresh. That looks good. And it works *great*! We can sort by each column... we can paginate. Filtering keeps our page... and keeps the search parameter. It's everything I want! And it's all happening via Ajax! Life is good!

The only hiccup now? That awkward scrolling whenever we do anything. I want this to feel like a standalone app that doesn't jump around. Tomorrow: we'll polish this thanks to Turbo Frames.

Chapter 14: Data Tables with Turbo Frames

Our data tables-like setup is working. And it's *almost* awesome. What I don't love is how it jumps around. Every time we click a link, it jumps back to the top of the page. Boo.

That's because Turbo is reloading the full page. And when it does that, it scrolls to the top... because that's usually what we want! But not this time. I want our data table to work like a little application: where the content changes without moving around.

Turbo 8 Morphing?

There are two great ways to solve this. In Turbo 8 - which is *not* released yet, it's Beta 1 at the time of recording this - there's a new feature called page refreshes that leverages morphing. Without nerding out too much - and I want to - when navigating to the same page - like our search form, column sorting and pagination links *all* do - we can tell Turbo to only update the content on the page that *changed*... *and* to preserve the scroll position. So, keep an eye out for that.

Adding a Turbo Frame

The second way - which works fantastically today - is to surround this entire table with a `<turbo-frame>`. In `homepage.html.twig`, find the `table`. Here it is: this `div` represents the table. Above it, add `<turbo-frame id="voyage-list">`. Indent this `div`... and also these pagination links: we want those to be inside the Turbo frame so that when we click on them, they advance the frame & update:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 55
56             <turbo-frame id="voyage-list">
57                 <div class="bg-gray-800 p-4 rounded">
58                     <table class="w-full text-white">
↕ // ... lines 59 - 120
121                     </table>
122                 </div>
123                 <div class="flex items-center mt-6 space-x-4">
↕ // ... lines 124 - 132
133                 </div>
134             </turbo-frame>
135         </section>
136     </div>
137 {% endblock %}
```

Let's try this. Zap that search clear. And oh... beautiful. Look at that! Everything moves *within* the frame. Try pagination. That too! All of our links point *back* to the homepage... and the homepage, of course, *has* this frame.

But remember: now that this table lives inside a Turbo frame, if we have any links inside, they'll stop working. Again, to fix that, on each link, add `data-turbo-frame="_top"`. Or to be more conservative, go up to the new `<turbo-frame>` and add `target="_top"`. If you do that, you'll also need to find the sorting and pagination links that *should* navigate the frame and add `data-turbo-frame="voyage-list"`.

But I'll remove those... because we don't have any links in the table.

Targeting the Search on the Form

At this point, the pagination and sorting links work perfectly! But... the search? The search is still a full page reload. That makes sense! I didn't put that inside the frame. Why? Because, if we had, when we typed and the frame reloaded, it would have *also* reloaded the search box... which would *still* reset my cursor position. So we *don't* want the form to reload.

Can we... *keep* this outside of the frame but *target* the frame when the form submits? We can! Up on the `form` element that submits, add `data-turbo-frame="voyage-list"`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
38             <form
↕ // ... lines 39 - 42
43                 data-turbo-frame="voyage-list"
44             >
↕ // ... lines 45 - 55
56             </form>
↕ // ... lines 57 - 135
136         </section>
137     </div>
138 {% endblock %}
```

Isn't that cool? Now when we refresh: watch. It's perfect! The table loads, but I *keep* my input focus. This is gorgeous.

Adding a Loading Screen

And now we have time to make things extra fancy! What about a loading indicator on the table while it's navigating? To make this obvious, go to our controller and add a `sleep()` for one second:

src/Controller/MainController.php

```
↕ // ... lines 1 - 14
15 class MainController extends AbstractController
16 {
17     #[Route('/', name: 'app_homepage')]
18     public function homepage(
↕ // ... lines 19 - 25
26     ): Response
27     {
↕ // ... lines 28 - 29
30         sleep(1);
↕ // ... lines 31 - 43
44     }
45 }
```

Now... it's *slow*... and when we click or search, we don't even getting any feedback that the site is *doing* anything.

How *can* we add a loading indicator? This is a spot where Turbo has our back. So here's the `<turbo-frame>` element. Watch the attributes at the end when I navigate. Did you see that? Turbo added an `aria-busy="true"` attribute while it was loading. That's there for accessibility, but it's also something that we can leverage within Tailwind!

Over on that `<turbo-frame>` element, here it is, say `class=""` with `aria-busy:opacity-50`.

This special syntax says that, *if* this element has an `aria-busy` attribute, apply the `opacity-50`. Add one more `aria-busy:` with `blur-sm` to blur the background. And for extra points, include `transition-all` so that the opacity and blur *transition* instead of happening abruptly:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 56
57             <turbo-frame id="voyage-list" class="aria-busy:opacity-50 aria-busy:blur-sm transition-all">
↕ // ... lines 58 - 134
135            </turbo-frame>
136        </section>
137    </div>
138 {% endblock %}
```

Tip

For an even nicer effect, you can also change the opacity & blur only if loading takes longer than, for example, 700ms. Do that by adding an `aria-busy:delay-700` class.

Refresh that and watch. Oh, that's lovely! And it all happens thanks to 3 CSS classes. And, though I love watching that, in `MainController`, remove the sleep.

Advancing the Frame

Is this mission accomplished? *Nearly*. There's one gigantic, horrible problem... with an easy solution. When we search or sort or paginate, the URL doesn't change. That's the default behavior of Turbo frames: when they navigate, they don't update the URL. However, we *can* tell Turbo that we *want* this. On the Turbo Frame, add `data-turbo-action="advance"`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 56
57             <turbo-frame id="voyage-list" data-turbo-action="advance"
class="aria-busy:opacity-50 aria-busy:blur-sm transition-all">
↕ // ... lines 58 - 134
135             </turbo-frame>
136         </section>
137     </div>
138 {% endblock %}
```

Advance means that it will update the URL and *advance* the browser history so that if we hit the "Back" button, it'll go the previous URL. You can also use `replace` to change the URL, but *without* adding to the history.

Watch: this time when we search... the URL updates! And as we navigate to page two or three... it updates... and we can hit back, back, and forward, forward.

We now have a truly *fantastic* data tables setup... entirely written without any custom JavaScript inside of Twig and Symfony. What a time to be alive.

The final teensy problem is this "30 results". As we search, that never changes: it's stuck on whatever number was there when the original page loaded. That's because this lives *outside* the Turbo frame. The easiest fix would be to move it *into* the frame... but I don't want it there! Visually, I want it up here!

We're going to leave that for now. But we'll fix it in a few days with Turbo Streams.

Tomorrow, we're going to dive into a brand-new browser feature! It's called View Transitions, and it'll let us apply visual transitions to any navigation.

Chapter 15: View Transitions

Day 15! We're already halfway through our adventure. And it only gets cooler from here.

To celebrate, today we'll work on something sparkly & new: the View Transitions API. This nifty new feature is supported in most browsers and allows us to have smooth transitions whenever *any* HTML changes on our page.

> **Tip**
>
> Actually, as of Dec 2023, view transitions are supported only in Chrome with support in Firefox and Safari reportedly planned.

And if your user has a browser that *doesn't* support it, that's ok! The transition is just skipped, but everything keeps working. No biggie.

Oh, and, View Transitions will come Standard in Turbo 8 for full page navigation. Though, we'll take them even a bit further.

Evil Martians & Demoing View Transitions

To use View Transitions, you do *not* need any external library. But we're going to use one called "turbo view transitions". This came out of a blog series where the author built a neat project called Turbo Music Drive. In two blog posts on Evil Martians, they talk all about morphing and doing crazy stuff with it in Turbo. They even created a live demo!

In the simplest form, view transitions adds a crossfade as you navigate. But you can also get more specific and connect elements between pages to give them a special transition. Watch: when I click this album, it moves up to the left. There's also a nice crossfade for the rest of the page.

Installing turbo-view-transitions

So let's try this out! Step one, get the `turbo-view-transitions` library. At your terminal, run:

```
php bin/console importmap:require turbo-view-transitions
```

Lovely! To activate transitions, we need to do two things. First, in `base.html.twig`, add a `meta` tag with `name="view-transition"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 6
        <meta name="view-transition">
// ... lines 8 - 14
    </head>
// ... lines 16 - 51
</html>
```

That's how you tell your browser you want these!

Second, in Turbo 7, we need to activate transitions in JavaScript. Head to `app.js`. This will be a rare time when we write JavaScript that doesn't need to live in a Stimulus controller. Copy from their example, paste... and move the `import` to the top:

assets/app.js

```
// ... lines 1 - 4
import { shouldPerformTransition, performTransition } from 'turbo-view-transitions';
// ... lines 6 - 9
document.addEventListener('turbo:before-render', (event) => {
    if (shouldPerformTransition()) {
        event.preventDefault();

        performTransition(document.body, event.detail.newBody, async () => {
            await event.detail.resume();
        });
    }
});

document.addEventListener('turbo:load', () => {
    // View Transitions don't play nicely with Turbo cache
    if (shouldPerformTransition()) Turbo.cache.exemptPageFromCache();
});
```

Done! That should be enough to make the Turbo Drive navigations use view transitions! This leverages an event from Turbo called `turbo:before-render`. The `shouldPerformTransition()` function checks to see if the user's browser supports transitions. If they don't, it's normal behavior. But if it *does*, then `performTransition()` will transition between the old and new body. There's also a little code down here to avoid the turbo page cache. I think that's something that'll work better in Turbo 8 when this is official.

Time for the big reveal! Hit refresh and watch. Oooooh. A smooth crossfade transition! So not only did we eliminate full page reloads, we now have a transition between our pages. Watch out Powerpoint, we're coming for you!

Transition Turbo Frames

But what about Turbo frames? When we click, that still happens instantly. We activated transitions for full page navigations, but not for frames. Can we? Sure!

Copy this listener, and paste below. This time, listen to `turbo:before-frame-render`... and adjust this part. Instead of `document.body`, use `event.target`. That will be the `<turbo-frame>`. And then the new element will be `event.detail.newFrame`:

assets/app.js

```
// ... lines 1 - 24
document.addEventListener('turbo:before-frame-render', (event) => {
    if (shouldPerformTransition()) {
        event.preventDefault();

        performTransition(event.target, event.detail.newFrame, async () =>
{
            await event.detail.resume();
        });
    }
});
```

Done! Refresh and.... click. Transition, check!

Debugging Transitions

And if the transition isn't obvious enough, you can open up your browser tools, click the little "...", go to "more tools", then Animations. It looks like I already had it open. Here, you can change the speed of your animations.

Now... it's super obvious. You can even see how it works. If you scroll during the transition, you can kind of see how it takes a screenshot of the old HTML and blends it with the new. This weird effect isn't normally a problem because it happens so fast, but it's cool to see.

Edge Case: Frames that Advance

To show a problem, let's remove the CSS transition on the table that we just added. So spin over to that template... and take of the `class`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 56
57             <turbo-frame id="voyage-list" data-turbo-action="advance">
↕ // ... lines 58 - 134
135             </turbo-frame>
136         </section>
137     </div>
138 {% endblock %}
```

Refresh... and try it. Huh. Nothing happens. I mean, it *works*... but there was no view transition. Why?

This is subtle. The transition breaks when you have a frame that *advances* the URL. The problem is that, in this situation, Turbo calls `turbo:before-frame-render`... then *also* calls `turbo:before-render` right after. These two, sort of, collide.

But it's an easy fix. Create a variable: `let skipNextRenderTransition` and set it to `false`. Below, if `shouldPerformTransition()` and *not* `skipNextRenderTransition`, then do it:

assets/app.js

```
↕ // ... lines 1 - 9
10 let skipNextRenderTransition = false;
11 document.addEventListener('turbo:before-render', (event) => {
12     if (shouldPerformTransition() && !skipNextRenderTransition) {
↕ // ... lines 13 - 17
18     }
19 });
↕ // ... lines 20 - 42
```

Finally, when our frame starts its transition, set this variable to true. Also include a `setTimeout()`, set that back to `false` and delay this for 100 milliseconds:

assets/app.js

```
↕ // ... lines 1 - 25
26 document.addEventListener('turbo:before-frame-render', (event) => {
27     if (shouldPerformTransition()) {
↕ // ... lines 28 - 29
30         // workaround for data-turbo-action="advance", which triggers
31         // turbo:before-render (and we want THAT to not try to transition)
32         skipNextRenderTransition = true;
33         setTimeout(() => {
34             skipNextRenderTransition = false;
35         }, 100);
↕ // ... lines 36 - 39
40     }
41 });
```

It's a bit weird. But hey, that's programming! We set this to true, Turbo triggers the other listener almost immediately... then 100 milliseconds we reset it. We could probably also replace the `setTimeout()` by setting `skipNextRenderTransition = false` up in the `turbo:before-render` listener.

The most important thing is that... *now* we have a transition! Let's set that back to full speed. I like it!

Disabling Transitions

Try the planet popover frame. Woh! That was weird. To be fully honest, I do *not* know what's happening here. For some reason, the view transition causes the popover to disappear... which is... let's say... *not* ideal. There's probably a way to fix that, but I couldn't crack it.

That's ok... and I think this weirdness is isolated to the popover behavior. Instead, we'll add a way to disable the transitions on a frame.

On the homepage, search for `turbo-frame`. Here it is. Add a new attribute called `data-skip-transition`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 56
57             <turbo-frame id="voyage-list" data-turbo-action="advance">
58                 <div class="bg-gray-800 p-4 rounded">
59                     <table class="w-full text-white">
↕ // ... lines 60 - 82
83                         <tbody>
84                             {% for voyage in voyages %}
85                             <tr class="border-b border-gray-700 {% if
    loop.index is odd %} bg-gray-800 {% else %} bg-gray-700 {% endif %}">
↕ // ... line 86
87                                 <td class="px-2 whitespace-nowrap">
88                                     <div
89                                         data-controller="popover"
90                                         data-action="mouseenter-
    >popover#show mouseleave->popover#hide"
91                                         class="relative"
92                                     >
↕ // ... lines 93 - 98
99                                         <template data-popover-
    target="content">
100                                             <div
101                                                 data-popover-target="card"
102                                                 class="max-w-sm rounded
    shadow-lg bg-gray-900 absolute left-0 bottom-10"
103                                             >
104                                                 <turbo-frame data-skip-
    transition id="planet-card-{{ voyage.planet.id }}" target="_top" src="{{
    path('app_planet_show_card', {
105                                                     'id':
    voyage.planet.id,
106                                                 }) }}">
↕ // ... lines 107 - 112
113                                                 </turbo-frame>
114                                             </div>
115                                         </template>
116                                     </div>
117                                 </td>
↕ // ... line 118
119                             </tr>
120                             {% endfor %}
121                         </tbody>
122                     </table>
```

```
123                 </div>
↕ // ... lines 124 - 134
135             </turbo-frame>
136         </section>
137     </div>
138 {% endblock %}
```

I totally made that up. Over an `app.js`, we can look for that. If `shouldPerformTransition()` and *not* `event.target.hasAttribute('data-skip-transition')`, *then* do the transition:

assets/app.js

```
↕ // ... lines 1 - 25
26 document.addEventListener('turbo:before-frame-render', (event) => {
27     if (shouldPerformTransition() && !event.target.hasAttribute('data-skip-transition')) {
↕ // ... lines 28 - 39
40     }
41 });
```

Now... fixed! And we have transitions on... virtually *every* type of navigation on our site. And in only about 10 minutes! It's crazy!

Now to tomorrow's mission: crafting a dazzling toast notification system that's easy to activate no matter where and how we need to add them. Seeya then!

Chapter 16: Toast Notifications

An important part of any functional beautiful site is a notification system. In Symfony, we often think of flash messages: messages that we render near the top of the page, for example, after the user submits a form. And yes, that *is* what I'm talking about. But just rendering them at the top of the page isn't good enough for us. Instead, I want to render them as rich, toast-style notifications in the upper right that disappear automatically with CSS transitions and can tie my shoes for me.

Rendering Flash Messages

On our CRUD controllers, I'm already setting a `success` flash message... but I'm not rendering it anywhere. In the `templates/` directory, create a new `_flashes.html.twig`. To start, just loop over the success messages with `for message in app.flashes('success')`... and `endfor`:

templates/_flashes.html.twig

```
1  {% for message in app.flashes('success') %}
↕  // ... lines 2 - 4
5  {% endfor %}
```

Inside, I'll paste a very simple flash message, which will start fixed to the bottom of the page:

templates/_flashes.html.twig

```
1  {% for message in app.flashes('success') %}
2      <div class="fixed bottom-0 right-0 m-4 p-4 bg-green-500 text-white
   rounded shadow">
3          {{ message }}
4      </div>
5  {% endfor %}
```

Next, in `base.html.twig`, instead of rendering the flashes somewhere near the top of the body, put them at the bottom. Say `<div id="flash-container">` then `{{ include('_flashes.html.twig') }}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div id="flash-container">
            {{ include('_flashes.html.twig') }}
        </div>
    </body>
</html>
```

The `id="flash-container"` isn't important yet, but it *will* be useful later when we talk about Turbo streams.

So let's see if this works! Hit "Save" and... there we go! It's in a weird spot, but it shows up.

Making the Notification Pretty!

To make this look nicer, let's take a trip to Flowbite. Search for "toast". Ah, this has some great examples for different styles of toast notifications. This has me feeling dangerous!

Tip

I also recommend adding a `data-turbo-temporary` attribute to the root `<div>`. This will remove the flash message before Turbo takes its "snapshot" for caching, This means that if the user clicks "Back" to a page, the toast won't still be visible.

Back in `_flashes.html.twig`, I'll paste in some content:

templates/_flashes.html.twig

```
{% for message in app.flashes('success') %}
    <div
        class="fixed top-5 right-5 flex items-center w-full max-w-xs p-4 mb-4 text-gray-500 bg-white rounded-lg shadow dark:text-gray-400 dark:bg-gray-800"
        role="alert"
    >
        <div class="inline-flex items-center justify-center flex-shrink-0 w-8 h-8 text-green-500 bg-green-100 rounded-lg dark:bg-green-800 dark:text-green-200">
            <svg class="w-5 h-5" aria-hidden="true" xmlns="http://www.w3.org/2000/svg" fill="currentColor" viewBox="0 0 20 20">
                <path d="M10 .5a9.5 9.5 0 1 0 9.5 9.5A9.51 9.51 0 0 0 10 .5Zm3.707 8.207-4 4a1 1 0 0 1-1.414 0l-2-2a1 1 0 0 1 1.414-1.414L9 10.586l3.293-3.293a1 1 0 0 1 1.414 1.414Z"/>
            </svg>
            <span class="sr-only">Check icon</span>
        </div>
        <div class="ms-3 text-sm font-normal">{{ message }}</div>
        <button
            type="button"
            class="ms-auto -mx-1.5 -my-1.5 bg-white text-gray-400 hover:text-gray-900 rounded-lg focus:ring-2 focus:ring-gray-300 p-1.5 hover:bg-gray-100 inline-flex items-center justify-center h-8 w-8 dark:text-gray-500 dark:hover:text-white dark:bg-gray-800 dark:hover:bg-gray-700"
            aria-label="Close"
        >
            <span class="sr-only">Close</span>
            <svg class="w-3 h-3" aria-hidden="true" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 14 14">
                <path stroke="currentColor" stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="m1 1 6 6m0 0 6 6M7 7l6-6M7 7l-6 6"/>
            </svg>
        </button>
    </div>
{% endfor %}
```

This is heavily inspired by the Flowbite examples. But nothing really changed: we're still looping over the same collection and still dumping out the message. We've just got nice markup around this.

And I can't want to see it! I'll go to edit and "Save". Oh, that is wonderful! In the upper right where I want it and all done with CSS.

Making the Toast Closeable

Though, it doesn't auto close yet. Heck, it doesn't close at all! Since "closing" things will be a common problem, let's create a reusable Stimulus controller that can do that.

In `assets/controller/`, add a new `closeable_controller.js`. I'll cheat and grab the code from another controller... clear it out... then add a `close()` method. When this is called, it'll remove the entire element that the controller is attached to:

assets/controllers/closeable_controller.js

```
1  import { Controller } from '@hotwired/stimulus';
2
3  export default class extends Controller {
4      close() {
5          this.element.remove();
6      }
7  }
```

To use this, in `_flashes.html.twig`, attach the controller to the top level element because that's what should be removed on close. Then, down on the button, say `data-action="closeable#close"`:

templates/_flashes.html.twig

```
1  {% for message in app.flashes('success') %}
2      <div
↕  // ... lines 3 - 4
5          data-controller="closeable"
6      >
↕  // ... lines 7 - 13
14         <button
↕  // ... lines 15 - 17
18             data-action="closeable#close"
19         >
↕  // ... lines 20 - 23
24         </button>
25     </div>
26 {% endfor %}
```

We don't need the `click` because this is a `button`, so Stimulus already knows that we want this to trigger on the `click` event.

Let's try it! Hit edit and Save. It's there... it's gone.

In just a few minutes of work, we created a beautiful and functional toast notification system! But, darn it, this is *not* cool enough for our 30 Days of LAST Stack mission! So tomorrow, we'll fancy-ify this with auto-close, CSS transitions and an animated timer bar.

Chapter 17: Fancier Toasts: Auto-close & Fading

Yesterday, we cooked up a beautiful Toast notification system that's powered entirely with CSS and Symfony's normal flash system. Ok, and just a *tiny* bit of JavaScript to, boop, close it.

Today we're going to take this to the next level. I want these toasts to be amazing.

Adding Auto-Close

The first feature we'll add is auto-close: a classic in the toast world where the message graces our screen, then closes automatically after a few seconds. But I also want to keep our closeable controller reusable. There may be other parts of the site where we want to be able to close something... but not have it close itself automatically.

So, we need a way to *activate* the auto-close on a case-by-case basis. The way to pass info into a controller is via values. Add `static values` equals... and I'll invent a new one called `autoClose`, which will be a `Number`:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
4     static values = {
5         autoClose: Number,
6     };
↕ // ... lines 7 - 18
19 }
```

Next, add a `connect()` method. The idea is that if we have `this.autoCloseValue` - that's how you reference that - then... that's actually perfect! We'll use `setTimeout` to close after that many milliseconds:

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 7
    connect() {
        if (this.autoCloseValue) {
            setTimeout(() => {
                this.close();
            }, this.autoCloseValue);
        }
    }
// ... lines 15 - 18
}
```

To finish, go to where we use this controller - `_flashes.html.twig` - to pass in the new `autoClose` value. We do that on the same element as the `data-controller`. Add `data-closeable-auto-close-value` equals and use 5,000 for 5 seconds:

templates/_flashes.html.twig

```
{% for message in app.flashes('success') %}
    <div
// ... lines 3 - 6
        data-closeable-auto-close-value="5000"
    >
// ... lines 9 - 26
    </div>
{% endfor %}
```

The format is `data-` the name of the controller, `auto-close` - that's the name of the value `autoClose`... but because we're in an HTML attribute, we use the "dash case" - then the word `value` equals and finally what we want to pass in. This format is harder to remember than just `data-controller`. But as you saw, if you have this Stimulus plugin for PhpStorm, it auto-completes it, which helps a lot.

Let's do this! Edit this record, save and 1, 2, 3, 4, 5... whoosh! It vanishes.

Auto-close Timer Bar

What's next on our quest for toast greatness? What about a timer bar that shows when the toast will close? A little bar that animates smaller and smaller, then finally disappears right as the toast auto-closes itself.

That sounds cool! Here's the plan: we're going to add an element down here then animate its width from 100% to 0% over those 5 seconds. To be able to *find* that element, inside the controller, we're going to use a target. Add `static targets = ['timerbar']`:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 7
8     static targets = ['timerbar']
↕ // ... lines 9 - 26
27 }
```

Then down in `connect()`, check for that: if `this.hasTimerbarTarget`, then `this.timerbarTarget.style.width = 0`:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 9
10     connect() {
11         if (this.autoCloseValue) {
↕ // ... lines 12 - 15
16             if (this.hasTimerbarTarget) {
↕ // ... line 17
18                 this.timerbarTarget.style.width = 0;
↕ // ... line 19
20             }
21         }
22     }
↕ // ... lines 23 - 26
27 }
```

Assuming we've added a CSS transition to this element, that should animate the change from full width to 0. Oh, but one other detail: add a `setTimeout` and put this inside with a 10-millisecond delay:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 9
10     connect() {
11         if (this.autoCloseValue) {
↕ // ... lines 12 - 15
16             if (this.hasTimerbarTarget) {
17                 setTimeout(() => {
18                     this.timerbarTarget.style.width = 0;
19                 }, 10);
20             }
21         }
22     }
↕ // ... lines 23 - 26
27 }
```

This will allow the element to *establish* itself on the page with a full 100% width, before changing it to 0. This is a CSS transition trick. If you add or unhide an element and *immediately* change its width to 0... the CSS transition won't work. You need to let the element *be* on the page with 100% width for 1 animation frame, *then* change it.

Alrighty, with the stage set, time to add the timer bar. At the bottom of `_flashes.html.twig`, I'll paste it in:

templates/_flashes.html.twig

```
1  {% for message in app.flashes('success') %}
2      <div
↕ // ... lines 3 - 7
8      >
↕ // ... lines 9 - 27
28         <div
29             class="absolute bottom-0 left-0 h-1 bg-green-500 w-full
   transition-all duration-[5000ms] ease-linear"
↕ // ... line 30
31         ></div>
32     </div>
33 {% endfor %}
```

This has an absolute position on the bottom, left of the parent with a height and green background. It also has an explicit width: that's the `w-full`. That's important for the transition.

To make this a target, add `data-closeable-target="timerbar"`:

templates/_flashes.html.twig

```
{% for message in app.flashes('success') %}
    <div
// ... lines 3 - 7
    >
// ... lines 9 - 27
        <div
            class="absolute bottom-0 left-0 h-1 bg-green-500 w-full transition-all duration-[5000ms] ease-linear"
            data-closeable-target="timerbar"
        ></div>
    </div>
{% endfor %}
```

Ok! Let's see what this looks like. Hit edit, save, and it opens... but no animation. Let's do some debugging! No errors in my console. Ah... here's the problem: I should have listened to my editor: `timerbarTarget`.

Let's close this. Save and... *that's* what I want to see! And right as it gets to 0, boop, it closes.

Ok, I *love* how this looks. But our toast deserves one last detail: a graceful fade out... instead of this abrupt exit.

CSS Transition on Close

Fading things out is a bit tricky. You can use CSS transitions - and we will - to go from opacity 100 to 0. But then you also need some JavaScript to *wait* for that CSS transition to finish so that it can finally remove the element from the page or at least set its display to none.

To help us with this, we're going to use a library called `stimulus-use`. Stimulus Components - as we saw earlier - are a list of reusable stimulus controllers. `stimulus-use` is a group of *behaviors* that you can add to your Stimulus controllers. And there are a lot of interesting tools here.

The one we're going to use is called `useTransition`. So step one, let's get this installed. Run:

```
php bin/console importmap:require stimulus-use
```

Awesome! Then over in the controller, import that with `import { useTransition } from 'stimulus-use'`:

assets/controllers/closeable_controller.js

```
↕ // ... line 1
2 import { useTransition } from 'stimulus-use';
↕ // ... lines 3 - 36
```

To activate a behavior, you call it from `connect()`: `useTransition(this)` then pass any options you need. I'll paste a few in:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 3
4 export default class extends Controller {
↕ // ... lines 5 - 10
11     connect() {
12         useTransition(this, {
13             leaveActive: 'transition ease-in duration-200',
14             leaveFrom: 'opacity-100',
15             leaveTo: 'opacity-0',
16             transitioned: true,
17         });
↕ // ... lines 18 - 29
30     }
↕ // ... lines 31 - 34
35 }
```

Here's what this means. While this element is "leaving" or hiding, the library will add these three classes. This establishes that, in case any CSS properties change on this element, we want to have a 200 millisecond transition. The `leaveFrom` means that, at the moment it *starts* hiding, the library will give it this class: setting its opacity to 100. Then, one millisecond later, it will remove this class and add `opacity-0`. That change will trigger the 200 millisecond transition. Finally, `transitioned` true is a way for us to tell the library that we are *starting* in a visible state... because you can also use this library to start hidden and then transition *in* to make your element visible.

Now that we've initialized the behavior, our controller magically has two new methods: `leave()` and `enter()`. Down here in `close()`, instead of removing the element ourselves, say `this.leave()`:

assets/controllers/closeable_controller.js

```
↕ // ... lines 1 - 3
4 export default class extends Controller {
↕ // ... lines 5 - 31
32     close() {
33         this.leave();
34     }
35 }
```

Let's try this! Spin over, refresh, and save. Watch. Ah, it was quick, but that is *exactly* what we wanted! Our toast notification is polished and done.

Tomorrow's adventure: diving into the third and final part of Turbo: *Streams*. These are the Swiss army knife of Turbo, and will let us solve a whole new set of problems.

Chapter 18: Turbo Streams: Update any Element

Today, we're diving headfirst into the finale of the Turbo trilogy: Turbo Streams. Streams allow us to solve problems that we... just don't have a solution for yet.

Take, for instance, our homepage: we have this really nice data tables system... with one teeny tiny problem. When we type into this box, that number of results doesn't change. It's stuck at whatever it was on page load. The Turbo Frame is around this *table*, so the result count is *outside* of that.

This is where Turbo Streams comes in. When you're dealing with a Turbo Frame and you need to update something *outside* of it, you need a stream. Streams have a fancy name, but it's a simple idea. A Turbo Stream is actually a custom HTML element. I could take this, put it onto my page, and it would instantly *execute*. It would find the element on the page whose `id` is `messages` and append this content. And there are actions for everything: prepend, replace, update, etc. We can use a Turbo Stream to make any change we want to any element on the page... from anywhere. The power!

Adding a `<turbo-stream>` Right on the Page

To prove this, copy the Turbo Stream that's an update. Back on our site, inspect element on the "Space Inviters" name. Temporarily, give this an `id` called `company_name` so we can target it.

Now, *anywhere* else on the page - so how about down here in the footer - edit as HTML and paste that Turbo Stream. In this case, we want the target to be `company_name` and inside the template element, say "Space Invaders!". Now, check this out. As soon as I click out of this, the `<turbo-stream>` custom element will become active and will execute its action. Watch. Boom! It found that element and updated it!

Take a peek back at the footer: that `<turbo-stream>` is gone! It executes, then self-destructs and removes itself from the page. It's the most noble of custom elements.

And even if it *were* on the page for a moment, remember: all `<turbo-streams>` have a `template` element inside. We talked about that element on Day 11: anything inside a

`<template>`... isn't *really* on the page at all: it's completely hidden and inactive. So even if this *were* on the page for a moment, it would be invisible.

Streams *just* work.

Updating the Result Count with a Stream

So let's use them to solve our problem! Open `templates/main/homepage.html.twig` and search for "results". Here's the element we need to update. To target this, give it an `id`: how about `voyage-result-count`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
38             <form
↕ // ... lines 39 - 54
55                 <div id="voyage-result-count" class="whitespace-nowrap m-2 mr-4">{{ voyages|length }} results</div>
56             </form>
↕ // ... lines 57 - 141
142         </section>
143     </div>
144 {% endblock %}
```

Copy that. When we search on the page, it's actually this `<turbo-frame>` that's navigating. So *anywhere* inside this - I'll go to the bottom - we can add a `<turbo-stream>`. Say: `<turbo-stream action="replace"`, `target=""` and paste. Then add the `<template>` element - don't forget that - and I'll hard-code a message to start:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 56
57             <turbo-frame id="voyage-list" data-turbo-action="advance">
↕ // ... lines 58 - 135
136                 <turbo-stream action="replace" target="voyage-result-
    count">
137                     <template>
138                         Is this thing on?
139                     </template>
140                 </turbo-stream>
141             </turbo-frame>
142         </section>
143     </div>
144 {% endblock %}
```

Ok, watch what happens when I refresh. Boom! Because the `<turbo-stream>` element exists on page load, it immediately executes and replaces the element with the custom content.

Replacing the Real Content with a Block

So *now*... let's put in the *real* content. Essentially, we want to copy this entire div... and paste it down here. Except... without *actually* duplicating this.

To do this, we'll use a trick with Twig blocks. Surround the result count with a new block called `result_count`... then `endblock` below:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
38             <form
↕ // ... lines 39 - 43
44             >
↕ // ... lines 45 - 54
55                 {% block result_count %}
56                     <div id="voyage-result-count" class="whitespace-nowrap m-2 mr-4">{{ voyages|length }} results</div>
57                 {% endblock %}
58             </form>
↕ // ... lines 59 - 143
144         </section>
145     </div>
146 {% endblock %}
```

In Twig, you're free to add blocks wherever you want. When you do, they don't *do* anything immediately. When this renders, Twig will see this block.... ignore it... and render the `div` like normal.

But now, we can go down inside our `<turbo-stream>` and say `{{ block('result_count') }}`:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
↕ // ... lines 38 - 58
59             <turbo-frame id="voyage-list" data-turbo-action="advance">
↕ // ... lines 60 - 137
138                 <turbo-stream action="replace" target="voyage-result-count">
139                     <template>
140                         {{ block('result_count') }}
141                     </template>
142                 </turbo-stream>
143             </turbo-frame>
144         </section>
145     </div>
146 {% endblock %}
```

I think we're ready! Start by going to the homepage so we see the full 30 results. And then as we type... ah, beautiful! The count updates as the results reload. Dang, that was easy!

For those of you that are nerds for details, first, we love you, and second, yes, on page load, we're rendering the result count twice: here... and, even though we can't see it, we're *also* rendering it down here inside the Turbo Stream. So it's being rendered twice inside the HTML. And that's not a problem at all, unless, for some reason, calculating the result count takes a lot of work. *If* you had that situation, you could set the count to a Twig variable, then render in both spots.

All right, tomorrow we'll start into the biggest, boldest part of this entire series: building a reusable modal system that just absolutely rocks. I'm so excited!

Chapter 19: HTML dialog for Modals

Welcome to day 19. Today we have the luck to play around with a little-known HTML element that absolutely *rocks* when it comes to building modals. The `<dialog>` element. If you're in a hurry for modal magnificence, you can skip ahead to snag the final markup and Stimulus controller. But I promise that today's journey is going to be *fun*.

Open up `templates/voyage/index.html.twig`. For the `h1`, I'm going to paste some new content:

templates/voyage/index.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="m-4 p-4 bg-gray-800 rounded-lg">
    <div
        class="flex justify-between"
    >
        <h1 class="text-xl font-semibold text-white mb-4">Voyages</h1>
        <button
            class="flex items-center space-x-1 bg-blue-500 hover:bg-blue-700 text-white text-sm font-bold px-4 rounded"
        >
            <span>New Voyage</span>
            <svg xmlns="http://www.w3.org/2000/svg" class="w-4 inline" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M3 12a9 9 0 1 0 18 0a9 9 0 0 0 -18 0" /><path d="M9 12h6" /><path d="M12 9v6" /></svg>
        </button>
    </div>
// ... lines 18 - 45
</div>
{% endblock %}
```

This adds a "New voyage" button.

At the bottom, I'll remove the old button. There's nothing special with this new code: it's just... a button. And when we go to the right page... there it is! But it doesn't do anything yet.

Hello `<dialog>`

Back in the template, right after the button, add a `<dialog>` element. Inside I'll proclaim "I am a dialog". Also add an `open` attribute:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          class="flex justify-between"
9      >
↕ // ... lines 10 - 17
18         <dialog open>
19             I am a dialog!
20         </dialog>
21     </div>
↕ // ... lines 22 - 49
50 </div>
51 {% endblock %}
```

Hit refresh and behold the `dialog` element. It's... interesting. The `dialog` is absolutely positioned on the page, centered horizontally and near, but not *at* the top vertically. That's because the `<dialog>` element is *designed* for modals... or really any dialog, like a dismissable alert or any sub window. It's a normal HTML element, but with a bunch of superpowers that we're going to experience.

Making a Pretty dialog

But first, we gotta make it prettier. Back in the template, I'll paste over that dialog:

templates/voyage/index.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="m-4 p-4 bg-gray-800 rounded-lg">
    <div
        class="flex justify-between"
    >
// ... lines 10 - 18
        <dialog
            open
            class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%]"
        >
            <div class="flex grow p-5">
                <div class="grow overflow-auto p-1">
                    <div class="text-white space-y-4">
                        <div class="flex justify-between items-center">
                            <h2 class="text-xl font-bold">Create new Voyage</h2>
                            <button class="text-lg absolute top-5 right-5">
                                <svg xmlns="http://www.w3.org/2000/svg" class="w-4" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M18 6l-12 12"/><path d="M6 6l12 12"/></svg>
                            </button>
                        </div>
                        <p class="text-gray-400">
                            Join us on an exciting journey through the cosmos! Discover the
                            mysteries of the universe and explore distant galaxies.
                        </p>
                        <div class="flex justify-end">
                            <button
                                class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
                                Let's Go!
                            </button>
                        </div>
                    </div>
                </div>
            </div>
        </dialog>
    </div>
// ... lines 47 - 74
</div>
```

```
76 {% endblock %}
```

This is adapted from Flowbite with some AI help. And a designer could create this no problem. Because, there's nothing special: we still have a `dialog`, it's still `open`... and even the Tailwind classes are pretty boring. I set a width... and round the corners. But most of the positioning details are already built into the element. And most of the code is dummy modal content to get us started.

The result... is *awesome*. Though... the close button doesn't do its job yet! No worries: this is a *great* opportunity to show off one of dialog's superpowers!

Find the close button. Around it, add a `<form method="dialog">`:

templates/voyage/index.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="m-4 p-4 bg-gray-800 rounded-lg">
    <div
        class="flex justify-between"
    >
// ... lines 10 - 18
        <dialog
            open
            class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%]"
        >
            <div class="flex grow p-5">
                <div class="grow overflow-auto p-1">
                    <div class="text-white space-y-4">
                        <div class="flex justify-between items-center">
// ... line 27
                            <form method="dialog">
                                <button class="text-lg absolute top-5 right-5">
                                    <svg xmlns="http://www.w3.org/2000/svg" class="w-4" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M18 6l-12 12"/><path d="M6 6l12 12"/></svg>
                                </button>
                            </form>
                        </div>
// ... lines 34 - 43
                    </div>
                </div>
            </div>
        </dialog>
    </div>
// ... lines 49 - 76
</div>
{% endblock %}
```

This is a normal button: it will naturally submit the form when we click it, but the button doesn't have anything special on it.

But now when we click X... it *closes*!

Opening with a modal Stimulus Controller

To *really* make the `<dialog>` element shine, we need a bit of JavaScript. Head up to `assets/controllers/` and create a new file called `modal_controller.js`. I'll cheat, steal some content from another controller... and clear it out. This controller will be simple. Start by adding a `static targets = ['dialog']` so we can quickly find the `<dialog>` element. Next add an `open` method. Here, say `this.dialogTarget.show()`:

assets/controllers/modal_controller.js

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

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

    open() {
        this.dialogTarget.show();
    }
}
```

This is another superpower of the `<dialog>` element: it has a `show()` method! Built *into* the `<dialog>` element is this core idea of showing and hiding.

To use the new controller, over in `index.html.twig`, find the `div` that holds the `button` and the `dialog` and add `data-controller="modal"`. Then, on the button, say `data-action="modal#open"`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          data-controller="modal"
9          class="flex justify-between"
10     >
↕ // ... lines 11 - 12
13         <button
14             data-action="modal#open"
15             class="flex items-center space-x-1 bg-blue-500 hover:bg-blue-
   700 text-white text-sm font-bold px-4 rounded"
16         >
↕ // ... lines 17 - 18
19         </button>
↕ // ... lines 20 - 49
50     </div>
↕ // ... lines 51 - 78
79 </div>
80 {% endblock %}
```

Finally, we need to set the `<dialog>` as a target. Remove the `open` attribute so it starts closed and replace it with `data-modal-target="dialog"`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          data-controller="modal"
9          class="flex justify-between"
10     >
↕ // ... lines 11 - 20
21         <dialog
22             class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-
   full md:w-fit md:max-w-[50%] md:min-w-[50%]"
23             data-modal-target="dialog"
24         >
↕ // ... lines 25 - 49
50     </div>
↕ // ... lines 51 - 78
79 </div>
80 {% endblock %}
```

I like it! Over here, it starts closed. And when we click, open! Close, open, close!

Opening as a Modal

A `<dialog>` element has two *modes*: the normal mode that we've been using and a *modal* mode... which is much more useful. To use the modal mode, instead of `show()`, use `showModal()`:

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 5
    open() {
        this.dialogTarget.showModal();
    }
}
```

Now when we click, it still opens, but there are some subtle differences. The first is that we can close it by hitting `Esc`. Cool! The second is that it has a backdrop. Watch: when I click, the screen will get just a little bit darker. Did you see that? This also *blocks* me from interacting with the rest of the page. And we get this for *free* thanks to `<dialog>`. That's *huge*.

Styling the Backdrop

Inspect and find the `<dialog>` element - there it is. The backdrop is added via a pseudo-element called `backdrop`. So it takes care of adding that for us... but it's a *real* element that can *style*. And I do want to style it!

Back in the template, find the `dialog` element. Thanks to Tailwind, we can style the backdrop pseudo-element directly. Add `backdrop:bg-slate-600` and `backdrop:opacity-80`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
7     <div
8         data-controller="modal"
9         class="flex justify-between"
10    >
↕ // ... lines 11 - 20
21        <dialog
22            class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-
      full md:w-fit md:max-w-[50%] md:min-w-[50%] backdrop:bg-slate-600
      backdrop:opacity-80"
23            data-modal-target="dialog"
24        >
↕ // ... lines 25 - 48
49        </dialog>
50    </div>
↕ // ... lines 51 - 78
79 </div>
80 {% endblock %}
```

Watch the effect. That is starting to feel really, really smooth.

Removing Background Page Scroll

One thing the `dialog` element doesn't handle automatically is... the page in the background still scrolls. It doesn't hurt anything... but it's not the behavior we expect.

To fix this, over in the `open()` method, say `document.body` to get the body element, `.classList.add('overflow-hidden')`:

assets/controllers/modal_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 5
6     open() {
↕ // ... line 7
8         document.body.classList.add('overflow-hidden');
9     }
10 }
```

And now... that's what we want!

Cleaning up on Close

Though... if we close, I still can't scroll! We need to *remove* that class.

To do that, copy the `open()` method, paste and name it `close()`. To close the dialog, call `close()`... then remove `overflow-hidden`:

> **Tip**
>
> To code more defensively (Firefox may need this), use:
>
> ```
> if (this.hasDialogTarget) {
>     this.dialogTarget.close();
> }
> ```

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 10
    close() {
        this.dialogTarget.close();
        document.body.classList.remove('overflow-hidden');
    }
}
```

I like it! There's just one tiny problem: we're not *calling* the `close()` method! If we hit X or press Esc, the dialog is closing, yes, but I still can't scroll because nothing calls this `close()` method on our controller.

Fortunately, the `dialog` element has our back. Whenever a `dialog` element closes - for any reason - it dispatches an event called `close`. We can listen to that.

On the `<dialog>` element, add a `data-action` set to `close->modal#close`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
7     <div
8         data-controller="modal"
9         class="flex justify-between"
10    >
↕ // ... lines 11 - 20
21        <dialog
↕ // ... lines 22 - 23
24            data-action="close->modal#close"
25        >
↕ // ... lines 26 - 49
50        </dialog>
51    </div>
↕ // ... lines 52 - 79
80 </div>
81 {% endblock %}
```

So no matter *how* the `dialog` closes - I'll press Escape - we can now scroll because the `close()` method on our controller *was* called.

Blurring the Background

Tip

Thanks to help from Rob Meijer, you can do this in pure CSS. On the `<dialog>` element use `backdrop:bg-opacity-80` instead of `backdrop:opacity-80` then add `backdrop:backdrop-blur-sm`. No JS needed!

Ok, I'm excited. What else can we do? How about blurring the background? You might try to do this by blurring the backdrop. I *totally* tried that... but couldn't make it work. That's ok. What we *can* blur is the body. Add one more class: `blur-sm` and remove the `blur-sm` in `close()`:

assets/controllers/modal_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 5
6     open() {
↕ // ... line 7
8         document.body.classList.add('overflow-hidden', 'blur-sm');
9     }
10
11    close() {
↕ // ... line 12
13        document.body.classList.remove('overflow-hidden', 'blur-sm');
14    }
15 }
```

Let's see how this look. That is *really* cool!

Close on Click Outside

But if I try to click outside the modal, it doesn't close. That's another thing the `dialog` element doesn't handle. Fortunately, there's a quick one-time fix.

Up on the root element of our controller... Actually, we can put it down here on the `dialog`. Add a new action: `click->modal#clickOutside`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
7     <div
8         data-controller="modal"
9         class="flex justify-between"
10    >
↕ // ... lines 11 - 20
21        <dialog
↕ // ... lines 22 - 23
24            data-action="close->modal#close click->modal#clickOutside"
25        >
↕ // ... lines 26 - 49
50        </dialog>
51    </div>
↕ // ... lines 52 - 79
80 </div>
↕ // ... lines 81 - 82
```

I bet that looks odd - it'll be called whenever we click *anywhere* in the dialog - so let's go write that method. Say `clickOutside()`, give it an `event` argument, then if `event.target === this.dialogTarget`, `this.dialogTarget.close()`:

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 15
    clickOutside(event) {
        if (event.target === this.dialogTarget) {
            this.dialogTarget.close();
        }
    }
}
```

Tip

To make the "click outside" work perfectly, instead of adding padding directly to the `dialog`, add an element inside and give *it* the padding. We've done that already - but it's an important detail.

`event.target` will be the *actual* element that received the click. It turns out, the only way to click *exactly* on the `dialog` element itself is if you click the backdrop. If you click anywhere else inside, `event.target` will be one of these elements. So it's a clever three lines of code, but the result is perfect. Click in here, no problem. Click out there, closed.

CSS Animation to Fade In

At this point, I am happy! But this tutorial isn't about making good things, it's about making *great* things. Next up: I want the `dialog` element to fade in. We *could* do this with a CSS transition. But another option is a CSS animation. I know, transitions, animations - CSS has a lot.

An animation is something you apply to an element and... it'll just... *do* that animation forever. Or you can make it animate just once. Like, we can make this button animate up and down forever. One of the nice things about animations is that you can make an animation only happen once... and it won't start until the element becomes *visible* on the page. For example, we could create an animation from opacity 0 to opacity 100, which would execute as soon as our `dialog` becomes visible.

Tailwind *does* have some built-in animations, but not one for fading in. So, we'll add it. Down in `tailwind.config.js`, I'll paste over the `theme` key:

tailwind.config.js

```
↕ // ... lines 1 - 3
4 module.exports = {
↕ // ... lines 5 - 9
10   theme: {
11     extend: {
12       animation: {
13         'fade-in': 'fadeIn .5s ease-out;',
14       },
15       keyframes: {
16         fadeIn: {
17           '0%': { opacity: 0 },
18           '100%': { opacity: 1 },
19         },
20       },
21     },
22   },
↕ // ... lines 23 - 27
28 }
```

This is mostly CSS animation stuff: it adds a new one called `fade-in` that will go from opacity 0 to 100 in 1/2 a second.

To use this, find the `dialog` element and add `animate-fade-in`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          data-controller="modal"
9          class="flex justify-between"
10     >
↕ // ... lines 11 - 20
21         <dialog
22             class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
↕ // ... lines 23 - 24
25         >
↕ // ... lines 26 - 49
50         </dialog>
51     </div>
↕ // ... lines 52 - 79
80 </div>
81 {% endblock %}
```

Try it out. Gorgeous! Could we fade out? Sure, but I actually like that it closes immediately. So I'm going to skip that.

Modals & Turbo Page Cache

Ok, I have *one* last detail before I let you go for the day. When we added view transitions, in `app.js`, we disabled a feature in Turbo called page cache... because it apparently doesn't always play nicely with view transitions. When view transitions become standard in Turbo 8, I'm guessing this won't be a problem.

Anyway, when caching is enabled:

assets/app.js

```
↕ // ... lines 1 - 20
21 document.addEventListener('turbo:load', () => {
22     // View Transitions don't play nicely with Turbo cache
23     // if (shouldPerformTransition()) Turbo.cache.exemptPageFromCache();
24 });
↕ // ... lines 25 - 42
```

the moment you click away from a page, Turbo takes a snapshot of the page before navigating away. When we click back, it's instant: boom! Instead of making a network request, it uses the cached version of this page. There's more to it than that, but you get the idea.

With caching enabled, one thing we need to worry about is removing any temporary elements from the page *before* the snapshot is taken, like toast messages or modals. Because, when you click "Back", you don't want a toast notification to be sitting up here.

The way that we normally solve this, for example in `_flashes.html.twig`, is to add a `data-turbo-temporary` attribute:

templates/_flashes.html.twig

```
{% for message in app.flashes('success') %}
    <div
// ... lines 3 - 4
        data-turbo-temporary
// ... lines 6 - 7
    >
// ... lines 9 - 31
    </div>
{% endfor %}
```

That tells Turbo to remove this element before it takes the snapshot.

Let's try adding this to our `dialog` so it's not in the snapshot. To see what happens, open the modal and click back. That just took a snapshot of the previous page. Now click forward. Woh. We're in a strange state. It looks like the dialog is gone... but we can't scroll and the page is blurred.

That's because we need to do *more* than just hide the `dialog`: we need to remove these classes from the body. Basically, before Turbo takes the snapshot, we need something to call the `close()` method!

And we can do that! In `index.html.twig`, on the root controller element - though this could go anywhere - add a `data-action=""`. Right before Turbo takes its snapshot, it dispatches an event called `turbo:before-cache`. We can listen to that and then call `modal#close`. The only detail is that the `turbo:before-cache` event isn't dispatched on a specific element. So listening to it on *this* element won't work. It's dispatched above us, on the window. It's a global event.

Fortunately, Turbo gives us a simple way to listen to global events by adding `@window`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
7     <div
↕ // ... line 8
9         data-action="turbo:before-cache@window->modal#close"
↕ // ... line 10
11    >
↕ // ... lines 12 - 51
52    </div>
↕ // ... lines 53 - 80
81 </div>
82 {% endblock %}
```

It's a little technical, but with this one-time fix, we can open the modal, go back, forward, and the page looks beautiful.

Wowza! Today was a huge day, but look what we accomplished! A beautiful modal system that we have total control over. Tomorrow is going to be just as big as we bring this modal to life with real dynamic content and forms. See you then.

Chapter 20: AJAX Modal!

Yesterday we built a killer modal system thanks to the `dialog` element. With just this markup and a small Stimulus controller, I'm feeling dangerous.

So let me tell you about today's goal, which is big and bold! When I click "New Voyage", I want to AJAX-load the "new modal form" and pop it into the modal. But more than that! When I submit the form, validation errors should stay in the modal, it should close on success & we should see toast notifications. *And*, maybe most importantly, I want this entire system to be reusable so that we can quickly load *any* form on our site in a modal. We're going to do it, or die trying. Hopefully we'll do it... I think we'll do it.

Adding a modal Frame to the Layout

To get this going, copy the entire modal markup. There we go. Then go into `base.html.twig` and, all the way at the bottom, before the closing `body` tag, paste:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
// ... lines 18 - 70
            <dialog
                class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
                data-modal-target="dialog"
                data-action="close->modal#close click->modal#clickOutside"
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <div class="text-white space-y-4">
                            <div class="flex justify-between items-center">
                                <h2 class="text-xl font-bold">Create new Voyage</h2>
                                <form method="dialog">
                                    <button class="text-lg absolute top-5 right-5">
                                        <svg xmlns="http://www.w3.org/2000/svg" class="w-4" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M18 6l-12 12"/><path d="M6 6l12 12"/></svg>
                                    </button>
                                </form>
                            </div>
                            <p class="text-gray-400">
                                Join us on an exciting journey through the cosmos! Discover the
                                mysteries of the universe and explore distant galaxies.
                            </p>
                            <div class="flex justify-end">
                                <button
                                    class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
                                    Let's Go!
                                </button>
                            </div>
                        </div>
                    </div>
                </div>
```

```
100             </dialog>
101         </div>
102     </body>
103 </html>
```

Back in `index.html.twig`, remove the `dialog`... and we don't need the modal controller stuff anymore:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="m-4 p-4 bg-gray-800 rounded-lg">
7     <div
8         class="flex justify-between"
9     >
10        <h1 class="text-xl font-semibold text-white mb-4">Voyages</h1>
11
12        <button
↕ // ... line 13
14            class="flex items-center space-x-1 bg-blue-500 hover:bg-blue-
  700 text-white text-sm font-bold px-4 rounded"
15        >
↕ // ... lines 16 - 17
18        </button>
19    </div>
↕ // ... lines 20 - 47
48 </div>
49 {% endblock %}
```

This is now a normal `h1` and a normal button... that doesn't do anything. In `base.html.twig`, do the opposite: remove the `button`, the `h1` and the class on the div:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
            data-controller="modal"
            data-action="turbo:before-cache@window->modal#close"
        >
            <dialog
                class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
                data-modal-target="dialog"
                data-action="close->modal#close click->modal#clickOutside"
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <div class="text-white space-y-4">
                            <div class="flex justify-between items-center">
                                <h2 class="text-xl font-bold">Create new Voyage</h2>
                                <form method="dialog">
                                    <button class="text-lg absolute top-5 right-5">
                                        <svg xmlns="http://www.w3.org/2000/svg" class="w-4" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M18 6l-12 12"/><path d="M6 6l12 12"/></svg>
                                    </button>
                                </form>
                            </div>
                            <p class="text-gray-400">
                                Join us on an exciting journey through the cosmos! Discover the
                                mysteries of the universe and explore distant galaxies.
                            </p>
                            <div class="flex justify-end">
                                <button
                                    class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
                                    Let's Go!
                                </button>
                            </div>
```

```
                        </div>
                    </div>
                </div>
            </dialog>
        </div>
    </body>
</html>
```

It's now a div that contains a `dialog`... that's closed.

Now for the magic touch: remove the guts of the `dialog`: only keep these two divs: they help give us padding and nice scroll behavior. Inside, add a `<turbo-frame>` with `id="modal"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
            data-controller="modal"
            data-action="turbo:before-cache@window->modal#close"
        >
            <dialog
                class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
                data-modal-target="dialog"
                data-action="close->modal#close click->modal#clickOutside"
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame id="modal"></turbo-frame>
                    </div>
                </div>
            </dialog>
        </div>
    </body>
</html>
```

That, my friends, was a coding power move. On every page, we now have a `<turbo-frame id="modal">` that we can dynamically load content into! *And*, it lives inside a dialog!

Loading Content into the modal Frame

Watch: on the index page, change the new voyage button to an `a` tag and set its `href` to the `app_voyage_new` route. It's a normal tag that would take us to that page. But now add `data-turbo-frame="modal"`:

templates/voyage/index.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="m-4 p-4 bg-gray-800 rounded-lg">
    <div
        class="flex justify-between"
    >
        <h1 class="text-xl font-semibold text-white mb-4">Voyages</h1>

        <a
            href="{{ path('app_voyage_new') }}"
            data-turbo-frame="modal"
            class="flex items-center space-x-1 bg-blue-500 hover:bg-blue-700 text-white text-sm font-bold px-4 rounded"
        >
// ... lines 17 - 18
        </a>
    </div>
// ... lines 21 - 48
</div>
{% endblock %}
```

Check it out. Refresh and click. Instead of changing the page, it loaded the content into the `modal` frame. But... nothing happened.

Ok, it *did* make an AJAX call to the new voyage page. But if we open that up in a new tab, there's no `modal` frame on this page. Well, actually there *is*. Like *every* page, at the bottom, it has an empty `modal` frame. So when we click that link, it *does* work... but the result is that the Turbo frame stays empty. Not super helpful.

To fix this, in `new.html.twig`, add a `<turbo-frame id="modal">` around everything... with a closing tag at the bottom:

templates/voyage/new.html.twig

```
// ... lines 1 - 4
{% block body %}
    <turbo-frame id="modal">
    <div class="m-4 p-4 bg-gray-800 rounded-lg">
// ... lines 8 - 22
    </div>
    </turbo-frame>
{% endblock %}
```

Check it out. When we click now, yes! Inside the `<turbo-frame>`, we have the form! The modal isn't opening yet, but it's *ready*.

Adding the modal Base Layout

Now, before we figure out how to open the modal, we have a problem... and an opportunity. If we went directly to the new voyage page, we would have *two* `<turbo-frame id="modal">` elements: the one around the form, and the empty one on the bottom. That's... kind of invalid.

Also, even though I want to be able to load this form inside the modal, I *also* want it to behave like *normal* if we go to the page directly. Watch: right now, if I fill this out successfully and save, weird things happen! I submitted that into a `<turbo-frame id="modal">`... it redirected to the index page... which has that matching frame... but it's empty.

That's not what I want. If I go to this page directly, I want it to act like normal.

We're going to handle this with a trick. In `new.html.twig`, remove the `<turbo-frame>`... and extend a new base template called `modalBase.html.twig`:

templates/voyage/new.html.twig

```
{% extends 'modalBase.html.twig' %}
// ... lines 2 - 24
```

Ooh. Copy that name and in the `templates/` directory, create it: `modalBase.html.twig`. This will have one line: an extends tag that's dynamic. If `app.request.headers.get('turbo-frame')` equals `modal` - so if an AJAX request is being made to this page from the `modal` turbo frame, extend a new `modalFrame.html.twig`. Else, extend the normal `base.html.twig`:

templates/modalBase.html.twig

```
{% extends app.request.headers.get('turbo-frame') == 'modal' ?
'modalFrame.html.twig' : 'base.html.twig' %}
```

If we go to the page like normal, it will extend `base.html.twig`. There's no turbo frame here, it's completely normal, and it will submit like normal.

Let's create that other base template. Copy its name and, in `templates/`, create `modalFrame.html.twig`. All this needs is a `<turbo-frame id="modal">`... with `{% block body %}` and `{% endblock %}` inside:

templates/modalFrame.html.twig

```
<turbo-frame id="modal">
    {% block body %}{% endblock %}
</turbo-frame>
```

So if we make a request to this page from the `modal` frame, the *entire* response will be this single frame with the page's content inside. *That* solves our problem. And it means that if we want a page to be able to load its form inside a modal... the only line we to need to change is right here. I'll prove that on Day 23.

Auto-Opening the Modal

But right now, we're back to the situation where we click this link and... if I dig into the modal elements, it *is* loading the form into the `turbo-frame`... but the modal isn't opening. How can we do that?

Well, I have 2 requirements for opening the modal. The first is that I want it to be super easy to open. If HTML appears inside this `turbo-frame` - no matter *how* it's added - I want the system to be smart enough to see that and open the modal. And second, I want the modal to open instantly. I don't want to click this button... then wait for 500 milliseconds before I see the modal. That's not a good user experience.

For part one - opening this modal as soon as there's content in the `turbo-frame` - we're going to use a trick inside our Stimulus controller. Let me close a few files. In `base.html.twig`, make this `turbo-frame` a target: `data-modal-target="dynamicContent"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
            <dialog
// ... lines 61 - 63
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame id="modal" data-modal-target="dynamicContent"></turbo-frame>
                    </div>
                </div>
            </dialog>
        </div>
    </body>
</html>
```

Here's the idea: if a modal has this target and HTML gets inside of this element for any reason, I want our code to *notice* that and open the modal. To do that, in `modal_controller.js`, add that target:

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    static targets = ['dialog', 'dynamicContent'];
// ... lines 5 - 51
}
```

And then I'll paste in the most complex code that we're going to see in this tutorial:

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 5
    observer = null;

    connect() {
        if (this.hasDynamicContentTarget) {
            // when the content changes, call this.open()
            this.observer = new MutationObserver(() => {
                const shouldOpen =
this.dynamicContentTarget.innerHTML.trim().length > 0;

                if (shouldOpen && !this.dialogTarget.open) {
                    this.open();
                } else if (!shouldOpen && this.dialogTarget.open) {
                    this.close();
                }
            });
            this.observer.observe(this.dynamicContentTarget, {
                childList: true,
                characterData: true,
                subtree: true
            });
        }
    }

    disconnect() {
        if (this.observer) {
            this.observer.disconnect();
        }
        if (this.dialogTarget.open) {
            this.close();
        }
    }
// ... lines 36 - 51
}
```

But, hold on: even if it looks complex, what it's *doing* is simple. If we have a `dynamicContent` target, this code watches that element for any changes. Anytime there *is* a change, it calls our function. Then we check to see if the `dynamicContentTarget` element has any HTML. If it does, open it! If it doesn't, close it. It's that simple.

In `disconnect()`, we deactivate that system. And also, just in case, if our modal controller element is ever removed from the page for any reason, this will close the dialog and do the

cleanup.

The result of this is... pretty incredible. Refresh the page. Let's play. I'm going to edit the `<turbo-frame>` as HTML and type: "will this open?". Boom! It does! And if we empty the content... it closes.

And, more importantly, when we click the "New" link, it pops open with the form! Amazing!

Ok, I think that's enough for today. Tomorrow, we're going to make sure this form submits. And because I can't help myself, we'll add a few more goodies: like opening the modal *instantly* instead of waiting for the AJAX call to finish.

Chapter 21: Fantastic Modal UX with a Loading State

Let's pick up where we left off yesterday. The Ajax-powered modal loads! Try to submit it. Uh oh - something went wrong. It went to some page that didn't have a `<turbo-frame id="modal">`... which is odd, because *every* page now has one. That's because... the response was an error. If we look down on the web debug toolbar, there was a 405 status code. Open that up. Interesting:

> *"No route found for `POST` /voyage/"*

That's weird because we're submitting the new voyage form... so the URL should be `/voyage/new`.

Adding action Attributes to the Forms

Here's the problem: when I generated the voyage crud from MakerBundle, it created forms that *don't* have an `action` attribute. That's fine when the form lives on `/voyage/new` because no `action` means it submits back to the current URL. But as soon as we decide to embed our forms on other pages, we need to be responsible and always set the `action` attribute.

To do that, open up `src/Controller/VoyageController.php`. At the bottom, I'll paste in a simple private method. Hit Okay to add the `use` statement:

src/Controller/VoyageController.php

```
// ... lines 1 - 9
use Symfony\Component\Form\FormInterface;
// ... lines 11 - 15
class VoyageController extends AbstractController
{
// ... lines 18 - 88
    private function createVoyageForm(Voyage $voyage = null): FormInterface
    {
        $voyage = $voyage ?? new Voyage();

        return $this->createForm(VoyageType::class, $voyage, [
            'action' => $voyage->getId() ? $this->generateUrl('app_voyage_edit', ['id' => $voyage->getId()]) : $this->generateUrl('app_voyage_new'),
        ]);
    }
}
```

We can pass a voyage or not... and this creates the form but sets the `action`. If the voyage has an id, it sets the action to the edit page, else it sets it to the new page.

Thanks to this, up in the `new` action, we can say `this->createVoyageForm($voyage)`. Copy that... because we need the exact line down in `edit`:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 26
27     public function new(Request $request, EntityManagerInterface $entityManager): Response
28     {
↕ // ... line 29
30         $form = $this->createVoyageForm($voyage);
↕ // ... lines 31 - 45
46     }
↕ // ... lines 47 - 56
57     public function edit(Request $request, Voyage $voyage, EntityManagerInterface $entityManager): Response
58     {
59         $form = $this->createVoyageForm($voyage);
↕ // ... lines 60 - 73
74     }
↕ // ... lines 75 - 96
97 }
```

Lovely. Back over, we don't even need to refresh. Open the modal, save and... Ah, that is *absolutely* lovely! It's submitted and we got the response *right* back inside the modal. Because... of course! That's the whole point of a Turbo frame. It keeps the navigation inside itself.

Loading the Modal Instantly

Before we talk about what happens on success, I want to *perfect* this. My second requirement for opening the modal was that it needs to open immediately. Over in the `new` action, add a `sleep(2)`... to pretend our site is getting slammed by aliens planning their spring break trips:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 26
27     public function new(Request $request, EntityManagerInterface
   $entityManager): Response
28     {
↕ // ... lines 29 - 31
32         sleep(2);
↕ // ... lines 33 - 46
47     }
↕ // ... lines 48 - 97
98 }
```

When we click the button now... nothing happens. No user feedback at *all* until the Ajax request finishes. That is *not* good enough. Instead, I want the modal to open immediately with a loading animation.

Over in the modal controller, add a new target called `loadingContent`:

assets/controllers/modal_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
4     static targets = ['dialog', 'dynamicContent', 'loadingContent'];
↕ // ... lines 5 - 60
61 }
```

Here's my idea: if you want some loading content, you'll define what that looks like in Twig and set this target on it. We'll do that in a moment.

At the bottom, create a new method called `showLoading()`. If `this.dialogTarget.open`, so if the dialog is already open, we don't need to show the loading, so return. Otherwise, say `this.dynamicContentTarget` - for us, that's the `<turbo-frame>` that the Ajax content will eventually be loaded into - `.innerHTML` equals `this.loadingContentTarget.innerHTML`:

assets/controllers/modal_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 52
    showLoading() {
        // do nothing if the dialog is already open
        if (this.dialogTarget.open) {
            return;
        }

        this.dynamicContentTarget.innerHTML = this.loadingContentTarget.innerHTML;
    }
}
```

Finally, add that target. In `base.html.twig`, after the `dialog`, I'll add a `template` element. Yes, my beloved `template` element: it's perfect for this situation because anything inside won't be visible or active on the page. It's a template we can steal from. Add a `data-modal-target="loadingContent"`. I'll paste some content inside:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
// ... lines 60 - 75
            <template data-modal-target="loadingContent">
                <div class="bg-space-pattern bg-cover rounded-lg p-8">
                    <div class="space-y-2">
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                    </div>
                </div>
            </template>
        </div>
    </body>
</html>
```

Nothing special here: just some Tailwind classes with a cool pulse animation.

If we try this now... no loading content! That's because nothing is calling the new `showLoading()` method. Over in `base.html.twig`, find the frame. I'll break this onto multiple lines. Let's think: as soon as the `turbo-frame` starts loading, we want to call `showLoading()`. Fortunately, Turbo dispatches an event when it starts an AJAX request. And we can listen to that.

Add a `data-action` to listen to `turbo:before-fetch-request` - that's the name of the event - then `->modal#showLoading`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
            <dialog
// ... lines 61 - 63
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
                            id="modal"
                            data-modal-target="dynamicContent"
                            data-action="turbo:before-fetch-request-
>modal#showLoading"
                        ></turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 75 - 90
        </div>
    </body>
</html>
```

All right, let's check out the effect! Refresh the page and... oh, it's wonderful! It opens instantly, we see that loading content... and it's replaced when the frame finishes!

I love how this works. When this calls `showLoading()`, that method puts content into `dynamicContentTarget`. And... do you remember what happens the moment any HTML goes into that? Our controller notices it, and opens the dialog. That's some great teamwork!

Loading Indication on Form Submit

We're nearly there to making this perfect, but I'm not satisfied! While we still have the `sleep`, submit the form. Nothing happens! There's no feedback while that's loading.

> **Tip**
>
> For an even nicer effect, you can also change the opacity only if loading takes longer than, for example, 700ms. Do that by adding an `aria-busy:delay-700` class.

Lucky for us, we've been down this road before with a different Turbo frame. Add class `aria-busy:opacity-50`, and `transition-opacity`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
            <dialog
// ... lines 61 - 63
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
// ... lines 68 - 70
                            class="aria-busy:opacity-50 transition-opacity"
                        ></turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 76 - 91
        </div>
    </body>
</html>
```

I'll reload... click, loading animation and submit. Yes! The low opacity tells us that something is happening.

And with that, I will happily remove our `sleep`:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 26
27     public function new(Request $request, EntityManagerInterface
   $entityManager): Response
28     {
↕ // ... lines 29 - 31
32         sleep(2);
↕ // ... lines 33 - 46
47     }
↕ // ... lines 48 - 97
98 }
```

Conditional Modal Styling

Ok, one final detail that I want to get right: this extra padding. This exists because the content from the `new` page has an element with `m-4` and `p-4`. So the modal has some padding... and then extra padding comes from that page.

On the page, the margin and padding make sense. It comes from over here in `new.html.twig`. So we *do* want this on the full page... but not in the modal.

To help us do this, we're going to use a Tailwind trick. In `tailwind.config.js`, add one more variant. Call this `modal`, and activate it whenever we are inside a `dialog` element:

tailwind.config.js

```
↕ // ... lines 1 - 3
4 module.exports = {
↕ // ... lines 5 - 22
23   plugins: [
24     plugin(function({ addVariant }) {
↕ // ... line 25
26       addVariant('modal', 'dialog &');
27     }),
28   ],
29 }
```

Now, in `new.html.twig`, keep the margin and padding for the normal situation. But if we're in a modal, use `modal:m-0`, and `modal:p-0`:

templates/voyage/new.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="m-4 p-4 modal:m-0 modal:p-0 bg-gray-800 rounded-lg">
// ... lines 7 - 21
    </div>
{% endblock %}
```

Back on the new page, this shouldn't change. Looks good! But in the modal... *that* is what we want.

Our modal system now opens instantly, AJAX-loads content, we can submit it and even closes itself on success! Watch: fill in a purpose, select a planet... and... the modal closed!

How? It's cool! The `new` action redirects to the index page. And because `index.html.twig` extends the normal `base.html.twig`, it *does* have a `modal` frame... but it's that empty one at the bottom. That causes the `turbo-frame` on the *page* to become empty. And thanks to our modal controller, we notice that and close the dialog.

The only thing we're missing now, if you were watching closely, is the toast notification! Tomorrow, we'll talk all about handling success when a form is submitted inside a frame... *including* doing cool things like automatically adding the new row to the table on this page. See ya tomorrow.

Chapter 22: Fancy things on Modal Form Success

We have been busy. We've cooked up a reusable AJAX-powered modal system that I *love*. Submitting with validation errors already works. And success? It's nearly there. We when save... no toast notification, but the modal *did* close.

The reason it closed is important. In the `new()` action, we redirect to the index page. That page extends the normal `base.html.twig`... so it *does* have a `<turbo-frame id="modal">` on it... but it's this empty one. This means the modal frame becomes empty, our modal Stimulus controller notices that then closes it.

Planning: When Forms are in Frames

In general, when you add a `<turbo-frame>` around something - like on the homepage with our planets sidebar - you need to think about where the links inside point to. We need to make sure each goes to a page that has a matching `<turbo-frame>`.

When a *form* lives inside a `<turbo-frame>`, we need to think about what happens on *submit*. The error case is easy: it always renders the same page that has the same frame with the errors inside. But on success, we need to think about where the form redirects to and ask: does that page have a matching `<turbo-frame>` and does it contain the right content?

In the case of this modal and the index page, it's perfect: there *is* a matching frame, it's empty and the modal closes.

Rendering Success Flashes with a Turbo Streams

Ok, back to the missing toast notification! This is a situation where we need to update the `<turbo-frame>` - to empty it - and we *also* need to update another area on the page: we need to render the success flash messages into the flash container.

This is a super common need when a form submits inside a `<turbo-frame>`. So we're going to solve this, I think, in a cool and global way. When we redirect on success, this

`<turbo-frame>` is ultimately loaded on the page, which causes the modal to close. Inside it, add a `<turbo-stream>` with `action="append"` and `target="flash-container"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
            <dialog
// ... lines 61 - 63
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
                            id="modal"
// ... lines 69 - 71
                        >
                            <turbo-stream action="append" target="flash-container">
// ... line 74
                            </turbo-stream>
                        </turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 80 - 95
        </div>
    </body>
</html>
```

When we added the toast system, we added an element with `id="flash-container`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div id="flash-container">
            {{ include('_flashes.html.twig') }}
        </div>
// ... lines 55 - 96
    </body>
</html>
```

We didn't need that then, but now it's going to come in handy because we can target that to add flash messages into it.

Inside the stream, add the `template` tag, of course, then `{{ include('_flashes.html.twig') }}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 55
        <div
// ... lines 57 - 58
        >
            <dialog
// ... lines 61 - 63
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
                            id="modal"
// ... lines 69 - 71
                        >
                            <turbo-stream action="append" target="flash-container">
                                <template>{{ include('_flashes.html.twig') }}</template>
                            </turbo-stream>
                        </turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 80 - 95
        </div>
    </body>
</html>
```

This will render the flash messages... and the stream will append them into that container.

Let's try it! Fill out a new voyage, submit and... absolutely nothing happens. The problem... is subtle. When we redirect to the index page, Symfony renders that entire page... even though Turbo will only use the `<turbo-frame id="modal">`. This means that, right before we render this code, our flash container renders the flash messages... which *removes* them from the flash system. So the flashes messages *are* in the HTML that we return from the Ajax call... but because they're not inside the `<turbo-frame>`, they don't make it onto the page.

The fix is easy: make sure your flash container is *after* the modal:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div
            data-controller="modal"
            data-action="turbo:before-cache@window->modal#close"
        >
// ... lines 56 - 91
        </div>

        <div id="flash-container">
            {{ include('_flashes.html.twig') }}
        </div>
    </body>
</html>
```

Give this a go. Refresh... and fill in the form. Got it! The Modal closes, then the `<turbo-stream>` triggers the toast!

And this is really neat! When we redirect, the `<turbo-frame>` is now *not* empty: it contains the flash `<turbo-stream>`. But remember: as soon as a `<turbo-stream>` activates, it executes itself and then disappears. Once that happens, the `<turbo-frame>` becomes empty and the modal closes. I really dig that.

Stream Extras: Prepending the Table

What I love about the modal system is that it works... and we haven't needed to make any changes to our controller. But now, we get to think about any optional *extra* behavior that we might want.

For example, could we prepend the table with the new voyage? Because, right now we don't see it until after we refresh. Let's try!

In `index.html.twig`, find the `table`. We need to prepend into the `tbody`. To target this, on the `table`, add an `id="voyage-list"`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
↕ // ... lines 7 - 21
22     <table class="min-w-full bg-gray-800 text-white" id="voyage-list">
↕ // ... lines 23 - 39
40     </table>
41 </div>
42 {% endblock %}
```

Let's think: this is another case where we need to update something that lives outside the `<turbo-frame>`. So, we need a stream.

Open `new.html.twig` and after the `body` block, add a new block called `stream_success`, then `endblock`. Inside, we'll add any Turbo streams we need to make the submit *really* shine. Add a `<turbo-stream>` `action="prepend"` then `targets=""`. The "s" on targets means we can use a CSS selector: `#voyage-list tbody`. Add the `<template>` element... and, for now, a `<tr><td>` `{{ voyage.purpose }}`:

templates/voyage/new.html.twig

```
↕ // ... lines 1 - 24
25 {% block stream_success %}
26     <turbo-stream action="prepend" targets="#voyage-list tbody">
27         <template>
28             <tr><td>{{ voyage.purpose }}</td></tr>
29         </template>
30     </turbo-stream>
31 {% endblock %}
```

Ok, so we have a new block in our template... that nobody is using. Somehow, we need to grab this Turbo stream... and, after the redirect, render it on the *next* page in the modal `<turbo-frame>`.

How do we do that? We have two options - and I'll show the second on Day 24. But here's the system I like.

First, we only need to worry about prepending the table row when we're submitting inside a `<turbo-frame>`. If we went to the new voyage page directly - which doesn't have a frame - and submitted, we wouldn't need any Turbo Stream stuff. This would navigate the full page and render normally. Nice & simple.

So, in the controller, start with if `$request->headers->has('turbo-frame')`. So if this form submit is happening inside a `<turbo-frame>`, then we want to use our stream. Render that block with `$stream` equals then a relatively new controller method: `$this->renderBlockView()` passing `voyage/new.html.twig`. Instead of rendering the entire template, to render a single block pass this, you guessed it, `stream_success`. Actually... I think I'm missing an "s". I am! Better.

Pass the template a `voyage` variable.

To pass the `<turbo-stream>` string to the next page add it to a new flash called `stream`:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 25
26     #[Route('/new', name: 'app_voyage_new', methods: ['GET', 'POST'])]
27     public function new(Request $request, EntityManagerInterface
   $entityManager): Response
28     {
↕ // ... lines 29 - 32
33         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 34 - 38
39             if ($request->headers->has('turbo-frame')) {
40                 $stream = $this->renderBlockView('voyage/new.html.twig',
   'stream_success', [
41                     'voyage' => $voyage
42                 ]);
43
44                 $this->addFlash('stream', $stream);
45             }
↕ // ... lines 46 - 47
48         }
↕ // ... lines 49 - 53
54     }
↕ // ... lines 55 - 104
105 }
```

Finally, when we redirect to the index page and this `<turbo-frame>` is rendered, *output* that flash: `for stream in app.flashes('stream')`, `endfor` with `{{ stream|raw }}` so it renders the raw HTML elements:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div
// ... lines 53 - 54
        >
            <dialog
// ... lines 57 - 59
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
                            id="modal"
// ... lines 65 - 67
                        >
// ... lines 69 - 71
                            {% for stream in app.flashes('stream') %}
                                {{ stream|raw }}
                            {% endfor %}
                        </turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 79 - 94
        </div>
// ... lines 96 - 99
    </body>
</html>
```

I think we're ready! Refresh... add a new voyage and... that's incredible! The Ajax call redirected to the index page, where the modal frame had 2 Turbo streams: one to render the toast and the other to prepend the table.

Prepending with Real Content

Last step, prepend the real content. What we want is this `tr`. To get that from inside of `new.html.twig`, we need to isolate it into its own template. Copy that, delete it, then include `voyage/_row.html.twig`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          class="flex justify-between"
9      >
↕ // ... lines 10 - 21
22     <table class="min-w-full bg-gray-800 text-white" id="voyage-list">
↕ // ... lines 23 - 30
31         <tbody class="divide-y divide-gray-600">
32             {% for voyage in voyages %}
33                 {{ include('voyage/_row.html.twig') }}
34             {% else %}
35                 <tr>
36                     <td colspan="4" class="px-6 py-4 whitespace-nowrap
   text-center text-gray-400">No records found</td>
37                 </tr>
38             {% endfor %}
39         </tbody>
40     </table>
41 </div>
42 {% endblock %}
```

Go create that template... then paste:

templates/voyage/_row.html.twig

```
1  <tr class="even:bg-gray-700 odd:bg-gray-600">
2      <td class="px-6 py-4 whitespace-nowrap">{{ voyage.id }}</td>
3      <td class="px-6 py-4">{{ voyage.purpose }}</td>
4      <td class="px-6 py-4 whitespace-nowrap">{{ voyage.leaveAt ?
   voyage.leaveAt|date('Y-m-d H:i:s') : '' }}</td>
5      <td class="px-6 py-4 whitespace-nowrap">
6          <a href="{{ path('app_voyage_show', {'id': voyage.id}) }}"
   class="text-blue-400 hover:text-blue-600">show</a>
7          <a href="{{ path('app_voyage_edit', {'id': voyage.id}) }}"
   class="ml-4 text-yellow-400 hover:text-yellow-600">edit</a>
8      </td>
9  </tr>
```

Easy.

Copy the `include()` statement and, in `new.html.twig`, use that for the stream:

templates/voyage/new.html.twig

```
// ... lines 1 - 24
{% block stream_success %}
    <turbo-stream action="prepend" targets="#voyage-list tbody">
        <template>
            {{ include('voyage/_row.html.twig') }}
        </template>
    </turbo-stream>
{% endblock %}
```

Let's try this! Create another voyage and... beautiful! Modal closes, toast notification renders & the page updates. It's everything we want.

Tomorrow we're going to put our new modal system to the test by opening the edit link inside a modal. I promised it would be reusable, and tomorrow we'll prove it... with a few curve balls to make it more realistic.

Chapter 23: More with fun Modals! Editing & Deleting

Welcome to day 23 - the grand finale in our modal system saga. Though, we will revisit it in a few days when we talk about Twig components.

So if our new modal system is as reusable as I've promised, we should be able to easily open the edit form in a modal too, right?

Opening the Edit Form in a Modal

To opt into the modal system, the only thing we need to change - in `edit.html.twig` - is to extend `modalBase.html.twig`. And while we're here, take out the extra padding with `modal:m-0` and `modal:p-0`:

templates/voyage/edit.html.twig

```
{% extends 'modalBase.html.twig' %}
// ... lines 2 - 4
{% block body %}
    <div class="m-4 p-4 modal:m-0 modal:p-0 bg-gray-800 rounded-lg">
// ... lines 7 - 22
    </div>
{% endblock %}
```

Next, make the edit link *target* the `modal` frame. This lives in `_row.html.twig`. I'll break this onto multiple lines.... then add `data-turbo-frame="modal"`:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap">
// ... line 6
        <a
            href="{{ path('app_voyage_edit', {'id': voyage.id}) }}"
            class="ml-4 text-yellow-400 hover:text-yellow-600"
            data-turbo-frame="modal"
        >edit</a>
    </td>
</tr>
```

Moment of truth. Refresh. And... darn it! It just works! Even if we save successfully, *that* works. We get the toast, the modal closes, my goodness!

This works because, in `VoyageController`, the `edit` action, like `new`, redirects to the `index` page:

src/Controller/VoyageController.php

```
// ... lines 1 - 15
class VoyageController extends AbstractController
{
// ... lines 18 - 64
    public function edit(Request $request, Voyage $voyage, EntityManagerInterface $entityManager): Response
    {
// ... lines 67 - 69
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 71 - 74
            return $this->redirectToRoute('app_voyage_index', [], Response::HTTP_SEE_OTHER);
        }
// ... lines 77 - 81
    }
// ... lines 83 - 104
}
```

That has an empty modal frame, so the modal closes.

When the Modal Doesn't Close

But... I want to be tricky. The edit form now appears in two contexts, the modal, but also on its standalone page. What if, when we're on this page, on success, we want to redirect right back

here so we can keep editing.

Easy! Change the route to `app_voyage_edit` and set `id` to `$voyage->getId()`:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 64
65     public function edit(Request $request, Voyage $voyage,
   EntityManagerInterface $entityManager): Response
66     {
↕ // ... lines 67 - 69
70         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 71 - 74
75             return $this->redirectToRoute('app_voyage_edit', ['id' =>
   $voyage->getId()], Response::HTTP_SEE_OTHER);
76         }
↕ // ... lines 77 - 81
82     }
↕ // ... lines 83 - 104
105 }
```

Cool. Now when we save, it works! But... how did that affect the form in the modal? When we edit and save... nothing happens. The modal is still here and no toast notification.

Rendering the "Frame Streams" in all Frames

Let's work on the missing toast notification first. In `base.html.twig`, inside of the `modal` frame, we render the flash messages in a `<turbo-stream>`. The problem is... when we redirect to the edit page, because it extends `modalBase.html.twig`, the frame that's returned is *this* one. And this `<turbo-frame>` does *not* render these streams.

It turns out, these lines should really live inside *any* `<turbo-frame>` that might be rendered after a form submit.

To help with that, copy this and, inside the `templates/` directory, create a new file called `_frameSuccessStreams.html.twig`. Paste inside:

templates/_frameSuccessStreams.html.twig

```
<turbo-stream action="append" target="flash-container">
    <template>{{ include('_flashes.html.twig') }}</template>
</turbo-stream>
{% for stream in app.flashes('stream') %}
    {{ stream|raw }}
{% endfor %}
```

But before we use this, I want to add one other detail: `if app.request.headers.get('turbo-frame')` equals a new `frame` variable, then render this, else, do nothing:

templates/_frameSuccessStreams.html.twig

```
{% if app.request.headers.get('turbo-frame') == frame %}
    <turbo-stream action="append" target="flash-container">
        <template>{{ include('_flashes.html.twig') }}</template>
    </turbo-stream>
    {% for stream in app.flashes('stream') %}
        {{ stream|raw }}
    {% endfor %}
{% endif %}
```

I'm coding for an edge-case, so let me explain. Imagine we have *two* `<turbo-frame>` elements on the same page: `id="login"` and `id="registration"`. And they both include this partial. In this case, the `<turbo-frame id="login">` would always render the flash messages... leaving nothing for the poor `registration` frame. And so, when we *are* submitting inside the `registration` Turbo Frame... we wouldn't see the toast notifications.

To fix this, when we use this partial - `include('_frameSuccessStreams.html.twig')` - pass the name of the frame you're inside: `modal`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div
// ... lines 53 - 54
        >
            <dialog
// ... lines 57 - 59
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
// ... lines 64 - 67
                        >
                            {{ include('_frameSuccessStreams.html.twig', { frame: 'modal' }) }}
                        </turbo-frame>
                    </div>
                </div>
            </dialog>
// ... lines 74 - 89
        </div>
// ... lines 91 - 94
    </body>
</html>
```

That way, if the current frame is something *else*, this won't eat the flash messages.

Copy this, and in `modalFrame.html.twig`, paste that here too:

templates/modalFrame.html.twig

```
<turbo-frame id="modal">
    {% block body %}{% endblock %}
    {{ include('_frameSuccessStreams.html.twig', { frame: 'modal' }) }}
</turbo-frame>
```

Let's do this! Refresh, Edit... and save. The modal still stays open, but look back there: we see the toast!

Closing the Modal when it wants to stay open

Now: how can we close this pesky modal. When we put a form inside a frame, our Symfony controller might not need to change. Flash messages will work and, depending on where you redirect, the modal might even close.

But you *do* need to ask yourself: where are all the places my form will be used? And: am I returning the right response for each situation? Right now, in the modal situation, our response *isn't* what we want: it *doesn't* cause the modal to close.

And that's okay! Remember: in addition to letting the Turbo frame update with the content after the redirect, we can *also* use streams to do anything extra.

In `new.html.twig`, steal the `stream_success` from the bottom. In `edit.html.twig`, paste. This time, we want to update the `<turbo-frame id="modal">` element to *empty* its content so the modal will close. Do that with `action="update"`, `target="modal"`, and set the `<template>` to nothing:

templates/voyage/edit.html.twig

```
↕ // ... lines 1 - 25
26 {% block stream_success %}
27     <turbo-stream action="update" target="modal">
28         <template></template>
29     </turbo-stream>
30 {% endblock %}
```

In the controller, to add the "extra stuff", copy the if statement from `new`... paste it down here, change the template to `edit.html.twig` and... we should be good!

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 15
16 class VoyageController extends AbstractController
17 {
↕ // ... lines 18 - 64
65     public function edit(Request $request, Voyage $voyage,
   EntityManagerInterface $entityManager): Response
66     {
↕ // ... lines 67 - 69
70         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 71 - 73
74             if ($request->headers->has('turbo-frame')) {
75                 $stream = $this->renderBlockView('voyage/edit.html.twig',
   'stream_success', [
76                     'voyage' => $voyage
77                 ]);
78
79                 $this->addFlash('stream', $stream);
80             }
↕ // ... lines 81 - 82
83         }
↕ // ... lines 84 - 88
89     }
↕ // ... lines 90 - 111
112 }
```

Ok, hit "Edit" and save. Hmm, I saw the toast, but the modal didn't close. Let me look at the stream to make sure I have everything. Ah! With `targets`, you use a CSS selector. But with `target`, it's just the id:

templates/voyage/edit.html.twig

```
↕ // ... lines 1 - 25
26 {% block stream_success %}
27     <turbo-stream action="update" target="modal">
↕ // ... line 28
29     </turbo-stream>
30 {% endblock %}
```

So the Turbo Stream was executing... but wasn't matching anything.

Let's try that again. When we hit save, that will redirect back to the edit page, and that *is* going have a `<turbo-frame id="modal">` *with* content: it *won't* be empty. But then, our *stream* should empty it and the modal should close.

And... gorgeous!

Updating the Row in Edit

Can I add one last polishing detail to edit? It would be *so* cool if, when we change a voyage, it updated the row instantly. This is another "extra", and... it's going to be easy.

First, to target this, in `_row.html.twig`, add an `id`, `voyage-list-item-`, `{{ voyage.id }}`:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 12
</tr>
```

Copy that, head over to `edit.html.twig` and add one more Turbo Stream: `action="replace"` and `target="voyage-list-item-"` `voyage.id`. Add the `<template>` and then include `voyage/_row.html.twig`:

templates/voyage/edit.html.twig

```
// ... lines 1 - 25
{% block stream_success %}
// ... lines 27 - 29
    <turbo-stream action="replace" target="voyage-list-item-{{ voyage.id }}">
        <template>{{ include('voyage/_row.html.twig') }}</template>
    </turbo-stream>
{% endblock %}
```

This is where things *really* start to shine. Edit, remove those exclamation points and... the page updates instantly. Our edit modal - even with all the complications I threw in - is done!

Handling Delete

With our last 3 minutes, let's make sure the "delete" button is working. Oh... it is! The modal closes and the toast renders! Like the other actions, after deleting, it redirects to the `index` page and the empty `modal` frame closes the modal. It's brilliant!

Except... that the row I deleted is *still* there until we refresh.

But hold up. The delete button is a form that submits. And the only reason this submits into a `<turbo-frame>` is because it happens to live inside the modal frame.

But the delete action doesn't *need* to submit into a frame. We're never going to click "Delete" then want to *show* something in the modal. A full page navigation would be *fine*.

To do that, in `_delete_form.html.twig`, on the frame, add `data-turbo-frame="_top"`:

templates/voyage/_delete_form.html.twig

```
1  <form method="post" data-turbo-frame="_top" action="{{
   path('app_voyage_delete', {'id': voyage.id}) }}" onsubmit="return
   confirm('Are you sure you want to delete this item?');">
↕  // ... lines 2 - 5
6  </form>
```

Now, edit, delete, and... the redirect causes a full page navigation, which is *fine*.

Extra-Fancy Delete

Though, yes, it *could* be smoother. Scroll down a bit... and delete one. The page scrolls back to the top.

Like with anything, if this is important to us, we can improve it. Remove the `data-turbo-frame="_top"`:

templates/voyage/_delete_form.html.twig

```
1  <form method="post" action="{{ path('app_voyage_delete', {'id':
   voyage.id}) }}" onsubmit="return confirm('Are you sure you want to delete
   this item?');">
↕  // ... lines 2 - 5
6  </form>
```

When a form - even our delete form - exists inside a `<turbo-frame>`, we need to ask: where is this being used and what do I need to update to make the page *perfect* after success? In this case, we need to remove the row. So we need to do something *extra*, outside the frame. And we know how to do that!

In `edit.html.twig`, steal the `stream_success` block. Then create a new template called `delete.html.twig`. Delete doesn't normally have its own template... and we're going to use this just for the `stream_success`. Use this one, change `action` to `remove` and `target` `voyage-list-item-` but just use an `id` variable. And for remove, we don't need the `<template>` at all:

templates/voyage/delete.html.twig

```twig
{% block success_stream %}
    <turbo-stream action="remove" target="voyage-list-item-{{ id }}"></turbo-stream>
{% endblock %}
```

In `VoyageController`, scroll up, steal the if statement.... and down in delete, paste that. Change the template to `delete.html.twig` and pass an `id` variable set to `$id`. We can't use `$voyage->getId()` because it'll already be empty since we deleted it. Instead, pass `$id`... and before we delete, set that: `$id = $voyage->getId()`:

src/Controller/VoyageController.php

```php
// ... lines 1 - 15
class VoyageController extends AbstractController
{
// ... lines 18 - 91
    public function delete(Request $request, Voyage $voyage, EntityManagerInterface $entityManager): Response
    {
        if ($this->isCsrfTokenValid('delete'.$voyage->getId(), $request->request->get('_token'))) {
            $id = $voyage->getId();
// ... lines 96 - 100
            if ($request->headers->has('turbo-frame')) {
                $stream = $this->renderBlockView('voyage/delete.html.twig', 'success_stream', [
                    'id' => $id,
                ]);

                $this->addFlash('stream', $stream);
            }
        }
// ... lines 109 - 110
    }
// ... lines 112 - 120
}
```

Let's do this! Scroll way down here and delete ID 22. Watch. Boom. The row is gone, we get the toast notification and the page doesn't budge.

Ok, the last few days have been... wow. Tomorrow, we're going to take it easier and learn one other way we can use Turbo Streams. See you then!

Chapter 24: Turbo Stream Responses

For day 24, strap in for a quick adventure. We've learned that Turbo Streams are custom HTML elements that you can throw onto the page *anywhere*... and they execute! But there's another way to use Streams that's actually more commonly-documented, even if I'm using it a bit less lately.

In `VoyageController`, scroll up to find the `new()` action. Instead of redirecting, like we normally do for a form submit, the other option is to return a response that is *entirely* filled with Turbo streams.

Returning a Response of Streams

Watch: remove the flash and *return* `$this->renderBlockView()`... except change it to `renderBlock()`. That does the same thing, but returns a `Response` object instead of a string. The last detail is `$request->setRequestFormat()` `TurboBundle::STREAM_FORMAT`:

src/Controller/VoyageController.php

```
↕ // ... lines 1 - 13
14 use Symfony\UX\Turbo\TurboBundle;
↕ // ... lines 15 - 16
17 class VoyageController extends AbstractController
18 {
↕ // ... lines 19 - 27
28     public function new(Request $request, EntityManagerInterface $entityManager): Response
29     {
↕ // ... lines 30 - 33
34         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 35 - 39
40             if ($request->headers->has('turbo-frame')) {
41                 $request->setRequestFormat(TurboBundle::STREAM_FORMAT);
42
43                 return $this->renderBlock('voyage/new.html.twig', 'stream_success', [
44                     'voyage' => $voyage
45                 ]);
46             }
↕ // ... lines 47 - 48
49         }
↕ // ... lines 50 - 54
55     }
↕ // ... lines 56 - 121
122 }
```

It's a bit techy, but this will set a `Content-Type` header on the response that tells Turbo:

> *"Hey! This is not a normal full page response. I'm returning just a set of Turbo Streams that I want you to process."*

Drumroll, please. Refresh, go to New Voyage... fill out the fields... and save. What happened? The modal is still open and no Toast notification. But if you were watching closely, the row in the table *did* prepend!

In the network tools, find the POST request. Look at that! The response is nothing more than the `<turbo-stream>`: that's the only thing our app returned.

Returning All the Streams Needed

The takeaway is: because we're not redirecting to another page, we no longer get the normal `<turbo-frame>` behavior where it finds the frame on the next page and renders that. In our case, the empty `<turbo-frame>` is what closed the modal *and* rendered the flash messages.

When you decide to return a stream response, you are 100% responsible for updating *everything* on the page. So, in `new.html.twig`, down here, we need a couple more streams! Open `edit.html.twig` and steal the one that closes the modal. Pop that here.... then, from `_frameSuccessStreams.html.twig`, steal the stream that appends to the flash container:

templates/voyage/new.html.twig

```
// ... lines 1 - 24
{% block stream_success %}
    <turbo-stream action="prepend" targets="#voyage-list tbody">
        <template>
            {{ include('voyage/_row.html.twig') }}
        </template>
    </turbo-stream>
    <turbo-stream action="update" target="modal">
        <template></template>
    </turbo-stream>
    <turbo-stream action="append" target="flash-container">
        <template>{{ include('_flashes.html.twig') }}</template>
    </turbo-stream>
{% endblock %}
```

I think that's all we need! Give this another shot. Here's our toast notification finally from the *previous* submit. Create a new voyage... and ... save. That's it! Toast notification, modal closed, row prepended.

Turbo Mercure

This idea of returning *just* a `<turbo-stream>` is similar to how the Turbo and Mercure integration works. If you don't know, Mercure is a tool that allows you to get real-time updates on your front end... kind of like web sockets, but cooler. And Mercure pairs really well with Turbo. From inside your controller, you publish an `Update` to Mercure... which will be sent to the frontends of every browser that's listening to this `chat` topic.

The content of that `Update` is a set of Turbo Streams. I'll scroll down to that template. So we publish streams... those streams are sent to frontend via Mercure, and Turbo processes them.

On the frontend, it might look like this. We edit a voyage, add a few exclamation points and hit save. Of course, *our* page updates thanks to the normal Turbo mechanisms we've talked about. But, if we were using Mercure, we could make it so that anyone *else* on this page could receive a Stream update that *also* says to prepend this row. So I add the exclamation points, and *you* suddenly *also* see them on your screen, without refreshing.

It's *super* cool and powered via Streams.

Ok, even though this is working nicely, let's go back to our old way... which was *also* working nicely. Remove the new Turbo Streams... and undo the code in the controller.

Tomorrow, we move on to one of my *favorite* parts of LAST Stack - and the key to organizing your site into reusable chunks: Twig Components.

Chapter 25: Twig Components

Today, we get to talk about one of my favorite new-ish PHP libraries: Twig Components. They... do kind of what their name sounds like. But let's dive in and *see* them in action.

Installing Twig Components

Find your terminal and install the package with:

```
composer require symfony/ux-twig-component
```

Twig Components is a pure PHP library... and an easy way to think about it is: a fancier and more powerful way to do a Twig `include()`.

Over in our browser, open the edit page in a new tab so we can see the full page. Then open the form for this: `_form.html.twig`. When you use Tailwind, creating a button is... kind of a lot of work. Twig Components will help us centralize this.

make:twig-component

Because this is our first Twig Component, let's be lazy and generate it. Run:

```
php bin/console make:twig-component
```

Call it Button... and say no to a live component. We get to talk about *those* in 2 days.

This created two files. The first lives in `src/Twig/Components/Button.php`:

src/Twig/Components/Button.php

```
// ... lines 1 - 2
namespace App\Twig\Components;

use Symfony\UX\TwigComponent\Attribute\AsTwigComponent;

#[AsTwigComponent]
class Button
{

}
```

It's... an empty class. And it's not even needed yet! In fact, we could *delete* this and the first half of today would work fine without it. We'll come back to this later.

The more important thing is: `templates/components/Button.html.twig`. A pretty boring-looking Twig template. Change the div to be a `<button>`, and inside, I'll say, "Press me!":

templates/components/Button.html.twig

```
<button {{ attributes }}>Press me!</button>
```

To use this, over in `_form.html.twig`, say `{{ component('Button') }}`:

templates/voyage/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 3
    {{ component('Button', {
// ... lines 5 - 6
    }) }}
// ... lines 8 - 11
{{ form_end(form) }}
```

If we *just* did that, it would work. We get a button that says, "press me".

Passing Attributes to a Component

One of the first interesting things about Twig Components is that you can pass attributes into them. As a second argument, pass `formnovalidate` set to `true`, then `class`... copy this long class list... and paste:

templates/voyage/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 3
    {{ component('Button', {
        formnovalidate: true,
        class: 'px-4 py-2 border border-transparent text-sm font-medium rounded-md text-white bg-green-600 hover:bg-green-700',
    }) }}
// ... lines 8 - 11
{{ form_end(form) }}
```

When we do that, we get an error... because I forgot my closing comma. Better. As I was saying, when we do that... we see a button with those Tailwind classes! This is thanks to a cool `attributes` variable that we have in any Twig Component template. It collects what we pass into the component - called `props` - and renders them.

The Optional HTML Syntax

One of my *favorite* features of Twig Components is that it has an optional, but wonderful, HTML syntax. Instead of the Twig function, we can say `<twig:Button>`. Now props are passed like normal HTML attributes. I'll copy them from the real `<button>` tag and paste:

templates/voyage/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 3
    <twig:Button
        formnovalidate
        class="px-4 py-2 border border-transparent text-sm font-medium rounded-md text-white bg-green-600 hover:bg-green-700"
    />
// ... lines 8 - 11
{{ form_end(form) }}
```

What does it look like? The same darn thing! This special syntax comes from Twig Components and is for *rendering* Twig Components. Some people are "meh" on this syntax, while others *love* it. Choose whatever you want. I like it because it *feels* like a native HTML element. And if you've ever used a front-end framework like React, it will feel natural.

Passing Content to the Twig Component

But, we still have hard-coded "Press me!" content. That's not very helpful. To make this dynamic, we can use a block. That's right, a good old-fashioned Twig block! I called this one `content`:

templates/components/Button.html.twig

```
1  <button {{ attributes }}>{% block content %}{% endblock %}</button>
```

To pass *in* that block, copy the button label below, change this to a *not* self-closing tag, paste... then add the closing tag:

templates/voyage/_form.html.twig

```
1  {{ form_start(form) }}
↕  // ... lines 2 - 3
4      <twig:Button
5          formnovalidate
6          class="px-4 py-2 border border-transparent text-sm font-medium
   rounded-md text-white bg-green-600 hover:bg-green-700"
7      >
8          {{ button_label|default('Save') }}
9      </twig:Button>
↕  // ... lines 10 - 13
14 {{ form_end(form) }}
```

And... it works! What!? When you put content between the Twig component HTML tags, it becomes a block called `content`. That's just built in. If you also had other blocks in your component and needed to pass *those* in too, you can do that. And you would specify those using the normal `block`, `endblock` syntax. But you get this `content` block for free, which *looks* fantastic.

Celebrate by removing our old HTML button:

templates/voyage/_form.html.twig

```
1  {{ form_start(form) }}
2      {{ form_widget(form) }}
3
4      <twig:Button
5          formnovalidate
6          class="px-4 py-2 border border-transparent text-sm font-medium
   rounded-md text-white bg-green-600 hover:bg-green-700"
7      >
8          {{ button_label|default('Save') }}
9      </twig:Button>
10 {{ form_end(form) }}
```

Default Component Attributes

But remember: the goal is to make buttons easier to create. And needing to specify all of these classes is... *entirely* the problem we want to fix! Copy those and delete the `class` attribute entirely:

templates/voyage/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 3
    <twig:Button formnovalidate>
        {{ button_label|default('Save') }}
    </twig:Button>
{{ form_end(form) }}
```

In the component template, we *could* add a `class` attribute right here and paste. But instead, call `attributes.defaults`, pass that an array with `class:` then the class string:

templates/components/Button.html.twig

```
<button {{ attributes.defaults({
    class: 'px-4 py-2 border border-transparent text-sm font-medium rounded-md text-white bg-green-600 hover:bg-green-700',
}) }}>{% block content %}{% endblock %}</button>
```

This will let us add *more* classes when we *use* this component. We'll do that in minute.

Over on the site... it still looks great! Now suppose, in this one situation, we need an extra class - `hover:animate-wiggle` - to make our button more fun:

templates/voyage/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 3
    <twig:Button formnovalidate class="hover:animate-wiggle">
// ... line 5
    </twig:Button>
{{ form_end(form) }}
```

This is a custom CSS animation I invented... so down in `tailwind.config.js`, I'll paste the `wiggle`... and its keyframe:

tailwind.config.js

```
// ... lines 1 - 3
module.exports = {
// ... lines 5 - 9
  theme: {
    extend: {
      animation: {
// ... line 13
        wiggle: 'wiggle 1s ease-in-out infinite',
      },
      keyframes: {
// ... lines 17 - 20
        wiggle: {
          '0%, 100%': { transform: 'rotate(-3deg)' },
          '50%': { transform: 'rotate(3deg)' },
        }
      },
    },
  },
// ... lines 28 - 33
}
```

Ok, refresh and hover! Pointless... but so fun! The important part is that we see the normal classes that come from the component template *and* the extra class at the end.

Passing Variables to a Component

Could we now reuse the component for the delete button? Let's try! This lives in `_delete_form.html.twig`. Change this to `<twig:` big "B" `Button`. Then most of these classes are in the component already. We only need the ones related to color:

templates/voyage/_delete_form.html.twig

```
<form method="post" action="{{ path('app_voyage_delete', {'id': voyage.id}) }}" onsubmit="return confirm('Are you sure you want to delete this item?');">
// ... lines 2 - 3
    <twig:Button class="text-white bg-red-600 hover:bg-red-700 focus:ring-4 focus:ring-red-300 focus:outline-none">
        Delete
    </twig:Button>
</form>
```

And... it works! But... kind of by accident. If we inspect that element, it has the `bg-green-600` from the twig component template *and* the `bg-red-600`. You might think... that makes sense! The later one overrides the earlier one right?

Actually, no. There's no rule that says that this one should win over this one or the green should win over the red. The reason red is winning is... luck! By chance, when Tailwind generated the CSS file, the `bg-red-600` was, apparently, generated *later* in the file. So, it's winning. In Tailwind, you need to avoid competing classes because the result isn't guaranteed.

What we really want to do is create different *variants* of the button. I'd like to be able to say something like `variant="danger"`:

templates/voyage/_delete_form.html.twig

```
1  <form method="post" action="{{ path('app_voyage_delete', {'id': voyage.id}) }}" onsubmit="return confirm('Are you sure you want to delete this item?');">
↕  // ... lines 2 - 3
4      <twig:Button variant="danger" class="text-white bg-red-600 hover:bg-red-700 focus:ring-4 focus:ring-red-300 focus:outline-none">
↕  // ... line 5
6      </twig:Button>
7  </form>
```

And... over in the other template, `variant="success"`:

templates/voyage/_form.html.twig

```
1  {{ form_start(form) }}
↕  // ... lines 2 - 3
4      <twig:Button
↕  // ... line 5
6          variant="success"
↕  // ... line 7
8      >
↕  // ... line 9
10     </twig:Button>
11 {{ form_end(form) }}
```

Right now, no surprise, that adds a `variant="danger"` attribute. That's not what I want: I want to *use* this value in my component to conditionally render different classes.

This is *finally* where our PHP class comes in handy. To convert a prop that we pass into our component from an attribute to a *variable*, we can add a public property with the same name: `public string $variant = 'default';`:

src/Twig/Components/Button.php

```
// ... lines 1 - 6
#[AsTwigComponent]
class Button
{
    public string $variant = 'default';
}
```

And now that we have a public property called `variant`, we get a local variable inside of Twig called `variant`. Watch `{{ variant }}`:

templates/components/Button.html.twig

```
<button {{ attributes.defaults({
    class: 'px-4 py-2 border border-transparent text-sm font-medium rounded-md text-white bg-green-600 hover:bg-green-700',
}) }}>{{ variant }}{% block content %}{% endblock %}</button>
```

And now... we see it in both spots! Danger up here, success down there.

Adding a Component PHP Method

Ok: we now need to use the `variant` variable to conditionally render different classes. We need... kind of a switch-case statement to map each variant to a set of classes. Writing something like that in Twig... isn't super fun.

But remember: we have a Twig component PHP class that's *bound* to this template. And we can add methods there! I'll paste in a new public function called `getVariantClasses()`:

src/Twig/Components/Button.php

```
// ... lines 1 - 7
class Button
{
// ... lines 10 - 11
    public function getVariantClasses(): string
    {
        return match ($this->variant) {
            'default' => 'text-white bg-blue-500 hover:bg-blue-700',
            'success' => 'text-white bg-green-600 hover:bg-green-700',
            'danger' => 'text-white bg-red-600 hover:bg-red-700 focus:ring-4 focus:ring-red-300 focus:outline-none',
            default => throw new \LogicException(sprintf('Unknown button type "%s"', $this->variant)),
        };
    }
}
```

It has a `match` statement... which based on `$this->variant`, returns a different set of classes.

One of the superpowers of Twig components is that this `Button` object is available inside the component template as a variable called `this`. That means we can go to the end of the `class` list, remove the color-specific ones, then concatenate with a `~` and `this.variantClasses`:

templates/components/Button.html.twig

```
<button {{ attributes.defaults({
    class: 'px-4 py-2 border border-transparent text-sm font-medium rounded-md '~this.variantClasses,
}) }}>{% block content %}{% endblock %}</button>
```

Go check it. Yes! We have green down here... and red up there! To *really* prove it's working, on the delete button, remove the extra classes:

templates/voyage/_delete_form.html.twig

```
<form method="post" action="{{ path('app_voyage_delete', {'id': voyage.id}) }}" onsubmit="return confirm('Are you sure you want to delete this item?');">
// ... lines 2 - 3
    <twig:Button variant="danger">
        Delete
    </twig:Button>
</form>
```

I love the way that looks in code... and on the site.

Pointing Tailwind at your Component PHP Classes

Though, one detail. Tailwind scans our source files, finds all the Tailwind classes we're using and includes those in the final CSS file. And because we're now including classes inside PHP, we need to make sure Tailwind sees those.

In `tailwind.config.js`, on top, I'll paste in one more line to make it scan our Twig component PHP classes:

```
tailwind.config.js
↕ // ... lines 1 - 3
4 module.exports = {
5   content: [
↕ // ... lines 6 - 8
9     "./src/Twig/Components/**/*.php"
10  ],
↕ // ... lines 11 - 34
35 }
```

Changing the Component Tag Name

Ok, we're *nearly* done for today - but I want to celebrate and use the new component in one more spot: on the voyage index page, for the "New Voyage" button.

Open `index.html.twig`... change this to a `<twig:Button>`... then we can remove most of these classes. The bold *is* specific to this spot I think:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          class="flex justify-between"
9      >
↕ // ... lines 10 - 11
12         <twig:Button
13             href="{{ path('app_voyage_new') }}"
14             data-turbo-frame="modal"
15             class="flex items-center space-x-1 font-bold"
16         >
17             <span>New Voyage</span>
18             <svg xmlns="http://www.w3.org/2000/svg" class="w-4 inline"
   viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none"
   stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0
   0h24v24H0z" fill="none"/><path d="M3 12a9 9 0 1 0 18 0a9 9 0 0 0 -18 0" />
   <path d="M9 12h6" /><path d="M12 9v6" /></svg>
19         </twig:Button>
20     </div>
↕ // ... lines 21 - 40
41 </div>
42 {% endblock %}
```

When we refresh... it renders! Though... when I click... nothing happens! You probably saw why: this is now a *button*, not an `a` tag.

That's okay: we can make our component just a *bit* more flexible. In `Button.php`, add another property: `string $tag` defaulting to `button`:

src/Twig/Components/Button.php

```
↕ // ... lines 1 - 7
8  class Button
9  {
↕ // ... line 10
11     public string $tag = 'button';
↕ // ... lines 12 - 21
22 }
```

Then in the template, use `{{ tag }}` for the starting tag and the ending tag:

templates/components/Button.html.twig

```
1  <{{ tag }} {{ attributes.defaults({
↕ // ... line 2
3  }) }}>{% block content %}{% endblock %}</{{ tag }}>
```

Finish in `index.html.twig`: pass `tag="a"`:

templates/voyage/index.html.twig

```
↕ // ... lines 1 - 5
6  <div class="m-4 p-4 bg-gray-800 rounded-lg">
7      <div
8          class="flex justify-between"
9      >
↕ // ... lines 10 - 11
12         <twig:Button
13             tag="a"
↕ // ... lines 14 - 16
17         >
↕ // ... lines 18 - 19
20         </twig:Button>
21     </div>
↕ // ... lines 22 - 41
42 </div>
43 {% endblock %}
```

Now over here... when we click... got it!

That's it! I hope you love Twig components as much as I do. But they can do even more! I didn't tell you how you can prefix any attribute with `:` to pass dynamic Twig variables or expressions to a prop. We also didn't discuss that the component PHP classes are *services*. Yea, you can add an `__construct()` function, autowire other services, like `VoyageRepository`, then use those to provide data to your template... making the entire component standalone and self-sufficient. That's one of my favorite features.

Tomorrow we're going to keep leveraging Twig components to create a modal component... then see just how easily we can use modals whenever we want.

Chapter 26: Modal Twig Component

Today is a good day. Today we get to combine our modal system with Twig components to achieve a goal! I want to be able to quickly add a modal *anywhere* in our app.

Creating the Modal Component

Start in `base.html.twig`. All the way at the bottom, copy the modal markup. You can see... it's quite a bit: not something we want to copy and paste somewhere else:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 51
        <div
            data-controller="modal"
            data-action="turbo:before-cache@window->modal#close"
        >
            <dialog
                class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
                data-modal-target="dialog"
                data-action="close->modal#close click->modal#clickOutside"
            >
                <div class="flex grow p-5">
                    <div class="grow overflow-auto p-1">
                        <turbo-frame
                            id="modal"
                            data-modal-target="dynamicContent"
                            data-action="turbo:before-fetch-request->modal#showLoading"
                            class="aria-busy:opacity-50 transition-opacity"
                        >
                            {{ include('_frameSuccessStreams.html.twig', { frame: 'modal' }) }}
                        </turbo-frame>
                    </div>
                </div>
            </dialog>

            <template data-modal-target="loadingTemplate">
                <div class="bg-space-pattern bg-cover rounded-lg p-8">
                    <div class="space-y-2">
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
```

```
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                    </div>
                </div>
            </template>
        </div>
// ... lines 91 - 94
    </body>
</html>
```

Copy, then delete it. Let's craft a `Modal` component, this time by hand. Create a new file in `templates/components/` called `Modal.html.twig`, and paste:

templates/components/Modal.html.twig

```
<div
    data-controller="modal"
    data-action="turbo:before-cache@window->modal#close"
>
    <dialog
        class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] md:min-w-[50%] animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
        data-modal-target="dialog"
        data-action="close->modal#close click->modal#clickOutside"
    >
        <div class="flex grow p-5">
            <div class="grow overflow-auto p-1">
                <turbo-frame
                    id="modal"
                    data-modal-target="dynamicContent"
                    data-action="turbo:before-fetch-request->modal#showLoading"
                    class="aria-busy:opacity-50 transition-opacity"
                >
                    {{ include('_frameSuccessStreams.html.twig', { frame: 'modal' }) }}
                </turbo-frame>
            </div>
        </div>
    </dialog>

    <template data-modal-target="loadingTemplate">
        <div class="bg-space-pattern bg-cover rounded-lg p-8">
            <div class="space-y-2">
                <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                <div class="h-4"></div>
                <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                <div class="h-4"></div>
                <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
            </div>
        </div>
    </template>
</div>
```

Like I said with the `Button`, we don't *need* a PHP class for a component. Because we don't have one, we call this an "anonymous component".

On top, render `attributes`... then add `.defaults()` so we can move these two attributes *into* that. Paste... then each of these needs a makeover to fit as Twig keys and values instead of HTML attributes:

templates/components/Modal.html.twig

```
<div
    {{ attributes.defaults({
        'data-controller': 'modal',
        'data-action': 'turbo:before-cache@window->modal#close',
    }) }}
>
// ... lines 7 - 40
</div>
```

I like it! Over in `base.html.twig`, render this with `<twig:Modal>`. Easy!

Adding Blocks to the Component

However, look closer at `Modal.html.twig`: there are some things that *shouldn't* be here. For example, the `<turbo-frame>`! Not every modal needs a frame. A lot of times, we'll render a modal with simple, hardcoded content inside.

Copy this, and replace it with, of course, `{% block content %}` and `{% endblock %}`:

templates/components/Modal.html.twig

```
<div
// ... lines 2 - 5
>
    <dialog
// ... lines 8 - 10
    >
        <div class="flex grow p-5">
            <div class="grow overflow-auto p-1">
                {% block content %}{% endblock %}
            </div>
        </div>
    </dialog>
// ... lines 18 - 33
</div>
```

In `base.html.twig`, paste the frame... and add a closing tag:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 54
        <twig:Modal>
            <turbo-frame
                id="modal"
                data-modal-target="dynamicContent"
                data-action="turbo:before-fetch-request->modal#showLoading"
                class="aria-busy:opacity-50 transition-opacity"
            >
                {{ include('_frameSuccessStreams.html.twig', { frame: 'modal' }) }}
            </turbo-frame>
        </twig:Modal>
    </body>
</html>
```

Let's keep going! The loading template down here? Yea, that's also something that *specific* to this *one* modal. If our modal is a hardcoded message, it won't need this at all!

Copy the loading `div`, delete, then around the `<template>` add: if `block('loading_template')`:

templates/components/Modal.html.twig

```
<div
// ... lines 2 - 5
>
// ... lines 7 - 18
    {% if block('loading_template') %}
        <template data-modal-target="loadingTemplate">
            {% block loading_template %}{% endblock %}
        </template>
    {% endif %}
</div>
```

So *if* we pass the block, render it inside the `<template>`.

Back in `base.html.twig`, anywhere, define that block. But instead of the normal `{% block %}` tag - which *would* work - when you're inside a Twig component, you can use a

special `<twig:block>` syntax with `name="loading_template"`. Then, paste:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
// ... lines 17 - 54
        <twig:Modal>
            <turbo-frame
// ... lines 57 - 62
            </turbo-frame>
            <twig:block name="loading_template">
                <div class="bg-space-pattern bg-cover rounded-lg p-8">
                    <div class="space-y-2">
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                        <div class="h-4 bg-gray-700 rounded w-3/4 animate-pulse"></div>
                        <div class="h-4"></div>
                        <div class="h-4 bg-gray-700 rounded w-1/2 animate-pulse"></div>
                    </div>
                </div>
            </twig:block>
        </twig:Modal>
    </body>
</html>
```

We just moved around a *lot* of stuff. And yet... the existing modal still works! And now, we have a leaner, meaner modal component that we can *reuse* in other places.

Delete Modal with Custom Content

Let's do exactly that. I want to add a delete link on each row that, on click, opens a modal with a confirmation. Open `_row.html.twig`. Copy the edit link, paste, and call it delete:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap">
// ... lines 6 - 11
        <a
            href="{{ path('app_voyage_edit', {'id': voyage.id}) }}"
            class="ml-4 text-yellow-400 hover:text-yellow-600"
            data-turbo-frame="modal"
        >edit</a>
    </td>
</tr>
```

To get this to work, one option is to create a new, standalone delete confirmation page, point to that and... done! The `data-turbo-frame="modal"` would load that page into the modal.

But since we've done that before, let's try something different. Delete the `href`, change this to a `button`, remove the `data-turbo-frame` attribute... and change the yellow colors to red:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap">
// ... lines 6 - 11
        <button
            class="ml-4 text-red-400 hover:text-red-600"
        >delete</button>
    </td>
</tr>
```

Let's go check out the look. Nice!

Back over, I'll paste in a modal:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap">
// ... lines 6 - 14
        <twig:Modal>
            <svg class="mx-auto mb-4 text-gray-400 w-12 h-12 dark:text-gray-200" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M12 12m-9 0a9 9 0 1 0 18 0a9 9 0 1 0 -18 0" /><path d="M12 9v4" /><path d="M12 16v.01" /></svg>

            <h3 class="mb-5 text-lg font-normal text-gray-500 dark:text-gray-400">
                Delete this thrilling voyage???
            </h3>
        </twig:Modal>
    </td>
</tr>
```

There's nothing special here. The big difference is, instead of a `<turbo-frame>`, the content we need is *right* here. When we refresh, *every* row now has a delete dialog inside of it. But that's totally okay! It's not open, so it's invisible.

Opening the Modal

Now for the tricky part. When we click this button, we need to open this modal. To make this happen, we need the button to physically live *inside* the `data-controller="modal"` element so that it can call the `open` action on the modal Stimulus controller.

To allow that, inside the modal template, add a new block called `trigger`, `endblock`:

templates/components/Modal.html.twig

```
<div
// ... lines 2 - 5
>
    {% block trigger %}{% endblock %}
// ... lines 8 - 25
</div>
```

Now, if you have a button that triggers the modal to open, you can put that right here! Over in `_row.html.twig`, copy the button, remove it, say `<twig:block name="trigger">` and paste.

And because we're inside the modal, add `data-action="modal#open"`:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap flex">
// ... lines 6 - 12
        <twig:Modal>
            <twig:block name="trigger">
                <button
                    class="ml-4 text-red-400 hover:text-red-600"
                    data-action="modal#open"
                >delete</button>
            </twig:block>
// ... lines 20 - 25
        </twig:Modal>
    </td>
</tr>
```

Let's try this! So excited! Refresh! The styling got weird. Before, we had three `a` tags, which are inline elements. Now we have two inline elements and a block element. So that *is* something that changes slightly, but it's an easy fix. Up on the `<td>`, add a `flex` class:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap flex">
// ... lines 6 - 26
    </td>
</tr>
```

Modal Conditional Size & the props Tag

And now... much better. Most importantly, when we hit Delete, the modal opens! However, you know me, I want this to be *perfect*. And I'm not happy with how *big* this is. When I open the edit

form, it makes sense to be half the screen width. But when I open the delete, we should let it shrink down to the size of the content inside.

To do that, in this one case, I want to be pass a new flag called `allowSmallWidth` set to `true`:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap flex">
// ... lines 6 - 12
        <twig:Modal :allowSmallWidth="true">
// ... lines 14 - 25
        </twig:Modal>
    </td>
</tr>
```

I added this `:` because, if I pass `allowSmallWidth="true"`, that will pass the *string* `true`. By adding a colon, this becomes Twig code, so that will pass the *Boolean* `true`. They would both work... but I like being stricter.

With the `Button`, we learned that if you want this to become a *variable* instead of an attribute, you can add a public property with that same name. And we *could* create a new `Modal.php` file.

But there's another way to convert from an attribute into a variable when using an anonymous component. At the top of `Modal.html.twig`, add a `props` tag that's special to Twig components. Add `allowSmallWidth` and default it to `false`:

templates/components/Modal.html.twig

```
{% props allowSmallWidth=false %}
// ... lines 2 - 28
```

Cool, huh? Below, we want to make this min-width conditional. Say `{{ allowSmallWidth }}` - if that is true, render nothing, else render the `md:min-w-[50%]`:

templates/components/Modal.html.twig

```
{% props allowSmallWidth=false %}
<div
// ... lines 3 - 6
>
// ... lines 8 - 9
    <dialog
        class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] {{ allowSmallWidth ? '' : 'md:min-w-[50%] ' }}animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80"
// ... lines 12 - 13
    >
// ... lines 15 - 19
    </dialog>
// ... lines 21 - 26
</div>
```

Back on the page, the edit link still opens with half width... but that delete link, ah, it's nice and small! Now it deserves some real content! In `_row.html.twig`, after the `<h3>`, I'll add some styling... then I want a cancel button that closes the modal. For that, we can go old-school. Add a `<form method="dialog">`, and inside a `<twig:Button>` that says Cancel. And I want the button to look like a link, so add `variant="link"`:

templates/voyage/_row.html.twig

```
<tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{ voyage.id }}">
// ... lines 2 - 4
    <td class="px-6 py-4 whitespace-nowrap flex">
// ... lines 6 - 12
        <twig:Modal :allowSmallWidth="true">
// ... lines 14 - 26
            <div class="flex justify-between">
                <form method="dialog">
                    <twig:Button variant="link">Cancel</twig:Button>
                </form>
// ... line 31
            </div>
        </twig:Modal>
    </td>
</tr>
```

That doesn't exist yet, so in the `Button` class, add it: `variant` and it just needs `text-white`:

src/Twig/Components/Button.php

```
↕ // ... lines 1 - 7
8  class Button
9  {
↕ // ... lines 10 - 12
13     public function getVariantClasses(): string
14     {
15         return match ($this->variant) {
↕ // ... lines 16 - 18
19             'link' => 'text-white',
↕ // ... line 20
21         };
22     }
23 }
```

After the form, to render the delete button, include `voyage/_delete_form.html.twig`:

templates/voyage/_row.html.twig

```
1  <tr class="even:bg-gray-700 odd:bg-gray-600" id="voyage-list-item-{{
   voyage.id }}">
↕ // ... lines 2 - 4
5      <td class="px-6 py-4 whitespace-nowrap flex">
↕ // ... lines 6 - 12
13         <twig:Modal :allowSmallWidth="true">
↕ // ... lines 14 - 26
27             <div class="flex justify-between">
28                 <form method="dialog">
29                     <twig:Button variant="link">Cancel</twig:Button>
30                 </form>
31                 {{ include('voyage/_delete_form.html.twig') }}
32             </div>
33         </twig:Modal>
34     </td>
35 </tr>
```

Oh, and that template has a built-in `confirm`. Delete that because we have something *way* nicer now.

Moment of truth! Refresh and delete. It looks great! Cancel closes the modal... and deleting *works*. And it shouldn't be a surprise that it works. The delete form is *not* inside a `<turbo-frame>`. So when we click delete, that triggers a normal form submit that redirects and causes a normal full page navigation.

Hiding Search Options in a Modal

Ok, I know this is already a full day, but I *really* want to use the modal in one more spot. And it's a cool use-case.

On the homepage, in my PHP & Symfony code, I won't show it, but I already added logic to filter this list by the *planets*. I only didn't add any planet checkboxes to the page because... we don't really have *space* for them.

So here's my idea: add a link here that opens a modal that *holds* the extra filtering options.

Open up `main/homepage.html.twig` and find that input. Start by adding a `<div class="w-1/3 flex">`... add the closing on the other side of the input... then remove `w-1/3` *from* the input. We're making space for that link:

templates/main/homepage.html.twig

```
// ... lines 1 - 27
{% block body %}
    <div class="flex">
// ... lines 30 - 36
        <section class="flex-1 ml-10">
            <form
                method="GET"
                action="{{ path('app_homepage') }}"
                class="mb-6 flex justify-between"
                data-controller="autosubmit"
                data-turbo-frame="voyage-list"
            >
                <div class="w-1/3 flex">
                    <input
                        type="search"
                        name="query"
                        value="{{ app.request.query.get('query') }}"
                        aria-label="Search voyages"
                        placeholder="Search voyages"
                        class="px-4 py-2 rounded bg-gray-800 text-white placeholder-gray-400"
                        data-action="autosubmit#debouncedSubmit"
                        autofocus
                    >
                </div>
// ... lines 57 - 59
            </form>
// ... lines 61 - 145
        </section>
    </div>
{% endblock %}
```

But I'll paste in a full modal:

templates/main/homepage.html.twig

```
// ... lines 1 - 27
{% block body %}
    <div class="flex">
// ... lines 30 - 36
        <section class="flex-1 ml-10">
            <form
                method="GET"
                action="{{ path('app_homepage') }}"
                class="mb-6 flex justify-between"
                data-controller="autosubmit"
                data-turbo-frame="voyage-list"
            >
                <div class="w-1/3 flex">
                    <input
                        type="search"
                        name="query"
                        value="{{ app.request.query.get('query') }}"
                        aria-label="Search voyages"
                        placeholder="Search voyages"
                        class="px-4 py-2 rounded bg-gray-800 text-white placeholder-gray-400"
                        data-action="autosubmit#debouncedSubmit"
                        autofocus
                    >

                    <twig:Modal>
                        <twig:block name="trigger">
                            <twig:Button
                                variant="link"
                                type="button"
                                data-action="modal#open"
                            >Options</twig:Button>
                        </twig:block>

                        <h3 class="text-white text-lg font-semibold mb-2">Search Options</h3>
                        <hr class="mb-4">

                        <div class="flex justify-end">
                            <twig:Button
                                variant="success"
                                data-action="modal#close"
                            >See Results</twig:Button>
                        </div>
                    </twig:Modal>
                </div>
```

```
↕ // ... lines 77 - 79
80                </form>
↕ // ... lines 81 - 165
166           </section>
167       </div>
168   {% endblock %}
```

This will be invisible except for the trigger. So we basically just added a button that says "options". But it's already set *up* to open the modal. Inside, to start, we have an `h3` and a `<twig:Button>` that closes the modal.

Adding a Modal Close Button

But the result when I click options... is nice! Though, it needs a close button on the upper right. We *could* add it to just this modal... but it might be nice if it were an *option* in the modal component.

Let's do it! In `Modal.html.twig`, add one more prop called `closeButton` defaulting to `false`:

templates/components/Modal.html.twig

```
1  {% props allowSmallWidth=false, closeButton=false %}
↕  // ... lines 2 - 37
```

If that's true, at the end of the `dialog`, I'll paste in a close button:

templates/components/Modal.html.twig

```
{% props allowSmallWidth=false, closeButton=false %}
<div
// ... lines 3 - 6
>
// ... lines 8 - 9
    <dialog
// ... lines 11 - 13
    >
// ... lines 15 - 19
        {% if closeButton %}
            <button
                class="absolute right-4 top-3 text-white flex items-center opacity-70 transition-opacity hover:opacity-100"
                data-action="modal#close"
                type="button"
            >
                <svg xmlns="http://www.w3.org/2000/svg" width="24" height="24" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M18 6l-12 12" /><path d="M6 6l12 12" /></svg>
            </button>
        {% endif %}
    </dialog>
// ... lines 30 - 35
</div>
```

Again, nothing special here: some absolute styling, an icon... and the important part: it calls `modal#close`.

In `homepage.html.twig` find that modal and add `closeButton="true"`... but with the `:` like last time:

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 27
28 {% block body %}
29     <div class="flex">
↕ // ... lines 30 - 36
37         <section class="flex-1 ml-10">
38             <form
↕ // ... lines 39 - 43
44             >
45                 <div class="w-1/3 flex">
↕ // ... lines 46 - 56
57                     <twig:Modal :closeButton="true">
↕ // ... lines 58 - 74
75                     </twig:Modal>
76                 </div>
↕ // ... lines 77 - 79
80             </form>
↕ // ... lines 81 - 165
166         </section>
167     </div>
168 {% endblock %}
```

Let's check it out! I *love* that!

Finally, let's frost this cake. Near the bottom of the content, I'll paste in the planet checkboxes:

templates/main/homepage.html.twig

```
// ... lines 1 - 27
{% block body %}
    <div class="flex">
// ... lines 30 - 36
        <section class="flex-1 ml-10">
            <form
// ... lines 39 - 43
            >
                <div class="w-1/3 flex">
// ... lines 46 - 56
                    <twig:Modal :closeButton="true">
// ... lines 58 - 65
                        <h3 class="text-white text-lg font-semibold mb-2">Search Options</h3>
                        <hr class="mb-4">
                        <h4 class="text-white text-sm font-semibold mb-2">
                            Planets
                        </h4>
                        {% for planet in planets %}
                            <div class="flex items-center mb-4">
                                <input
                                    type="checkbox"
                                    class="w-4 h-4 text-blue-600 bg-gray-100 border-gray-300 rounded focus:ring-blue-500 dark:focus:ring-blue-600 dark:ring-offset-gray-800 focus:ring-2 dark:bg-gray-700 dark:border-gray-600"
                                    name="planets[]"
                                    value="{{ planet.id }}"
                                    id="planet-search-{{ planet.id }}"
                                    {{ planet.id in searchPlanets ? 'checked' : '' }}
                                >
                                <label for="planet-search-{{ planet.id }}" class="ms-2 text-sm font-medium text-gray-900 dark:text-gray-300">{{ planet.name }}</label>
                            </div>
                        {% endfor %}
// ... lines 84 - 89
                    </twig:Modal>
                </div>
// ... lines 92 - 94
            </form>
// ... lines 96 - 180
        </section>
    </div>
{% endblock %}
```

This is more boring code! I loop over the planets and render input check boxes. My Symfony controller is already set up to read the `planets` parameter and filter the query.

Final test. Open it up. Lovely! Now watch: click a few. When I press "See Results", the table should update. Boom. It did!

But the coolest part is... *how* this worked! Think about it: I click this button... and the table reloads. That means the form is submitting. But... what caused that? Look at the button: there's no code to submit the form. So what's going on?

Remember: this `button`, the planet checkboxes and this modal physically live *inside* the `<form>` element. And what happens when you press a button that lives inside a form? It submits the form! We run the `modal#close`, but we also allow the browser to do the default behavior: submitting the form. This is ancient alien technology at work!

On the close button, I was a bit sneaky. When I added that, I included a `type="button"`. That tells the browser to *not* submit any form that it might be inside. That's why when we click "X", nothing updates. But when we click "see results", the form submits.

Woh! Best day ever! Tomorrow, it's time to look at Live components, where we take Twig components and let them re-render on the page via Ajax as the user interacts when them.

Chapter 27: Live Components

Happy Day 27 of Last Stack! We've accomplished a lot during the first 26 days with just *three* letters of LAST Stack: Asset Mapper, Stimulus, and Turbo. Today we crack the code on the L of LAST Stack: Live components. Live components let us take a Twig component... then re-render it via Ajax as the user interacts with it.

Our goal is this global search. When I click nothing happens! What I *want* to do is open a modal with a search box that, as we type, loads a live search.

Opening the Search Modal

Start inside `templates/base.html.twig`. Search for search! Perfect: this is the fake search input we just saw. Add a `<twig:Modal>` with `:closeButton="true"`, then a `<twig:block>` with `name="trigger"`. Put the fake input inside that. To make this open the modal, we need `data-action="modal#open"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal :closeButton="true">
                        <twig:block name="trigger">
                            <div
                                class="hidden md:flex pr-10 items-center space-x-2 border-2 border-gray-900 rounded-lg p-2 bg-gray-800 text-white cursor-pointer"
                                data-action="modal#open"
                            >
                                <svg xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 text-gray-500" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M10 10m-7 0a7 7 0 1 0 14 0a7 7 0 1 0 -14 0"/><path d="M21 21l-6 -6"/></svg>
                                <span class="pl-2 pr-10 text-gray-500">Search Cmd+K</span>
                            </div>
                        </twig:block>
                    </twig:Modal>
                </nav>
            </header>
// ... lines 45 - 54
        </div>
// ... lines 56 - 84
    </body>
</html>
```

Cool! If we refresh, nothing changes: the only visible part of the modal is the trigger. For the modal content, after the Twig block, I'll paste in a div:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal :closeButton="true">
                        <twig:block name="trigger">
                            <div
                                class="hidden md:flex pr-10 items-center space-x-2 border-2 border-gray-900 rounded-lg p-2 bg-gray-800 text-white cursor-pointer"
                                data-action="modal#open"
                            >
                                <svg xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 text-gray-500" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M10 10m-7 0a7 7 0 1 0 14 0a7 7 0 1 0 -14 0"/><path d="M21 21l-6 -6"/></svg>
                                <span class="pl-2 pr-10 text-gray-500">Search Cmd+K</span>
                            </div>
                        </twig:block>

                        <div class="relative">
                            <div class="absolute inset-y-0 left-0 pl-3 flex items-center pointer-events-none">
                                <svg xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 text-gray-500" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M10 10m-7 0a7 7 0 1 0 14 0a7 7 0 1 0 -14 0"/><path d="M21 21l-6 -6"/></svg>
                            </div>
                            <input
                                type="search"
                                aria-label="Search site"
                                placeholder="Search for anything"
                                class="px-4 py-2 pl-10 rounded bg-gray-800 text-white placeholder-gray-400 w-full outline-none"
                            />
                        </div>
                    </twig:Modal>
                </nav>
            </header>
// ... lines 57 - 66
```

```
67          </div>
↕ // ... lines 68 - 96
97      </body>
98  </html>
```

Nothing special here: just a *real* search input.

Back at the browser, when I click... uh oh. Nothing happens! Debugging always starts in the console. No errors, but when I click, watch: there's no log that says that the action is being triggered. We've got something wrong with that and maybe you saw my mistake? We added the `data-action` to a `div`. Unlike a `button` or a `form`, Stimulus doesn't have a default event for a `div`. Add `click->`:

templates/base.html.twig

```
1   <!DOCTYPE html>
2   <html>
↕ // ... lines 3 - 15
16      <body class="bg-black text-white font-mono">
17          <div class="container mx-auto min-h-screen flex flex-col">
18              <header class="my-8 px-4">
19                  <nav class="flex items-center justify-between mb-4">
↕ // ... lines 20 - 31
32                      <twig:Modal :closeButton="true">
33                          <twig:block name="trigger">
34                              <div
↕ // ... line 35
36                                  data-action="click->modal#open"
37                              >
↕ // ... lines 38 - 39
40                              </div>
41                          </twig:block>
↕ // ... lines 42 - 53
54                      </twig:Modal>
55                  </nav>
56              </header>
↕ // ... lines 57 - 66
67          </div>
↕ // ... lines 68 - 96
97      </body>
98  </html>
```

And now... got it!

Oh, and it auto-focused the input! That's.... just a feature of dialogs! They work like a mini page within a page: it autofocuses the first tabbable element... or you can use the normal

`autofocus` attribute for more control. It just works how you want it to.

Modal: Control the Padding

Anyway, I'm picky: this is more padding than I want. But that's ok! We can make our Modal component just a *bit* more flexible. In `components/Modal.html.twig`, the extra padding is this `p-5`. On top, add a third `prop`: `padding='p-5'`. Copy that. And down here, render `padding`:

templates/components/Modal.html.twig

```
{% props allowSmallWidth=false, closeButton=false, padding="p-5" %}
<div
// ... lines 3 - 6
>
// ... lines 8 - 9
    <dialog
// ... lines 11 - 13
    >
        <div class="flex grow {{ padding }}">
// ... lines 16 - 18
        </div>
// ... lines 20 - 28
    </dialog>
// ... lines 30 - 35
</div>
```

Over in `base.html.twig`, on the modal, add `padding` equals empty quotes:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal :closeButton="true" padding="">
// ... lines 33 - 53
                    </twig:Modal>
                </nav>
            </header>
// ... lines 57 - 66
        </div>
// ... lines 68 - 96
    </body>
</html>
```

Let's check it! And... much neater.

Creating the Twig Component

To bring the results to life, we *could* repeat the data-tables setup from the homepage. We could add a `<turbo-frame>` with the results right here and make the input autosubmit *into* that frame.

Another option is to build this with a live component. But before we talk about that, let's *first* organize the modal contents into a *twig* component.

In `templates/components/`, create a new file called `SearchSite.html.twig`. I'll add a div with `{{ attributes }}`. Then go steal the entire body of the modal, and paste it here:

templates/components/SearchSite.html.twig

```
<div {{ attributes }}>
    <div class="relative">
        <div class="absolute inset-y-0 left-0 pl-3 flex items-center pointer-events-none">
            <svg xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 text-gray-500" stroke-width="2" stroke="currentColor" fill="none" stroke-linecap="round" stroke-linejoin="round"><path stroke="none" d="M0 0h24v24H0z" fill="none"/><path d="M10 10m-7 0a7 7 0 1 0 14 0a7 7 0 1 0 -14 0"/><path d="M21 21l-6 -6"/></svg>
        </div>
        <input
            type="search"
            aria-label="Search site"
            placeholder="Search for anything"
            class="px-4 py-2 pl-10 rounded bg-gray-800 text-white placeholder-gray-400 w-full outline-none"
        />
    </div>
</div>
```

Over in `base.html.twig`, it's easy, right? `<twig:SearchSite />` and done:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal :closeButton="true" padding="">
                        <twig:block name="trigger">
// ... lines 34 - 40
                        </twig:block>

                        <twig:SearchSite />
                    </twig:Modal>
                </nav>
            </header>
// ... lines 47 - 56
        </div>
// ... lines 58 - 86
    </body>
</html>
```

At the browser, we get the same result.

Fetching Data with a Twig Component

The site search is really going to be a *voyage* search. To render the results, we have two options. First, we could... *somehow* get the voyages that we want to show inside of `base.html.twig` and pass them into `SearchSite` as a prop. But... fetching data from our base layout is tricky... we'd probably need a custom Twig function.

The second option is to leverage our Twig component! One of its superpowers is the ability to fetch its own data: to be standalone.

To do that, this Twig component now needs a PHP class. In `src/Twig/Components/`, create a new PHP class called `SearchSite`. The only thing that this needs to be recognized as a Twig component is an attribute: `#[AsTwigComponent]`:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 2
3 namespace App\Twig\Components;
↕ // ... lines 4 - 6
7 use Symfony\UX\TwigComponent\Attribute\AsTwigComponent;
8
9 #[AsTwigComponent]
10 class SearchSite
11 {
↕ // ... lines 12 - 22
23 }
```

This is exactly what we saw inside the `Button` class. A few days ago, I quickly mentioned that Twig component classes are *services*, which means we can autowire *other* services like `VoyageRepository`, `$voyageRepository`:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 5
6 use App\Repository\VoyageRepository;
↕ // ... lines 7 - 8
9 #[AsTwigComponent]
10 class SearchSite
11 {
12     public function __construct(private VoyageRepository
  $voyageRepository)
13     {
14     }
↕ // ... lines 15 - 22
23 }
```

To provide the data to the template, create a new method called `voyages()`! This will return an array... which will really be an array of `Voyage[]`. Inside `return $this->voyageRepository->findBySearch()`. That's the same method we're using on the homepage. Pass `null`, an empty array, and limit to 10 results:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 4
5 use App\Entity\Voyage;
↕ // ... lines 6 - 8
9 #[AsTwigComponent]
10 class SearchSite
11 {
↕ // ... lines 12 - 15
16     /**
17      * @return Voyage[]
18      */
19     public function voyages(): array
20     {
21         return $this->voyageRepository->findBySearch(null, [], 10);
22     }
23 }
```

The search query isn't dynamic yet, but we *do* now have a `voyages()` method that we can use in the template. I'll start with a bit of styling, then it's normal twig code: `{% for voyage in this` - that's our component object - `.voyages`. Add `endfor`, and in the middle, I'll paste that in:

templates/components/SearchSite.html.twig

```
<div {{ attributes }}>
// ... lines 2 - 13
    <div class="text-white py-2 rounded-lg">
        {% for voyage in this.voyages %}
            <a href="{{ path('app_voyage_show', { id: voyage.id }) }}" class="flex items-center space-x-4 px-4 p-2 hover:bg-gray-700 cursor-pointer">
                <img
                    class="h-10 w-10 rounded-full"
                    src="{{ asset('images/'~voyage.planet.imageFilename) }}"
                    alt="{{ voyage.planet.name }} planet"
                >

                <div>
                    <p class="text-sm font-medium text-white">{{ voyage.purpose }}</p>
                    <p class="text-xs text-gray-400">{{ voyage.leaveAt|ago }}</p>
                </div>
            </a>
        {% endfor %}
    </div>
</div>
```

Nothing special: an anchor tag, an image tag, and some info.

Let's try it. Open! Sweet! Though, of course, when we type, nothing updates! Lame!

Installing & Upgrading to a LiveComponent

This is where live components comes in handy. So let's get it installed!

```
composer require symfony/ux-live-component
```

To upgrade our Twig component to a Live component, we only need to do two things. First, it's `#[AsLiveComponent]`. And second, use `DefaultActionTrait`:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 6
7  use Symfony\UX\LiveComponent\Attribute\AsLiveComponent;
8  use Symfony\UX\LiveComponent\DefaultActionTrait;
9
10 #[AsLiveComponent]
11 class SearchSite
12 {
13     use DefaultActionTrait;
↕ // ... lines 14 - 25
26 }
```

That's an internal detail... but needed.

So far, nothing will change. It's still a Twig component... and we haven't added any *live* component superpowers.

Adding a Writable Prop

One of the key concepts with a Live Component is that you can add a property and allow the user to *change* that property from the frontend. For example, create a `public string $query` to represent the search string:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 10
11 #[AsLiveComponent]
12 class SearchSite
13 {
↕ // ... lines 14 - 16
17     public string $query = '';
↕ // ... lines 18 - 29
30 }
```

Below, use that when we call the repository:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 10
11 #[AsLiveComponent]
12 class SearchSite
13 {
↕ // ... lines 14 - 16
17     public string $query = '';
↕ // ... lines 18 - 25
26     public function voyages(): array
27     {
28         return $this->voyageRepository->findBySearch($this->query, [],
   10);
29     }
30 }
```

To allow the user to modify this property, we need to give it an attribute: `#[LiveProp]` with `writeable: true`:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 7
8  use Symfony\UX\LiveComponent\Attribute\LiveProp;
↕ // ... lines 9 - 10
11 #[AsLiveComponent]
12 class SearchSite
13 {
↕ // ... lines 14 - 15
16     #[LiveProp(writable: true)]
17     public string $query = '';
↕ // ... lines 18 - 29
30 }
```

Finally, to *bind* this property to the input - so that the `query` property changes as the user types - add `data-model="query"`:

templates/components/SearchSite.html.twig

```
1  <div {{ attributes }}>
2      <div class="relative">
↕ // ... lines 3 - 5
6          <input
7              type="search"
8              data-model="query"
↕ // ... lines 9 - 11
12         />
13     </div>
↕ // ... lines 14 - 30
31 </div>
```

That's it! Check out the result. We start with everything, but when we type... it filters! It even has built-in debouncing.

Backstage, it makes an AJAX request, populates the `query` property with this string, re-renders the Twig template and pops it right here.

Now that this is working, I don't think we need to load all the results at first. And, look, it's just PHP, so this is easy. If not `$this->query`, then return an empty array:

src/Twig/Components/SearchSite.php

```
↕ // ... lines 1 - 10
11 #[AsLiveComponent]
12 class SearchSite
13 {
↕ // ... lines 14 - 25
26     public function voyages(): array
27     {
28         if (!$this->query) {
29             return [];
30         }
↕ // ... lines 31 - 32
33     }
34 }
```

And in `SearchSite.html.twig`, add an if statement around this: `if this.voyages` is not empty, render that... with the `endif` at the bottom:

templates/components/SearchSite.html.twig

```
1  <div {{ attributes }}>
↕ // ... lines 2 - 14
15     {% if this.voyages is not empty %}
16     <div class="text-white py-2 rounded-lg">
17         {% for voyage in this.voyages %}
↕ // ... lines 18 - 29
30         {% endfor %}
31     </div>
32     {% endif %}
33 </div>
```

For those of you that are sticklers for details, yes, with `this.voyages`, we're calling the method *twice*. But there *are* ways around this - and my favorite is called `#[ExposeInTemplate]`. I won't show it, but it's a quick change.

Fixing the Modal to the Top

So, I'm happy! But, this isn't *perfect*... and I want that. One thing that bothers me is the position: it looks low when it's empty. And as we type, it jumps around. That's the native `<dialog>` positioning, which is normally *great*, but not when our content is changing. So in this one case, let's fix the position near the top.

In `Modal.html.twig`, add one last piece of flexibility to our component: a prop called `fixedTop = false`:

templates/components/Modal.html.twig

```
{% props
// ... lines 2 - 3
    padding="p-5",
    fixedTop=false
%}
// ... lines 7 - 42
```

Then, at the end of the `dialog` classes, if `fixedTop`, render `mt-14` to set the top margin. Else do nothing:

templates/components/Modal.html.twig

```
// ... lines 1 - 6
<div
// ... lines 8 - 11
>
// ... lines 13 - 14
    <dialog
        class="open:flex bg-gray-800 rounded-lg shadow-xl inset-0 w-full md:w-fit md:max-w-[50%] {{ allowSmallWidth ? '' : 'md:min-w-[50%] ' }}animate-fade-in backdrop:bg-slate-600 backdrop:opacity-80{{ fixedTop ? ' mt-14' : '' }}"
// ... lines 17 - 18
    >
// ... lines 20 - 33
    </dialog>
// ... lines 35 - 40
</div>
```

Over in `base.html.twig`, on the modal... it's time to break this onto multiple lines. Then pass `:fixedTop="true"`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
↕ // ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
↕ // ... lines 20 - 31
                    <twig:Modal :closeButton="true" padding="" :fixedTop="true">
↕ // ... lines 33 - 43
                    </twig:Modal>
                </nav>
            </header>
↕ // ... lines 47 - 56
        </div>
↕ // ... lines 58 - 86
    </body>
</html>
```

And now, ah. Much nicer and no more jumping around.

Setting the Search as Turbo Permanent

What else? Pressing up and down on my keyboard to go through the results *is* needed, though I'll save that for another time. But watch this. If I search, then click out and navigate to another page, not surprisingly, when we open the search modal, it's empty. It would be *really* cool if it *remembered* the search.

And we can do that with a trick from Turbo. In `base.html.twig`, on the modal, add `data-turbo-permanent`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal :closeButton="true" padding="" :fixedTop="true" data-turbo-permanent id="global-search-modal">
// ... lines 33 - 43
                    </twig:Modal>
                </nav>
            </header>
// ... lines 47 - 56
        </div>
// ... lines 58 - 86
    </body>
</html>
```

That tells Turbo to *keep* this on the page when it navigates. When you use this, it needs an id.

Let's see how this feels. Open the search, type something, click off, go to the homepage and open it again. So darn cool!

Opening Search on Ctrl+K

Ok, *final* thing! Up here, I'm advertising that you open the search with a keyboard shortcut. That's a lie! But we *can* add this... and, again, it's easy.

On the modal, add a `data-action`. Stimulus has built-in support for doing things on `keydown`. So we can say `keydown.`, then whatever key we want, like `K`. Or in this case, `Ctrl`+`K`.

If we stopped now, this would only trigger if the modal were focused and then someone pressed `Ctrl`+`K`. That's... not going to happen. Instead, we want this to open no matter *what* is focused. We want a *global* listener. Do that by adding `@window`.

Copy that, add a space, paste and also trigger on `meta+k`. Meta is the command key on a Mac:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body class="bg-black text-white font-mono">
        <div class="container mx-auto min-h-screen flex flex-col">
            <header class="my-8 px-4">
                <nav class="flex items-center justify-between mb-4">
// ... lines 20 - 31
                    <twig:Modal
// ... lines 33 - 37
                        data-action="keydown.meta+k@window->modal#open keydown.ctrl+k@window->modal#open"
                    >
// ... lines 40 - 50
                    </twig:Modal>
                </nav>
            </header>
// ... lines 54 - 63
        </div>
// ... lines 65 - 93
    </body>
</html>
```

Testing time! I'll move over and... keyboard! I love it! Done!

Lazy-Loading Live Component

Oh, and Live Components can also be loaded lazily via AJAX! Watch: add a `defer` attribute. When we refresh, we won't *see* any difference... because that component is hidden on page load anyway. But in reality, it just loaded *empty* then immediately made an Ajax call to load for real. We can see that down here in the web debug toolbar! This is a great way to defer loading something heavy, so it doesn't slow down your page.

It's not particularly useful in *our* case because the SearchSite component is so lightweight, so I'll remove it.

Tomorrow, we'll spend one more day with Live Components - this time to give a form real-time-validation superpowers *and* solve the age-old pesky problem of dynamic or dependent form fields.

Chapter 28: Real-Time Validation & Dependent Form Fields

For day 28, I want to show you one of the most common ways that people are using Live Components: forms. Because Live Components have this power to reload as you type, they give us interesting possibilities with forms, like real-time validation! So here's today's goal: convert the Voyage form into a Live Component and *see* some cool real-time validation for ourselves!

We already have a controller that takes care of creating the Voyage form and handles this submit. What *we're* going to do is wrap the frontend part of the form inside a Live Component so that as we type, it re-renders. But ultimately, when we save, it'll save like normal through the controller.

Moving the Form into a Twig Component

For step one, forget about Live Components: let's just convert the form rendering into a Twig Component. In this case, I know we're going to need a PHP class, so create a new one called `VoyageForm` and make it a Twig Component with `#[AsTwigComponent]`:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 2
3 namespace App\Twig\Components;
↕ // ... lines 4 - 5
6 use Symfony\UX\TwigComponent\Attribute\AsTwigComponent;
7
8 #[AsTwigComponent]
9 class VoyageForm
10 {
↕ // ... line 11
12 }
```

Perfect! The form itself lives in `templates/voyage/_form.html.twig` and uses a `form` variable, which we'll need to pass *into* the Twig component.

In the `VoyageForm` class, add a public property for this: `public FormView $form`, because `FormView` is the object type for the `form` variable:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 4
5 use Symfony\Component\Form\FormView;
↕ // ... lines 6 - 7
8 #[AsTwigComponent]
9 class VoyageForm
10 {
11     public FormView $form;
12 }
```

Next, in `templates/components/`, create the component template: `VoyageForm.html.twig`. Copy the entire form, paste it here:

templates/components/VoyageForm.html.twig

```
1  {{ form_start(form) }}
2      {{ form_widget(form) }}
3
4      <twig:Button
5          formnovalidate
6          variant="success"
7          class="hover:animate-wiggle"
8      >
9          {{ button_label|default('Save') }}
10     </twig:Button>
11 {{ form_end(form) }}
```

And then in `_form.html.twig`, it's simple: `<twig:VoyageForm />`:

templates/voyage/_form.html.twig

```
1 <twig:VoyageForm :form="form" />
```

And over at the browser... bah! We get:

> *"Variable `form` does not exist."*

Let's think about this. We *do* have a public property in the component class called `form`... so we *should* have a local variable with that name. *But*, the property is uninitialized because I forgot to pass in that value. My bad! Pass `:form="form"` - using `:` so that the value - `form` - is Twig code: that's the `form` variable:

templates/voyage/_form.html.twig

```twig
<twig:VoyageForm :form="form" />
```

And now... got it! Before we keep going, inside the template, remember to render the `attributes` variable. The easiest is to wrap this in a `div` and say `{{ attributes }}`. I'll put the closing tag... then indent the entire form:

templates/components/VoyageForm.html.twig

```twig
<div {{ attributes }}>
    {{ form_start(form) }}
    // ... lines 3 - 11
    {{ form_end(form) }}
</div>
```

So the form rendering is now a Twig component. But to give it *behavior*, we need a Live Component.

LiveComponent & Symfony Forms

Let's think. After changing any field, I want a Live Component to collect the value of every field and send them to the Live Component system via an Ajax call. The Live Component will then *submit* these values into the form object and rerender the template.

Using Symfony forms with Live Components is a bit more of a complex use-case than the *normal* case of Live components: where we create some public properties and make them writable.

Fortunately, Live Component ships with a trait to help. In `VoyageForm`, first, convert this to a Live Component by saying `#[AsLiveComponent]` then using the `DefaultActionTrait`:

src/Twig/Components/VoyageForm.php

```php
// ... lines 1 - 9
use Symfony\UX\LiveComponent\Attribute\AsLiveComponent;
// ... line 11
use Symfony\UX\LiveComponent\DefaultActionTrait;

#[AsLiveComponent]
class VoyageForm extends AbstractController
{
    use DefaultActionTrait;
    // ... lines 18 - 27
}
```

Next, because we want to bind this component to a form object, use `ComponentWithFormTrait`. When we do that, we don't need this public `form` property anymore because that lives inside the trait:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 10
11 use Symfony\UX\LiveComponent\ComponentWithFormTrait;
↕ // ... lines 12 - 13
14 #[AsLiveComponent]
15 class VoyageForm extends AbstractController
16 {
17     use DefaultActionTrait;
18     use ComponentWithFormTrait;
19
↕ // ... lines 20 - 27
28 }
```

However, this trait *does* require one new method. Go to "Code"->"Generate" - or `Cmd`+`N` on a Mac - and implement the one we need: `instantiateForm()`:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 7
8 use Symfony\Component\Form\FormInterface;
↕ // ... lines 9 - 14
15 class VoyageForm extends AbstractController
16 {
↕ // ... lines 17 - 19
20     protected function instantiateForm(): FormInterface
21     {
↕ // ... lines 22 - 26
27     }
28 }
```

This might look strange at first. But remember, as we change fields in our form, the form values will be sent via Ajax back to our Live component... which then needs to *submit* them into the form object so it can re-render. This means that, during the Ajax call, our Live Component needs to be able to create our form object. To do that, it calls this method.

To get the logic for this, in `VoyageController`, all the way at the bottom, copy the guts of `createVoyageForm()`... then paste them here. Hit okay to add the two `use` statements:

src/Twig/Components/VoyageForm.php

```
// ... lines 1 - 4
use App\Entity\Voyage;
use App\Form\VoyageType;
// ... lines 7 - 14
class VoyageForm extends AbstractController
{
// ... lines 17 - 19
    protected function instantiateForm(): FormInterface
    {
        $voyage = $voyage ?? new Voyage();

        return $this->createForm(VoyageType::class, $voyage, [
            'action' => $voyage->getId() ? $this->generateUrl('app_voyage_edit', ['id' => $voyage->getId()]) : $this->generateUrl('app_voyage_new'),
        ]);
    }
}
```

There's... just one problem: the `createForm()` and `generateUrl()` methods don't exist here! But I haven't told you about a crazy, cool thing: Live Components are Symfony controllers in disguise! And this means we can extend `AbstractController`:

src/Twig/Components/VoyageForm.php

```
// ... lines 1 - 6
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
// ... lines 8 - 14
class VoyageForm extends AbstractController
{
// ... lines 17 - 27
}
```

That's totally allowed and gives us access to all the shortcuts we know and love.

Ok, showtime! Move over. When I type, nothing happens. In this case, Live Components waits for the field to *change*... so it waits for us to move *off* of the field. As soon as we do, we'll see an Ajax request fire down here. Watch. Boom! See it? That sent the data back, submitted the form and *re-rendered* the form.

To prove this, clear out the field and hit tab. A validation error! That's coming from Symfony and the normal form validation rendering! Type something again, tab, it goes away. The best part? The planet field down here is *also* required thanks to Symfony's validation constraints. But the Live Component system is smart: it knows that the user hasn't *changed* this field yet, so it

shouldn't show the validation error. But if we *do* select a planet... then clear, when it re-renders, it shows the error.

Passing the Initial Form Data

This also works fine for the edit form. Hit edit & clear out a field.

Though, check out `instantiateForm()`. Hmm, we're always instantiating a *new* `Voyage` object: there's never a `$voyage` variable. We change a field, Live Components sends an Ajax request and, when it creates the form, it does it using a brand *new* `Voyage` object, not the *existing* `Voyage` object from the database.

And... that's probably okay... because it submits all the data onto it, and it renders correctly.

However, one thing you can do with Live components is submit the form directly *into* the Component object and handle the save logic there. We're not going to do that, but if we *did*, the `Voyage` object bound to the form would always be a *new* object... and it would always insert a new row into the database.

Passing in the Initial Form Data

So even though this works, it's a bit weird.

To tighten this up, we can store the existing `Voyage` object on the component and use *that* during form creation. Add a public `?Voyage` `$initialFormData` property. Above this, to make the component system *remember* this value through all of its Ajax requests, add `#[LiveProp]`:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 10
11 use Symfony\UX\LiveComponent\Attribute\LiveProp;
↕ // ... lines 12 - 14
15 #[AsLiveComponent]
16 class VoyageForm extends AbstractController
17 {
↕ // ... lines 18 - 20
21     #[LiveProp]
22     public ?Voyage $initialFormData = null;
↕ // ... lines 23 - 31
32 }
```

This is now a non-writable prop that our component will keep track of. And yes, it's non-writable: the user changes the *form* data directly, not this property. This is *just* here to help us create the form object on each Ajax call.

Below, change this to `$voyage` equals `$this->initialFormData`, else `new Voyage()`:

src/Twig/Components/VoyageForm.php

```
↕ // ... lines 1 - 14
15 #[AsLiveComponent]
16 class VoyageForm extends AbstractController
17 {
↕ // ... lines 18 - 20
21     #[LiveProp]
22     public ?Voyage $initialFormData = null;
23
24     protected function instantiateForm(): FormInterface
25     {
26         $voyage = $this->initialFormData ?? new Voyage();
↕ // ... lines 27 - 30
31     }
32 }
```

Finally, pass in the `initialFormData` by saying `:initialFormData="voyage"`, which is a Twig variable that we already have:

templates/voyage/_form.html.twig

```
1 <twig:VoyageForm :form="form" :initialFormData="voyage" />
```

So we won't notice a difference, but when we hit edit and change a field, that Ajax request now creates a Form object bound to this existing `Voyage` object.

That got a bit technical, but let's zoom out. By rendering out form through a Live Component, we get real-time validation for free! That's *cool*.

Dependent Form Fields

We're almost out of time, but I think we can tackle one more form problem today. In fact, maybe *the* most *painful* form problem in all of Symfony.

On this form, if the planet is *not* in our solar system, I want to render a new dropdown for an optional wormhole upgrade. This is the classic dependent form field problem. In Symfony, it's hard because we need to leverage form events. On the frontend it's hard too! Historically, we needed to write JavaScript to trigger an Ajax call to re-render the form.

But... that second part is now taken care of! Live Components is great at re-rendering the form when fields change. And the first part? Yea, there's a new library that makes *that* easy too!

It's called `symfonycasts/dynamic-forms`... created by us because this problem drove me absolutely crazy. Hat tip to Symfony dev Ben Davies who really cracked the code on this.

Copy the composer require line, spin over, and run that:

```
composer require symfonycasts/dynamic-forms
```

Using this is really pleasant. Find the form class: `src/Form/VoyageType.php`. The library uses decoration. At the top, say `$builder` equals `new DynamicFormBuilder()` and pass in `$builder`:

src/Form/VoyageType.php

```
// ... lines 1 - 12
use Symfonycasts\DynamicForms\DynamicFormBuilder;

class VoyageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder = new DynamicFormBuilder($builder);
// ... lines 20 - 52
```

This `DynamicFormBuilder` has the same methods as the original, but one extra: `addDependent()`. But before we use it, comment-out the `'autocomplete' => true`:

src/Form/VoyageType.php

```
↕ // ... lines 1 - 12
13 use Symfonycasts\DynamicForms\DynamicFormBuilder;
14
15 class VoyageType extends AbstractType
16 {
17     public function buildForm(FormBuilderInterface $builder, array
   $options): void
18     {
19         $builder = new DynamicFormBuilder($builder);
20         $builder
↕ // ... lines 21 - 24
25             ->add('planet', null, [
↕ // ... lines 26 - 27
28                 //'autocomplete' => true,
29             ])
↕ // ... lines 30 - 41
42         ;
43     }
↕ // ... lines 44 - 50
51 }
```

There's a bug with the autocomplete system and Live Components. It should be fixed soon, but I don't want it to get in the way.

Anyway, the `addDependent()` method takes three arguments. The first is the name of the new field: `wormholeUpgrade`. The second is an array of fields that this field *depends* on. In this case, that's only `planet`. The final argument is a callback function and *its* first argument will always be a `DependentField` object. We'll see how that's used in a minute. Then, this will receive the value of every field that it depends on. Because we depend only on `planet`, the callback will receive *that* as an argument: `?Planet` `$planet`:

src/Form/VoyageType.php

```
// ... lines 1 - 12
use Symfonycasts\DynamicForms\DynamicFormBuilder;

class VoyageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder = new DynamicFormBuilder($builder);
        $builder
// ... lines 21 - 24
            ->add('planet', null, [
// ... lines 26 - 27
                //'autocomplete' => true,
            ])
            ->addDependent('wormholeUpgrade', ['planet'], function (DependentField $field, ?Planet $planet) {
// ... lines 31 - 40
            })
        ;
    }
// ... lines 44 - 50
}
```

Inside, if we *don't* have a planet - because the user hasn't selected one yet *or* the planet is in the Milky Way, just return. And yes, I borked up my space science: I meant for this to be `isInOurSolarSystem()` - not the milky way. Forgive me Data!

Anyway, because we're returning, there won't be a `wormholeUpgrade` field at all. Else, add one with `$field->add()`. This method is identical to the normal `add()` method except that we don't need to pass the *name* of the field... because we already pass it earlier. So skip straight to `ChoiceType::class`... then the options with `choices` set to an array of "Yes" for true, and "No" for false:

src/Form/VoyageType.php

```
↕ // ... lines 1 - 7
8 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
↕ // ... lines 9 - 14
15 class VoyageType extends AbstractType
16 {
17     public function buildForm(FormBuilderInterface $builder, array $options): void
18     {
↕ // ... line 19
20         $builder
↕ // ... lines 21 - 29
30             ->addDependent('wormholeUpgrade', ['planet'], function (DependentField $field, ?Planet $planet) {
31                 if (!$planet || $planet->isInMilkyWay()) {
32                     return;
33                 }
34 
35                 $field->add(ChoiceType::class, [
36                     'choices' => [
37                         'Yes' => true,
38                         'No' => false,
39                     ],
40                 ]);
41             })
42         ;
43     }
↕ // ... lines 44 - 50
51 }
```

Done! Go check out the result. Refresh, edit and change to a planet that's not in our system. There it is! The field popped into existence! If we go back to a planet that *is* in our solar system... gone! And... the field saves just fine. When we edit the voyage, the form starts with it. It just works!

Ok, we're nearly at the end of our 30-day journey! Tomorrow, it's time to talk about how we can *test* our beautiful new frontend features.

Chapter 29: Testing Part 1: Twig & Live Components

All these nifty gadgets that we've built are just toys, unless we can test them. So, that's today's mission! Tons to tackle, so let's jump right in!

Run:

```
composer require phpunit
```

That installs the `symfony/test-pack`, gives us all the packages we need and puts them into `require-dev`.

Testing a Twig Component

For our first act, let's test a Twig Component. This is pretty cool: we can create the component object, call methods on it and assert how it's rendered, all in isolation. It's simple, but we'll test the `Button` component.

In the `tests/` directory, create an `Integration/` directory - because this will be an integration test - then `Twig/Components/`. If you're new to integration tests, check our Integration Testing tutorial.

Inside, create a new `ButtonTest` class... and extend the normal `KernelTestCase` for integration tests:

tests/Integration/Twig/Components/ButtonTest.php

```
↕ // ... lines 1 - 2
3 namespace App\Tests\Integration\Twig\Components;
↕ // ... lines 4 - 5
6 use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
↕ // ... lines 7 - 8
9 class ButtonTest extends KernelTestCase
10 {
↕ // ... lines 11 - 21
22 }
```

To help us work with the component, use a trait called `InteractsWithTwigComponents`, then add a new function: `testButtonRendersWithVariants()`:

tests/Integration/Twig/Components/ButtonTest.php

```
↕ // ... lines 1 - 6
7 use Symfony\UX\TwigComponent\Test\InteractsWithTwigComponents;
8
9 class ButtonTest extends KernelTestCase
10 {
11     use InteractsWithTwigComponents;
12
13     public function testButtonRendersWithVariants()
14     {
↕ // ... lines 15 - 20
21     }
22 }
```

Mounting the Component

The trait gives us two methods. The first lets us *create* the component object. Say `$this->mountTwigComponent()` passing the component name `Button` and any props, like `variant` set to `success`.

This should give us a `Button`: `assertInstanceOf`, `Button::class`, `$component`. Dump `$component` then `assertSame` that `success` is equal to `$component->variant`:

tests/Integration/Twig/Components/ButtonTest.php

```
// ... lines 1 - 8
class ButtonTest extends KernelTestCase
{
// ... lines 11 - 12
    public function testButtonRendersWithVariants()
    {
        $component = $this->mountTwigComponent('Button', [
            'variant' => 'success',
        ]);
        dump($component);
        $this->assertInstanceOf(Button::class, $component);
        $this->assertSame('success', $component->variant);
    }
}
```

Cool! To try this, run:

```
./vendor/bin/simple-phpunit tests/Integration
```

That'll download PHPUnit, and... it passes! We have some deprecation notices, but ignore those.

Rendering the Component

The second thing we can do is *render* a component. Copy the top, paste on the bottom, rename this to `$rendered` and call `renderTwigComponent()`. This has almost the same arguments, but we can also pass blocks. The third argument is a shortcut to pass the `content` block.

Dump `$rendered`:

tests/Integration/Twig/Components/ButtonTest.php

```
// ... lines 1 - 8
class ButtonTest extends KernelTestCase
{
// ... lines 11 - 12
    public function testButtonRendersWithVariants()
    {
        $component = $this->mountTwigComponent('Button', [
            'variant' => 'success',
        ]);
        $this->assertInstanceOf(Button::class, $component);
        $this->assertSame('success', $component->variant);

        $rendered = $this->renderTwigComponent('Button', [
            'variant' => 'success',
        ], '<span>Click me!</span>');
        dump($rendered);
    }
}
```

And let's see what this looks like!

```
./vendor/bin/simple-phpunit tests/Integration
```

Awesome! An object with the HTML inside. With this, we can get the raw string... *or* we can access a `Crawler` object. This is cool: `$this->assertSame()` that `Click Me!`, is equal to `$rendered->crawler()->filter()` - to find the `span` - then `->text()`:

tests/Integration/Twig/Components/ButtonTest.php

```
// ... lines 1 - 8
class ButtonTest extends KernelTestCase
{
// ... lines 11 - 12
    public function testButtonRendersWithVariants()
    {
        $component = $this->mountTwigComponent('Button', [
            'variant' => 'success',
        ]);
        $this->assertInstanceOf(Button::class, $component);
        $this->assertSame('success', $component->variant);

        $rendered = $this->renderTwigComponent('Button', [
            'variant' => 'success',
        ], '<span>Click me!</span>');
        $this->assertSame('Click me!', $rendered->crawler()->filter('span')->text());
    }
}
```

Super sweet! My editor's yelling 'syntax error', but it's being dramatic. Watch:

```
./vendor/bin/simple-phpunit tests/Integration
```

It passes!

Testing a Live Component

So how about integration testing a live component... like our fancy `SearchSite`? In the same directory, create a new class called `SearchSiteTest`, extend `KernelTestCase` and... this time use `InteractsWithLiveComponents`. Create a method: `testCanRenderAndReload()`:

tests/Integration/Twig/Components/SearchSiteTest.php

```
// ... lines 1 - 2
namespace App\Tests\Integration\Twig\Components;

use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
use Symfony\UX\LiveComponent\Test\InteractsWithLiveComponents;

class SearchSiteTest extends KernelTestCase
{
    use InteractsWithLiveComponents;

    public function testCanRenderAndReload()
    {
// ... lines 14 - 15
    }
}
```

With this trait, we can say `$testComponent` equals `$this->createLiveComponent()`. Pass the name - `SearchSite`... and we can also pass any props, but I won't. We'll let the `$query` start empty. `dd($testComponent)`:

tests/Integration/Twig/Components/SearchSiteTest.php

```
// ... lines 1 - 5
use Symfony\UX\LiveComponent\Test\InteractsWithLiveComponents;

class SearchSiteTest extends KernelTestCase
{
    use InteractsWithLiveComponents;

    public function testCanRenderAndReload()
    {
        $testComponent = $this->createLiveComponent('SearchSite');
        dd($testComponent);
    }
}
```

When we run this:

```
./vendor/bin/simple-phpunit tests/Integration
```

The object is *humongous*... but it's a `TestLiveComponent`. And it has a *ton* of goodies. We can say `$testComponent->component()` to get the underlying component object, we can

render it, and we can even mimic user behavior, like changing a model value, calling live actions, emitting events or even logging in.

Test Database Setup

To test the search, we need to add some voyages to the database. On top, `use ResetDatabase` and `use Factories`:

tests/Integration/Twig/Components/SearchSiteTest.php

```
// ... lines 1 - 7
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;

class SearchSiteTest extends KernelTestCase
{
    use InteractsWithLiveComponents;
    use ResetDatabase;
    use Factories;
// ... lines 16 - 26
}
```

Down here, use `VoyageFactory::createMany()` to create 5 voyages... and give them all the same `purpose` so we can easily search for them. Then create one more `Voyage` with any other random `purpose`:

tests/Integration/Twig/Components/SearchSiteTest.php

```
// ... lines 1 - 10
class SearchSiteTest extends KernelTestCase
{
// ... lines 13 - 16
    public function testCanRenderAndReload()
    {
        VoyageFactory::createMany(5, [
            'purpose' => 'first 5 voyages',
        ]);
        VoyageFactory::createOne();

        $testComponent = $this->createLiveComponent('SearchSite');
        dd($testComponent);
    }
}
```

Before we take advantage of these, try the test again:

```
./vendor/bin/simple-phpunit tests/Integration
```

A database connection error! I'm running the database via Docker & using the `symfony` binary to set the `DATABASE_URL` environment variable. To inject that variable when running the test, prefix the command with `symfony php`:

```
symfony php vendor/bin/simple-phpunit tests/Integration
```

And... we're back! One risky test because we don't have any assertions. Let's add those!

Remember: if there is no `query`, our component returns no voyages. And in the template: `templates/components/SearchSite.html.twig`, when we *do* have results, each is an `a` tag.

In the test, `$this->assertCount()` that 0 is equal to `$testComponent->render()`, then use that same `->crawler()` to filter for `a` tags.

Here's the *really* cool part: call `$testComponent->set()` `query` to `first 5` to mimic the user typing into the search box. And now we should have 5 results:

tests/Integration/Twig/Components/SearchSiteTest.php

```
↕ // ... lines 1 - 10
11 class SearchSiteTest extends KernelTestCase
12 {
↕ // ... lines 13 - 16
17     public function testCanRenderAndReload()
18     {
↕ // ... lines 19 - 23
24         $testComponent = $this->createLiveComponent('SearchSite');
25
26         $this->assertCount(0, $testComponent->render()->crawler()->filter('a'));
27         $testComponent->set('query', 'first 5');
28         $this->assertCount(5, $testComponent->render()->crawler()->filter('a'));
29     }
30 }
```

Do it!

```
symfony php vendor/bin/simple-phpunit tests/Integration
```

Green! Ok, today is a bit unorthodox because... we're out of time... but I have more to say! Next up is part *two* where we take on functional tests for our JavaScript-powered frontend.

Chapter 30: Testing Part 2: Functional Testing

Welcome back to part 2 of day 29. I bent the rules today and made it a double feature. We talked about testing Twig & Live components... but we *also* need to talk about functional - or end-to-end - testing in general. That's where we programmatically control a browser, have it click links, fill out forms, etc.

Two things about this. First, we're going to create a system that I *really* like. And second, the road to *get* there is going to be... honestly, a bit bumpy. It's *not* a smooth process and that's something we as a community should work on.

zenstruck/browser

Symfony has built-in functional testing tools, but I like to use another library. At your terminal, install it with:

```
composer require zenstruck/browser --dev
```

Next, in the `tests/` folder, I'll create a new directory called `Functional/`... then a new class called `VoyageControllerTest`. And I guess I *could* put that into a `Controller/` directory also.

For the guts, I'll paste in a finished test:

tests/Functional/VoyageControllerTest.php

```php
// ... lines 1 - 2
namespace App\Tests\Functional;

use App\Factory\PlanetFactory;
use App\Factory\VoyageFactory;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use Zenstruck\Browser\Test\HasBrowser;
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;

class VoyageControllerTest extends WebTestCase
{
    use ResetDatabase;
    use Factories;
    use HasBrowser;

    public function testCreateVoyage()
    {
        PlanetFactory::createOne([
            'name' => 'Earth',
        ]);
        VoyageFactory::createOne();

        $this->browser()
            ->visit('/')
            ->click('Voyages')
            ->click('New Voyage')
            ->fillField('Purpose', 'Test voyage')
            ->selectFieldOption('Planet', 'Earth')
            ->click('Save')
            ->assertElementCount('table tbody tr', 2)
            ->assertSee('Bon voyage')
        ;
    }
}
```

Ok, we're using `ResetDatabase` and `Factories`... it extends the normal `WebTestCase` for functional tests... and then `HasBrowser` comes from the Browser library and gives us the ability to call `$this->browser()` to control a browser with this really smooth API. This goes through the flow of going to the voyage page, clicking "New voyage", filling out the form, saving and asserting at the bottom. The test starts with a single `Voyage` in the database, so after we create a new one, we assert that there are *two* on the page.

To run this, use the same command, but target the `Functional/` directory:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

And... it actually passes! Sweet!

Testing JavaScript with Panther

But hold your horses. Behind the scenes, this is *not* using a real browser: it's just making fake requests in PHP. That means it doesn't execute JavaScript. We're testing the experience a user would have if they had JavaScript *disabled*. That's fine for many situations. However, *this* time, I want to test all the modal fanciness.

To run the test using a *real* browser that supports JavaScript - like Chrome - change to `$this->pantherBrowser()`:

tests/Functional/VoyageControllerTest.php

```
// ... lines 1 - 11
class VoyageControllerTest extends WebTestCase
{
// ... lines 14 - 17
    public function testCreateVoyage()
    {
// ... lines 20 - 24
        $this->pantherBrowser()
// ... lines 26 - 33
        ;
    }
}
```

Try it:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

No dice! But a nice error: we need to install `symfony/panther`. Let's do that!

```
composer require symfony/panther --dev
```

Panther is a PHP library that can programmatically control *real* browsers on your machine. To use it, we *also* need to extend `PantherTestCase`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 6
7 use Symfony\Component\Panther\PantherTestCase;
↕ // ... lines 8 - 11
12 class VoyageControllerTest extends PantherTestCase
13 {
↕ // ... lines 14 - 35
36 }
```

Try it again:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

We don't *see* the browser - it opens invisibly in the background - but it's now using Chrome! And the test fails - pretty early:

> *"Clickable element "New Voyage" not found."*

Hmm. It clicked "Voyages", but didn't find the "New Voyage" button. A fantastic feature of `zenstruck/browser` with Panther is that, when a test fails, it takes a *screenshot* of the failure.

Inside the `var/` directory... here it is. Huh, the screenshot shows that we're still on the homepage - as if we never clicked "Voyages"... though you can kind of see that the voyages link looks active.

The problem is that the page navigation happens via Ajax... and our tests don't know to *wait* for that to finish. It clicks "Voyages"... then immediately tries to click "New Voyage". This will be the *main* thing that we need to fix.

Loading a "test" Dev Server

But before that, I see a bigger problem! Look at the data: this is *not* coming from our test database! This is coming from our dev site!

Even though we can't see it, Panther *is* controlling a *real* browser. And... a real browser needs to access our site using a real web server via a real web address. Because we're using the Symfony web server, Panther detected that and... used it!

But... that's *not* what we want! Why? Our server is using the `dev` environment and the `dev` database. Our tests should use the `test` environment and the `test` database.

To fix this, open up `phpunit.xml.dist`. I'll paste in two environment variables:

phpunit.xml.dist

```
↕ // ... lines 1 - 3
4 <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
↕ // ... lines 5 - 9
10 >
11     <php>
↕ // ... lines 12 - 17
18         <server name="SYMFONY_PROJECT_DEFAULT_ROUTE_URL" value="" />
19         <server name="PANTHER_APP_ENV" value="test" />
20     </php>
↕ // ... lines 21 - 40
41 </phpunit>
```

The first... is kind of a hack. That tells Panther to *not* use our server. Instead, Panther will now silently start its *own* web server using the built-in PHP web server. The second line tells Panther to use the `test` environment when it does that.

Over in the test, to make it even easier to see if this is working, after we click voyages, call `ddScreenshot()`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 11
12 class VoyageControllerTest extends PantherTestCase
13 {
↕ // ... lines 14 - 17
18     public function testCreateVoyage()
19     {
↕ // ... lines 20 - 24
25         $this->pantherBrowser()
26             ->visit('/')
27             ->click('Voyages')
28             ->ddScreenshot()
↕ // ... lines 29 - 34
35         ;
36     }
37 }
```

Take a screenshot, then dump and die.

Run it:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

It hits that... and saved a screenshot! Cool! Find that in `var/`. And... ok. It looks like the new web server is being used... but it's missing all the styles!

Debugging by Opening the Browser

Time for some detective work! To understand what's going on, we can temporarily tell Panther to *actually* open the browser, like, so we can see it and play with it.

After we visit, say `->pause()`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 11
12 class VoyageControllerTest extends PantherTestCase
13 {
↕ // ... lines 14 - 17
18     public function testCreateVoyage()
19     {
↕ // ... lines 20 - 24
25         $this->pantherBrowser()
26             ->visit('/')
27             ->pause()
↕ // ... lines 28 - 35
36         ;
37     }
38 }
```

Then, to open the browser, prefix the test command with `PANTHER_NO_HEADLESS=1`:

```
PANTHER_NO_HEADLESS=1 symfony php vendor/bin/simple-phpunit tests/Functional
```

And... woh! It popped up the browser then paused. *Now* we can view the page source. Here's the CSS file. Open that. It's a 404 not found. Why?

In the dev environment, our assets are served through *Symfony*: they're not real, physical files. If you prefix the URL with `index.php`, it works. Panther uses the built-in PHP web server... and it needs a rewrite rule that tells it to send these URLs through Symfony. Honestly, it's an annoying detail, but we can fix it.

Back at the terminal, hit enter to close the browser. In `tests/`, create a new file called `router.php`. I'll paste in the code:

tests/router.php

```
↕ // ... lines 1 - 2
3 if (is_file($_SERVER['DOCUMENT_ROOT'].\DIRECTORY_SEPARATOR.$_SERVER['SCRIPT_NAM
  {
4     return false;
5 }
6
7 $script = 'index.php';
8
9 $_SERVER = array_merge($_SERVER, $_ENV);
10 $_SERVER['SCRIPT_FILENAME'] =
  $_SERVER['DOCUMENT_ROOT'].\DIRECTORY_SEPARATOR.$script;
11
12 $_SERVER['SCRIPT_NAME'] = \DIRECTORY_SEPARATOR.$script;
13 $_SERVER['PHP_SELF'] = \DIRECTORY_SEPARATOR.$script;
14
15 require $script;
```

This is a "router" file that will be used by the built-in web server. To tell Panther to use it, in `phpunit.xml.dist`, I'll paste in another env var: `PANTHER_WEB_SERVER_ROUTER` set to `../tests/router.php`:

phpunit.xml.dist

```
↕ // ... lines 1 - 3
4 <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
↕ // ... lines 5 - 9
10 >
11     <php>
↕ // ... lines 12 - 19
20         <server name="PANTHER_WEB_SERVER_ROUTER"
  value="../tests/router.php" />
21     </php>
↕ // ... lines 22 - 41
42 </phpunit>
```

Try it!

```
PANTHER_NO_HEADLESS=1 symfony php vendor/bin/simple-phpunit tests/Functional
```

And now... it works! Hit enter to finish. Then remove the `pause()`.

Run the test again, but without the env var:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

Waiting for the Turbo Page Load

Cool: it hit our screenshot line. Pop that open. Ok, we're back to the original problem: it's not waiting for the page to load after we click the link.

Solving this... isn't as simple as it should be. Say `$browser =`, close that and start a new chain with `$browser` below. In between, I'll paste in two lines. This is lower-level, but waits for the `aria-busy` attribute to be added to the `html` element, which Turbo does when it's loading. Then it waits for it to go away:

tests/Functional/VoyageControllerTest.php

```
// ... lines 1 - 11
class VoyageControllerTest extends PantherTestCase
{
// ... lines 14 - 17
    public function testCreateVoyage()
    {
// ... lines 20 - 24
        $browser = $this->pantherBrowser()
            ->visit('/')
            ->click('Voyages')
        ;
        $browser->client()->waitFor('html[aria-busy="true"]');
        $browser->client()->waitFor('html:not([aria-busy])');
        $browser
            ->ddScreenshot()
            ->click('New Voyage')
            ->fillField('Purpose', 'Test voyage')
            ->selectFieldOption('Planet', 'Earth')
            ->click('Save')
            ->assertElementCount('table tbody tr', 2)
            ->assertSee('Bon voyage')
        ;
    }
}
```

Try the test now:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

Then... pop open the screenshot. Woh! It *is* now waiting for the Ajax call to finish. But remember: we're also using view transitions. The page loaded... but it's still in the middle of the transition. We'll fix that in a minute.

Custom Browser & Base Test Class

But first, we need to clean this up: this is *way* too much work. What I would *love* is a new method on the browser itself - like `waitForPageLoad()`. And we can do that with a custom browser class!

In the `tests/` directory, create a new class called `AppBrowser`. I'll paste in the guts:

tests/AppBrowser.php

```php
// ... lines 1 - 2
namespace App\Tests;

use Zenstruck\Browser\PantherBrowser;

class AppBrowser extends PantherBrowser
{
    public function waitForPageLoad(): self
    {
        $this->client()->waitFor('html[aria-busy="true"]');
        $this->client()->waitFor('html:not([aria-busy])');

        return $this;
    }
}
```

This extends the normal `PantherBrowser` and adds a new method which those same two lines.

When we call `$this->pantherBrowser()`, we now want it to return *our* `AppBrowser` instead of the normal `PantherBrowser`. To do that, you guessed it, it's an environment variable: `PANTHER_BROWSER_CLASS` set to `App\Tests\AppBrowser`:

phpunit.xml.dist

```
↕ // ... lines 1 - 3
4 <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
↕ // ... lines 5 - 9
10 >
11     <php>
↕ // ... lines 12 - 20
21         <server name="PANTHER_BROWSER_CLASS" value="App\Tests\AppBrowser"
/>
22     </php>
↕ // ... lines 23 - 42
43 </phpunit>
```

To make sure this is working, `dd(get_class($browser));`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 11
12 class VoyageControllerTest extends PantherTestCase
13 {
↕ // ... lines 14 - 17
18     public function testCreateVoyage()
19     {
↕ // ... lines 20 - 24
25         $browser = $this->pantherBrowser()
26             ->visit('/')
27             ->click('Voyages')
28         ;
29         dd(get_class($browser));
↕ // ... lines 30 - 40
41     }
42 }
```

Run the test:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

And... yes! We get `AppBrowser`! Unfortunately, while the new method *would* work, we don't get autocompletion. Our editor has no idea that we swapped in a sub-class.

To improve this, let's do one last thing: in `tests/`, create a new base test class: `AppPantherTestCase`. I'll paste in the content:

tests/AppPantherTestCase.php

```
↕ // ... lines 1 - 2
3  namespace App\Tests;
4
5  use Symfony\Component\Panther\PantherTestCase;
6  use Zenstruck\Browser\Test\HasBrowser;
7
8  class AppPantherTestCase extends PantherTestCase
9  {
10     use HasBrowser {
11         pantherBrowser as parentPantherBrowser;
12     }
13
14     protected function pantherBrowser(array $options = [], array
   $kernelOptions = [], array $managerOptions = []): AppBrowser
15     {
16         return $this->parentPantherBrowser($options, $kernelOptions,
   $managerOptions);
17     }
18 }
```

It extends the normal `PantherTestCase`... then overrides the `pantherBrowser()` method, calls the parent, but changes the return type to be *our* `AppBrowser`.

Over in `VoyageControllerTest`, change this to `extend` `AppPantherTestCase`, then make sure to remove `use HasBrowser`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 6
7  use App\Tests\AppPantherTestCase;
↕ // ... lines 8 - 10
11 class VoyageControllerTest extends AppPantherTestCase
12 {
13     use ResetDatabase;
14     use Factories;
15
↕ // ... lines 16 - 35
36 }
```

Then we can tighten things up: reconnect all of these spots... then use the new method: `->waitForPageLoad()`... with auto-complete! Remove the `ddScreenshot()`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 10
11 class VoyageControllerTest extends AppPantherTestCase
12 {
↕ // ... lines 13 - 15
16     public function testCreateVoyage()
17     {
↕ // ... lines 18 - 22
23         $this->pantherBrowser()
24             ->visit('/')
25             ->click('Voyages')
26             ->waitForPageLoad()
27             ->click('New Voyage')
28             ->fillField('Purpose', 'Test voyage')
29             ->selectFieldOption('Planet', 'Earth')
30             ->click('Save')
31             ->assertElementCount('table tbody tr', 2)
32             ->assertSee('Bon voyage')
33         ;
34     }
35 }
```

And let's see where we are!

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

Further!

> *“Form field "Purpose" not found.”*

So it clicked Voyages, clicked "New Voyage"... but couldn't find the form field. If we look down at the error screenshot, we can see why: the modal content is still loading! You *might* see the form in your screenshot - sometimes the screenshot happens *just* a moment later, so the form is visible - but this *is* the problem.

Disabling View Transitions

Oh, but before we fix this, I also want to disable view transitions. In `templates/base.html.twig`, the easiest way to make sure view transitions don't muck up our tests is to remove them. Say if `app.environment != 'test`, then render this `meta` tag:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 6
        {% if app.environment != 'test' %}
            <meta name="view-transition">
        {% endif %}
// ... lines 10 - 16
    </head>
// ... lines 18 - 97
</html>
```

Waiting for the Modal to Load

Anyway, back to our failure. When we click to open the modal, what need wait for the modal to open - that's actually instant - but *also* wait for the `<turbo-frame>` inside to finish loading.

Open `AppBrowser`. I'll paste in two more methods:

tests/AppBrowser.php

```
// ... lines 1 - 4
use Facebook\WebDriver\WebDriverBy;
// ... lines 6 - 7
class AppBrowser extends PantherBrowser
{
// ... lines 10 - 17
    public function waitForDialog(): self
    {
        $this->client()->wait()->until(function() {
            return $this->crawler()->filter('dialog[open]')->count() > 0;
        });

        if ($this->crawler()->filter('dialog[open] turbo-frame')->count() > 0) {
            $this->waitForTurboFrameLoad();
        }

        return $this;
    }

    public function waitForTurboFrameLoad(): self
    {
        $this->client()->wait()->until(function() {
            return $this->crawler()->filter('turbo-frame[aria-busy="true"]')->count() === 0;
        });

        return $this;
    }
}
```

The first - `waitForDialog()` - waits until it sees a dialog on the page with an `open` attribute. And, *if* that open `dialog` has a `<turbo-frame>`, it waits for that to load: it waits until there aren't any `aria-busy` frames on the page.

In `VoyageControllerTest`, after clicking "New Voyage", say `->waitForDialog()`:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 10
11 class VoyageControllerTest extends AppPantherTestCase
12 {
↕ // ... lines 13 - 15
16     public function testCreateVoyage()
17     {
↕ // ... lines 18 - 22
23         $this->pantherBrowser()
↕ // ... lines 24 - 26
27             ->click('New Voyage')
28             ->waitForDialog()
29             ->fillField('Purpose', 'Test voyage')
↕ // ... lines 30 - 33
34         ;
35     }
36 }
```

And now:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

So close!

> "`table` `tbody` `tr` *expected 2 elements on the page but only found 1.*"

That comes from all the way down here! What's the problem this time? Back to the error screenshot! Ah: we filled out the form, it looks like we even hit Save... but we're asserting too quickly!

Remember: this submits into to a `<turbo-frame>`, so we need to wait for that frame to finish loading. And we have a way to do this: `->waitForTurboFrameLoad()`. I'll also add a line to assert that we cannot see any open dialogs: to check that the modal closed:

tests/Functional/VoyageControllerTest.php

```
↕ // ... lines 1 - 10
11 class VoyageControllerTest extends AppPantherTestCase
12 {
↕ // ... lines 13 - 15
16     public function testCreateVoyage()
17     {
↕ // ... lines 18 - 22
23         $this->pantherBrowser()
24             ->visit('/')
25             ->click('Voyages')
26             ->waitForPageLoad()
27             ->click('New Voyage')
28             ->waitForDialog()
29             ->fillField('Purpose', 'Test voyage')
30             ->selectFieldOption('Planet', 'Earth')
31             ->click('Save')
32             ->waitForTurboFrameLoad()
33             ->assertElementCount('table tbody tr', 2)
34             ->assertNotSeeElement('dialog[open]')
35             ->assertSee('Bon voyage')
36         ;
37     }
38 }
```

Run the test one more time:

```
symfony php vendor/bin/simple-phpunit tests/Functional
```

It passes. Woo! I admit, that was some work, too much work! But I do love the end result.

Tomorrow - for our final day - we're going to talk about performance. And unlike today, things are going to quickly fall into place - I promise.

Chapter 31: Performance

We've made it to the *last* day of LAST Stack! I've been waiting for 30 days to say that.

Today is all about performance, starting with the things that we are *not* doing.

No File Combining or Minifying

For example, we are *not* combining files to reduce requests. And, we are *not* minifying files. Nope, we're serving up raw source files from our `assets/` directory.

And yet, our frontend is fast! Open your debugging tools and go to Lighthouse. Let's profile this for performance on the desktop to keep things simple. Give this a few seconds to run and... boom! 99! That's amazing!

On Production: Compression & Caching

Scroll down to see what we could improve. The number one problem is missing compression. There are two things that you need to think about when you deploy your app with AssetMapper.

First: on your web server, enable compression, like gzip or Brotli. Or you can proxy your site through Cloudflare and it can do compression for you. That's what we do. This is why we don't need to worry about minification: if you just compress your CSS and JavaScript files, that does almost as good of a job as minification.

The second thing you need to do - which should be mentioned down here, ah yes:

> *“Serve static assets with an efficient cache policy.”*

Because all of our files have an automatic version hash in the filename, you should configure your web server to cache *everything* from your `assets/` directory... *forever*. This means that when your user downloads a file, they'll cache it forever: they'll never need to download it again. That's great for performance.

Unused CSS?

Let's see what else we have. Reduce unused CSS. That's probably *not* a problem. In fact, it's one of the *benefits* of Tailwind: it only builds the CSS that we're *actually* using. My guess is that the rest of the CSS is used on different pages. And the difference is even smaller than it looks. This is 38 kilobytes... before compression. On production, the difference would be much smaller.

Unused JavaScript

Under reduce unused JavaScript, there's one main item: it's the Live Components JavaScript, which *is* fairly big. We *are* using it, but it's true that we're not using a lot of its features yet. On production, due to compression, this would be smaller... and we *are* going to optimize it a bit.

Next is: eliminate render-blocking resources. This *is* important and it lists our CSS file. We'll come back to this in a few minutes.

But really... there's nothing major. We *could* minify CSS, but it would barely make a difference. Minifying JavaScript - 68 kilobytes looks good, but again, that's before it's compressed. And remember our score of 99! Our frontend is zippy!

Oh, though apparently my images are *way* too big. There *are* still some things you need to handle on your own.

Preloading

One of the main reasons that our app is already so fast is preloading. Look at the page source. We have the importmap, a bunch of preloads, then the all-important: `<script type="module">`, `import 'app'`.

When our browser sees this, it connects `app` to the real filename and starts downloading it. Module script tags are not "render blocking". This means that the browser starts downloading this file, but continues to render the page visually while it's doing that. But, of course, it can't execute our JavaScript until it's done downloading `app.js`.

And there's a problem hiding. Only *after* it finishes downloading `app.js` does it realize that... it also needs to download this file, and this file, and this file, and this file, and this file. And it's only

after downloading `bootstrap.js` that it realizes it needs to download *this* file. You can imagine a big waterfall: it finishes one JavaScript file, starts a few more, finishes those, then starts even more. It could take a long time for our JavaScript to finally execute.

This is where these preloads come in. This tells our browser:

> *"You don't realize it yet, but you should start downloading these files immediately."*

The way these are generated is *really* cool. Open `templates/base.html.twig`. All of this is rendered thanks to `importmap('app')`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
↕ // ... lines 4 - 13
        {% block javascripts %}
            {{ importmap('app') }}
        {% endblock %}
    </head>
↕ // ... lines 18 - 97
</html>
```

By passing `app`, the main effect is that it adds the script tag at the bottom that imports `app`.

But this *also* tells AssetMapper to parse `app.js`, find all the files that *it* imports and add them as preloads. And it does it recursively: it goes into `bootstrap.js` and finds *its* import. It finds *all* the JavaScript that's needed on page load and makes sure that every file is preloaded. It just works.

And we can see this visually. In `alien-greeting.js`: comment-out the import for the CSS file: the delay just makes the waterfall harder to see:

assets/lib/alien-greeting.js

```
export default function (message, inPeace = false) {
    if (!inPeace) {
        setTimeout(() => {
            //import('../styles/alien-greeting.css');
        }, 4000);
    }
↕ // ... lines 7 - 10
```

Then go to the Network tab, look just at JavaScript and do a force refresh. Check it out! All the JavaScript files start at the same time! It's not waiting for anything to download: they all start immediately. *That's* what we want to see.

The only file that starts later is `celebrate-controller.js`... because we set this up to be lazy. This means our JavaScript initializes, *then* it downloads this controller only when it's needed... which is *always* because it's on every page, but it's still delayed a bit.

Lazy-Loading Live Components

Sort this by filesize. The biggest file is the JavaScript for Live Components. This 123 kilobytes isn't compressed, so it'll be smaller on production. But since we only need this on the global search, we could choose to delay loading it.

To do that, inside `assets/controllers.json`, find the Live Component controller and set `fetch` to `lazy`:

assets/controllers.json

```
{
    "controllers": {
// ... lines 3 - 12
        "@symfony/ux-live-component": {
            "live": {
// ... line 15
                "fetch": "lazy",
// ... lines 17 - 18
                }
            }
        },
// ... lines 22 - 31
    },
// ... line 33
}
```

Do a force refresh. It's still there, but check out the initiator: it's from a JavaScript file and starts much later. In the source, search for `live_controller`. Previously, it was preloaded. When we refresh now, it's still in the importmap, but no longer preloaded. We preload the really important stuff, and let the live controller load itself later.

Preloading CSS with WebLink

Ok one last thing, magical thing. The most important thing that we saw inside Lighthouse was the render-blocking resource for our CSS file. When your browser sees a `<link rel="stylesheet">` tag, it freezes rendering the page until it finishes downloading the file. And that's a good thing: we don't want our page to render unstyled for a second.

And this is why we put our CSS `link` tags up in the `head` of the page: we want the browser to notice that it needs to download the file as *early* as possible. *However*, there *is* a way to tell our browser even *earlier* that it needs to download this file.

Find your terminal and run:

```
composer require symfony/web-link
```

This is a small package that can help add hints to your browser about what it needs to download. AssetMapper comes with special integration for it.

Watch: *just* by installing that, go to the Network tab, filter all, refresh and go to the top to the main request for the page. Look down here at the Response headers. There it is! Our app just added a new response header called `link` that points to the CSS file with `rel="preload"`.

This tells the browser that it should download this file. And it sees this header even *earlier* than it sees line 11 of the HTML. This helps performance just a *little* bit more.

Now that we've made a few changes, let's run Lighthouse again. There *is* some variability in these runs, so if your score doesn't change or even goes down a little, no worries. But a perfect 100! Woo!

More importantly.... we still have text compression... but we don't see the render-blocking resource warning.

The moral of the story is this: using AssetMapper is fast out of the box. Other than adding compression and caching to your web server, you can code in peace without worrying. And sure, later, it *is* helpful to run Lighthouse and see how you can improve, but it doesn't need to be something you think about day-by-day. Get your real work done instead.

And... we're finished! Thank you for spending these wild 30 days with me! It has been an absolute pleasure and a heck of a ride. Please, go build things and let us know what they are!

And if you have any questions, comments, doubts or bad jokes, we're always here for you down in the comments section.

Alright friends, see ya next time!

Chapter 32: Bonus: More on Flowbite

A bonus topic! Yeah, because I started to get questions - good questions - about Flowbite. On day 5 we added Tailwind and I introduced Flowbite as a site where you can copy and paste visual components. For example, you copy this markup, paste, and boom! You have a dropdown. The classes are all standard Tailwind classes.

And so, I mentioned that you don't need to install anything. However, depending on what you want, that's not the full story... and I confused people. So let's fix that!

Installing The Flowbite JavaScript

Beyond being a source to copy HTML, Flowbite itself has two other features. First, it has an optional JavaScript library for powering things like tabs and dropdowns: a little JavaScript so that when we click, this opens and closes.

We're *not* using this at SymfonyCasts... and it doesn't play well with Turbo. At least not out of the box. We prefer to create tiny Stimulus controllers to power things like this. But, we *can* get the Flowbite JavaScript to work.

Grab that dropdown markup and zip over to `templates/base.html.twig`. Just inside the `body`, paste:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 17
    <body class="bg-black text-white font-mono">
// ... lines 19 - 24
    <!-- Dropdown menu -->
    <div id="dropdown" class="z-10 hidden bg-white divide-y divide-gray-100 rounded-lg shadow w-44 dark:bg-gray-700">
        <ul class="py-2 text-sm text-gray-700 dark:text-gray-200" aria-labelledby="dropdownDefaultButton">
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Dashboard</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Settings</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Earnings</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Sign out</a>
          </li>
        </ul>
    </div>
// ... lines 42 - 120
    </body>
</html>
```

If we go over and refresh, you can see what I mean: it just works. Well, *visually*. But if we click, nothing happens.

To get the Flowbite JavaScript, find your terminal and run:

```
php bin/console importmap:require flowbite
```

This installs `flowbite` and it dependency `@popperjs/core`. It also grabbed the Flowbite CSS file... which is only needed if you didn't have Tailwind properly installed. Having it hanging around in `importmap.php` is harmless, but let's kick it out before it confuses me.

To use the JavaScript, open `assets/app.js`. On top `import 'flowbite'`:

```
assets/app.js
↕ // ... lines 1 - 5
6 import 'flowbite';
↕ // ... lines 7 - 43
```

Ok, refresh and... it works!

But there are two... quirks. Check out the console. We have a bunch of errors about modal and popover. If you use the modal component from Flowbite, it requires a `data-modal-target` attribute to connect the button to the target. The problem is that *we* have a modal *Stimulus* controller.... and we're using `data-modal-target` to leverage a *Stimulus* target. Those two ideas are colliding. You would need to work around this by using Flowbite's modal system or renaming your modal controller to something else. The same is true for Popover.

Fixing Flowbite JS & Turbo

The second quirk is that, though the Flowbite JavaScript works right now, as soon as we navigate, it breaks! Flowbite initializes the event listener on page load, but when we navigate and *new* HTML is loaded onto the page, it's not smart enough to reinitialize that JavaScript. That's why, in general, we write our JavaScript using Stimulus controllers.

Flowbite *does* ship with a version of itself for Turbo... but it doesn't *quite* work: it doesn't reinitialize correctly on form submits.

That's ok! We've got the skills to patch this up ourselves. Import `initFlowbite` from `flowbite`:

```
assets/app.js
↕ // ... lines 1 - 5
6 import { initFlowbite } from 'flowbite';
↕ // ... lines 7 - 50
```

Then at the bottom, I'll paste in two event listeners:

assets/app.js

```
// ... lines 1 - 43
document.addEventListener('turbo:render', () => {
    initFlowbite();
});
document.addEventListener('turbo:frame-render', () => {
    initFlowbite();
});
```

Flowbite handles initializing on the first page load. Then anytime we navigate with Turbo, this method will be called and will reinitialize the listeners. Or if we do something inside a Turbo frame, *this* will be called.

Let's try it. Refresh. And... it doesn't work: Look: `initFlobite`. Typo! Fix that then... ok. On page load, it works. And if we navigate, it *still* works.

The Flowbite Tailwind Plugin

So the first installable feature of Flowbite is this JavaScript library. The second is a Tailwind plugin. It adds extra styles if you use tooltips, forms, and charts.... as well as a few other things. You can find the package on npmjs.com and navigate its files to find the plugin: `plugin.js`.

If you're using tooltips, it adds new styles, same thing for forms... then *all* the way at the bottom, it tweaks some theme styles. This isn't necessarily something that you *need*, even if you're using some of the JavaScript from Flowbite.

But if you *do* want this plugin, you need to install it with npm. So far, we haven't had to do *anything* with npm... and that's been great! But if you *do* need a few JavaScript libraries, that's ok: that's npm's job. The most important thing is that we don't have a giant build system: we're just grabbing a library here or there that we need.

Find your terminal and run `npm init` to create a `package.json` file.

```
npm init
```

I'll hit `Enter` for all the questions. Then run:

```
npm add flowbite
```

To use this, open `tailwind.config.js`... here it is. Down in the `plugins` section, `require('flowbite/plugin')`:

tailwind.config.js

```
// ... lines 1 - 3
module.exports = {
// ... lines 5 - 28
  plugins: [
    require('flowbite/plugin'),
// ... lines 31 - 34
  ],
}
```

This is straight from their docs.

Whe we refresh, it works... but we don't see any difference. Like I said, it's not something that we *necessarily* need. Though if you open a form, huh: our labels are suddenly black! That's because Tailwind now thinks we're in light mode... and I was a bit too lazy to style my site for light mode.

By default, Tailwind reads whether you want light mode or dark mode from your operating system preferences. But Flowbite overrides that and changes it to read a `class` on your `body` element. It has documentation on their site on how you can use this and even make a dark mode, light mode switcher.

But I'm going to change this back to the old setting. Say `darkMode`, `media`:

tailwind.config.js

```
// ... lines 1 - 3
module.exports = {
// ... lines 5 - 10
  darkMode: 'media',
// ... lines 12 - 36
}
```

Check it: refresh and... we're back to normal! So that's the Tailwind plugin.

The Datepicker

In addition to these 2 Flowbite features, I've also seen people wanting to use their cool datepicker plugin. So let's get that working!

This datepicker is part of the main `flowbite` library. But if you want to import it directly from JavaScript... then, down here, you're supposed to install a different package. This confused me to be honest. But copy that, spin over and run:

```
php bin/console importmap:require flowbite-datepicker
```

Back at the top of the docs, it says that you can use the datepicker simply by taking an input and giving it a `datepicker` attribute. And that's true... except once again, it won't work with Turbo. It'll work at first... but stop after the first click.

Instead, we're going to initialize this with a Stimulus controller, and it's going to work great!

In `assets/controllers/`, create a new `datepicker_controller.js`. I'll paste in the contents:

assets/controllers/datepicker_controller.js

```
import { Controller } from '@hotwired/stimulus';
import { Datepicker } from 'flowbite-datepicker';

/* stimulusFetch: 'lazy' */
export default class extends Controller {
    datepicker;

    connect() {
        this.element.type = 'text';
        this.datepicker = new Datepicker(this.element, {
            format: 'yyyy-mm-dd',
            autohide: true,
        });
    }

    disconnect() {
        if (this.datepicker) {
            this.datepicker.destroy();
        }

        this.element.type = 'date';
    }
}
// ... lines 24 - 25
```

We're going to attach this controller to an `input` element. In `connect()`, this initializes the date picker and passes `this.element`. The `format` matches the default format that the Symfony `DateType` uses. And `autohide` makes the date picker close when you choose a date, which I like.

I'm also changing the `type` attribute on the `input` to `text` so that we don't have both the datepicker from Flowbite *and* the native browser date picker. In `disconnect()`, we do some cleanup.

We're going to use this on the voyage form: for "Leave at". Open the form type for this: `VoyageType`. Here's the field. Pass an `attr` option with `data-controller` set to `datepicker`:

src/Form/VoyageType.php

```
↕ // ... lines 1 - 14
15 class VoyageType extends AbstractType
16 {
17     public function buildForm(FormBuilderInterface $builder, array $options): void
18     {
↕ // ... line 19
20         $builder
↕ // ... line 21
22             ->add('leaveAt', DateType::class, [
↕ // ... line 23
24                 'attr' => [
25                     'data-controller' => 'datepicker',
26                 ]
27             ])
↕ // ... lines 28 - 44
45         ;
46     }
↕ // ... lines 47 - 53
54 }
```

Let's try this! Refresh and... that's fantastic!

Fixing the Datepicker in a Modal

Though... there's a catch. Go back and open this form in the modal. It doesn't work! Well, it kind of does. See it? It's hiding behind the modal. The datepicker works by appending HTML at the bottom of the `body`. But because that's not inside the `dialog`, it correctly appears *behind* the modal. It's kind of a shame that it doesn't work better with the beautiful native `dialog` element, but we can fix this.

In `datepicker_controller.js`, add a new option called container. This tells the datepicker which element it should add its custom HTML *into*. Say `document.querySelector()` and look for a `dialog[open]`. So if there's a `dialog` on the page that's open, then use that as the container. Else use the normal `body`:

assets/controllers/datepicker_controller.js

```
↕ // ... lines 1 - 4
5 export default class extends Controller {
↕ // ... lines 6 - 7
8     connect() {
↕ // ... lines 9 - 10
11         this.datepicker = new Datepicker(this.element, {
↕ // ... lines 12 - 13
14             container: document.querySelector('dialog[open]') ? 'dialog[open]' : 'body'
15         });
16     }
↕ // ... lines 17 - 24
25 }
↕ // ... lines 26 - 27
```

Making the Modal Click Outside Smarter

And *that* little detail takes care of our problem! Though... it does expose one other small issue. See how the datepicker extends the dialog vertically? If we click here, we're technically clicking on the `dialog` element directly... which triggers our click outside logic.

To fix that, let's make our `modal` controller just a *bit* smarter. At the bottom, I'll paste in a new private method called `isClickInElement()`:

assets/controllers/modal_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 65
66     #isClickInElement(event, element) {
67         const rect = element.getBoundingClientRect();
68         return (
69             rect.top <= event.clientY &&
70             event.clientY <= rect.top + rect.height &&
71             rect.left <= event.clientX &&
72             event.clientX <= rect.left + rect.width
73         );
74     }
75 }
```

If you pass this a click event, it will look at the physical dimensions of this element and see if the click was inside.

Up here in `clickOutside()`, let's change things. Copy this, then if the `event.target` is *not* the `dialog`, we're definitely not clicking outside. So, return.

And if not, `this.isClickInElement()` - passing `event` and `this.dialogTarget` - so if we did not click inside the `dialogTarget` - then we definitely want to close:

assets/controllers/modal_controller.js

```
↕ // ... lines 1 - 2
3 export default class extends Controller {
↕ // ... lines 4 - 46
47     clickOutside(event) {
48         if (event.target !== this.dialogTarget) {
49             return;
50         }
51
52         if (!this.#isClickInElement(event, this.dialogTarget)) {
53             this.dialogTarget.close();
54         }
55     }
↕ // ... lines 56 - 74
75 }
```

A bit more logic, but a bit smarter. Try it. Open the modal and if we click down here... the calendar closes - which is correct - but the modal stays open. Love that!

So I hope that explains Flowbite a bit more. Personally, I don't want most of this stuff, so I'm going to remove it. Inside `tailwind.config.js`, remove the plugin:

tailwind.config.js

```
↕ // ... lines 1 - 3
4 module.exports = {
↕ // ... lines 5 - 29
30   plugins: [
31     require('flowbite/plugin'),
↕ // ... lines 32 - 35
36   ],
37 }
```

Then delete `package.json` and `package-lock.json`.

I also don't want the JavaScript. In `importmap.php`, remove `flowbite` and `@popperjs/core`:

importmap.php

```
↕ // ... lines 1 - 15
16 return [
↕ // ... lines 17 - 51
52     'flowbite' => [
53         'version' => '2.2.1',
54     ],
55     '@popperjs/core' => [
56         'version' => '2.11.8',
57     ],
↕ // ... lines 58 - 60
61 ];
```

But that datepicker is cool, so let's keep that.

In `app.js`, remove the import from `flowbite` and the two functions at the bottom:

assets/app.js

```
↕ // ... lines 1 - 5
6  import { initFlowbite } from 'flowbite';
↕ // ... lines 7 - 43
44 document.addEventListener('turbo:render', () => {
45     initFlowbite();
46 });
47 document.addEventListener('turbo:frame-render', () => {
48     initFlowbite();
49 });
```

Finally, in `base.html.twig`, get rid of that random dropdown:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 17
    <body class="bg-black text-white font-mono">
// ... lines 19 - 24
    <!-- Dropdown menu -->
    <div id="dropdown" class="z-10 hidden bg-white divide-y divide-gray-100 rounded-lg shadow w-44 dark:bg-gray-700">
        <ul class="py-2 text-sm text-gray-700 dark:text-gray-200" aria-labelledby="dropdownDefaultButton">
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Dashboard</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Settings</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Earnings</a>
          </li>
          <li>
            <a href="#" class="block px-4 py-2 hover:bg-gray-100 dark:hover:bg-gray-600 dark:hover:text-white">Sign out</a>
          </li>
        </ul>
    </div>
// ... lines 42 - 120
    </body>
</html>
```

Now... no more JavaScript errors! But because that datepicker was pretty cool, we still have it.

Ok, bonus chapter done! Now back to work - seeya later!

With <3 from SymfonyCasts