Upgrading & What's in Symfony 7

Chapter 1: Upgrading to Symfony 6.4

Hey everyone! Symfony 7 is out! Woo! Well, of course *I'm* excited - I love all things Symfony, Twig, related. But what does it really *mean* that Symfony 7 is out?

Symfony's Delightfully Predictable Release Schedule

Honestly... not much! Thanks to Symfony's release schedule, a new major version isn't much of a big deal... though we try to pretend it is for marketing.

Every 6 months - in May & November - a new minor version is released, like 6.1 or 6.2. *Those* are the versions that contain new features. So it *totally* makes sense to get excited about Symfony 6.3 or fantastically amazing new features in Symfony 6.4. Then, each ".4" version, like 6.4, is released on the *same day* as the .0 version of the next major: 7.0. Yea, 6.4 and 7.0 were released on the *exact* same day and are, effectively, identical! They're twins!

The only difference is that, in 7.0, all the deprecated code paths are removed. And this is the core of what makes Symfony special. The release schedule and the deprecation policy mean that as users, we can upgrade our apps forever across major versions... without it being a big deal or breaking our apps. And that's exactly what we're going to do in this tutorial... followed by a tour of some of my favorite new features.

Project Set Up

As always, to get the most out of this tutorial, code along with me by downloading the course code from this page. After you unzip the file, you'll find a `start/` directory with the same code that you see here. The `README.md` file tells an inspiring tale of how to get the application up and running. I've already done most of the steps, including running `yarn install` and `yarn watch` in this tab.

The final step is to use the `symfony` binary to run:

```
symfony serve -d
```

to start a development web server. I'll click the link. Say hello to Mixed Vinyl: The app from several of our Symfony 6 tutorials, which is currently on 6.1.2.

Using a Newer PHP Version

Open up `composer.json`. Near the top, our app requires `php` 8.1 or greater. In my apps, down under `config.platform.php`, I also like to set the *specific* PHP version that we're using on production:

composer.json

```
{
// ... lines 2 - 4
    "require": {
        "php": ">=8.1",
// ... lines 7 - 32
    },
    "config": {
// ... lines 35 - 44
        "platform": {
            "php": "8.1.0"
        }
    },
// ... lines 49 - 96
}
```

This guarantees that Composer only gives me dependencies compatible with that version.

Locally, if I run `php -v`, I already have PHP 8.3 installed. I *also* have a second `php` binary installed for version 8.1. And thanks to the `8.1` in `composer.json`, when I started the `symfony` web server, it used *that* older version.

Change this to, just PHP `8.3`. Then run:

```
composer up
```

In `composer.json`, all my dependencies - whether they're Symfony or something else - are written in a way that only allows the last number or the *second* to last number to change. Assuming the package maintainers are doing their job, those updates *won't* contain backwards-compatibility breaks. We should be able to upgrade from 6.1 to 6.4... or 2.0 to 2.4 and our app *should* keep rocking like normal!

So running `composer up` to get these updates, in theory, is totally safe.

Encore & Minor Changes

Over in my `yarn` tab, the update triggered an error: something about a controller does not exist. This is special to Symfony UX & Encore. When you update your PHP dependencies, you may need to reinstall your `node` dependencies. Hit Ctrl+C, then run:

```
yarn install --force
```

Or `npm install --force` if you're using `npm`. Then

```
yarn watch
```

again. It's happy! In the main tab, run:

```
git status
```

Alongside the usual suspects, there's a new controller in `controllers.json`... which came from an update to `ux-turbo`. We won't use it, but it's fine there. In `package.json`, it added a new entry for stimulus bundle. This is a relatively new bundle that got installed during the upgrade, and we'll talk more about it soon.

Upgrading to 6.4

So we are *now* using PHP 8.3 and we've upgraded our dependencies a bit. But we're still using Symfony 6.1. To upgrade to 7, we first need to upgrade to 6.4. That'll give us a chance to prep for 7.0 by finding - and fixing - all the deprecations.

And... upgrading is easy! Find `6.1.*`, replace with `6.4.*` and replace all.

composer.json

```
{
// ... lines 2 - 4
    "require": {
// ... lines 6 - 17
        "symfony/asset": "6.4.*",
        "symfony/console": "6.4.*",
        "symfony/dotenv": "6.4.*",
// ... line 21
        "symfony/framework-bundle": "6.4.*",
        "symfony/http-client": "6.4.*",
// ... line 24
        "symfony/proxy-manager-bridge": "6.4.*",
        "symfony/runtime": "6.4.*",
        "symfony/twig-bundle": "6.4.*",
// ... lines 28 - 29
        "symfony/yaml": "6.4.*",
// ... lines 31 - 32
    },
// ... lines 34 - 81
    "extra": {
        "symfony": {
// ... line 84
            "require": "6.4.*",
// ... line 86
        }
    },
    "require-dev": {
// ... line 90
        "symfony/debug-bundle": "6.4.*",
// ... line 92
        "symfony/stopwatch": "6.4.*",
        "symfony/web-profiler-bundle": "6.4.*",
// ... line 95
    }
}
```

Though, be careful. Most of the time, the Symfony version constraints look like this. However, they *could* look like `^6.1`. So don't miss those: the goal is to upgrade every `symfony` package that comes from the *main* repository. That... can be confusing because, mixed in with the packages that we *do* want to upgrade are other packages that live under `symfony`, but are independent and follow their own release timeline & versioning. Ignore those for now - but we *will* make sure *every* package is upgraded by the end.

Also, near the bottom, under `extra.symfony.require`, make sure this is also updated to `6.4.*`. That's a composer optimization that tells it to only worry about 6.4 Symfony versions.

Back over at the terminal, let's do this!

```
composer up
```

Look at those beautiful upgrades from 6.1 to 6.4! *And*... when we try the site, the stinkin' thing still works!

Oh, but check out the PHP version: 8.1.27. When we started the `symfony` web server, it read the PHP 8.1 version from `composer.json`, found that version installed on my machine and used it. We changed this to 8.3, but we need to restart the server to use it. Run:

```
symfony server:stop
```

Then:

```
symfony serve -d
```

Yup: it found the PHP 8.3.1 version on my system. And on the site... got it!

Ok, this is working on Symfony 6.4. Our job *now* is to find every deprecation and fix them. On the web debug toolbar, we apparently hit 22 deprecated code paths on this page! To start fixing these, we'll... cheat... take a shortcut, by upgrading our Flex recipes.

Chapter 2: Flex Recipe Updates

When we install packages, many of them have Flex recipes. These add new files and sometimes modify existing files. They do everything needed so the package works immediately. I love that!

And, over time, these recipes tend to change. Maybe they decide to add a new line to a config file or change a default value.

Fortunately, Flex has a fancy recipe *update* system. And while you don't *need* to update your recipes, it's a great way to keep your app looking and feeling modern. The updates will also help fix some of the deprecation warnings we saw at the end of the previous chapter.

Before you start, make sure you've committed any changes to `git` - I already have - because the recipe update system *works* via Git.

To see the recipes, run:

```
composer recipes
```

Cool! It looks like we have about 8 updates. So let's get to work:

```
composer recipes:update
```

Updating recipes? Yea, it's one of my *favorite* things to do: it gives us a chance to peek into what's been changing in these packages... while we've been busy, you know, doing our real job. I'll hit enter to go down the list one-by-one.

doctrine/doctrine-bundle Recipe Update

First up is Doctrine Bundle: and it's a complex update. It even caused a conflict!

Sometimes we might see that a recipe update changes something - like updating a line in a config file - but we don't really understand *why*. To help, the command lists every pull request *behind* these changes. For example, this lazy ghosts thing... we can click the link to see the PR and the explanation behind it.

Back in my editor, woh! I guess the conflict was in `doctrine.yaml`! Specifically, `server_version` changed. The original recipe gave us config to work with Postgres 13. It now ships with code for Postgres 16.

You don't need to keep the new changes. If your production database is using Postgres 13, keep it! But I'll update to 16.

At the terminal, run:

```
git status
```

Add that file to `git` to resolve it. Then see *all* the changes with:

```
git diff --cached
```

Most of these are version changes: MySQL from 5.7 to 8 and Postgres from 13 to 16. The `doctrine.yaml` config *does* have a few new lines. These are flags where we're opting *into* some low-level change in the system. And there's a good chance that *not* having this config would trigger a deprecation. I'll let you dig deeper into these if you care, but they probably won't affect anything.

`docker-compose.yaml` contains more changes that go from Postgres 13 to 16. So again, you can keep these or get rid of them.

And then, lurking at the bottom, `symfony.lock` keeps track of which version of the recipe we have installed. So, we're good! Commit these changes... and use a better commit message than I am.

To use the new version of Postgres from `docker-compose.yaml`, run:

```
docker compose down
```

Then

```
docker compose up -d
```

We now have Postgres 16 running. Watch: the homepage still works because it doesn't talk to the database. But when we click "browse mixes", broken! An undefined table because we're using a fresh database. Fix that by running:

```
symfony console doctrine:migrations:migrate
```

Cool. And:

```
symfony console doctrine:fixtures:load
```

Double cool. Now... we're good!

doctrine/doctrine-migrations-bundle Recipe Update

Back to the terminal... and back to work:

```
composer recipes:update
```

On deck is `doctrine-migrations-bundle`. This is minor. The bundle comes with a profiler integration: it's this little icon on the web debug toolbar. It's not super useful... and so it's changed to be *not* enabled by default. Let's commit that... and update the next one.

```
composer recipes:update
```

symfony/framework-bundle Recipe Update

Framework bundle! The *core* of Symfony! Run `git diff --cached` to see the changes. Like Doctrine, most of these are low level where we opt into a new behavior. For example, annotations are deprecated, so we're turning them off. `handle_all_throwables` means that Symfony will transform exceptions into error pages but *also* other types of errors. And `storage_factory_id` was removed because that's the default value.

Easy! Commit that... then keep going:

```
composer recipes:update
```

symfony/monolog-bundle Recipe Update

Next up is monolog-bundle. The only change is a new `formatter` key at the end of `monolog.yaml`. This is a consistency change. Down here in the `prod` config, the main log handler already has this `formatter` key. It was added under `deprecations` so that everything is formatted the same. Minor, but nice! We'll talk more about this deprecation log soon.

So, commit! And...

```
composer recipes:update
```

symfony/routing Recipe Update

Routing. Dead simple. The code that imports the `#[Route]` attributes, apparently, needs a `namespace` key. Whatever.

symfony/translation Recipe update

Commit... and onto

```
composer recipes:update
```

symfony/translation. Another easy one: `translation.yaml` used to have some commented-out providers as an example... and now they're gone. But if you install one of these provider packages, *its* recipe will re-add the line.

Commit that... and we're down to the final 2 recipes! These are both related to changes with Webpack Encore and a new StimulusBundle. That deserves its own chapter, so let's do it next!

Chapter 3: Encore, StimulusBundle & their Recipe Changes

Let's keep upgrading recipes.

symfony/twig-bundle Recipe Update

Next up is TwigBundle. This has a conflict in the one file it updated: `templates/base.html.twig`.

And... it's odd. You can see our custom content here.... then the default title with the default favicon down below. Keep our custom stuff, and delete this comment. We don't need that.

Run:

```
git add templates
```

Then:

```
git diff --cached
```

This shows `symfony.lock` of course, but there *was* a change to `base.html.twig`: it removed `encore_entry_link_tags()` and `encore_entry_script_tags()`. Why?

The Rearranging of Recipes

One big recent addition to the Symfony frontend world was StimulusBundle. On its own, that's no big deal. *But*, when it was introduced, various recipes were rearranged. A few changes that used to live in the recipe for one package packed up and moved to another.

For example, these lines used to be part of TwigBundle's recipe, but they moved to the recipe for WebpackEncoreBundle. So when we update the TwigBundle recipe, it *looks* like these lines should be *removed*.

Of course, we *do* still need these, but accept this change temporarily. We'll see these get added back later when we upgrade the WebpackEncoreBundle recipe.

symfony/webpack-encore-bundle Recipe Update

Ok, commit this and... let's do our last recipe update: WebpackEncoreBundle!

And... more conflicts. We can't catch a break. Run:

```
git status
```

Ok, in `package.json`, we have a number of changes. The recipe is trying to upgrade us from Encore version 3 to 4. The biggest difference between 3 and 4 is that it's now your responsibility to have a few packages in *your* `package.json`, like `webpack` itself... or the babel packages.

Let's keep version 4... and keep everything else. This is a mixture of custom packages that we've added and the new ones needed for Encore 4.

Run:

```
git add package.json
```

Then check out what else changed with `git diff`. Some meaningless config, `package.json` and `symfony.lock`. `webpack.config.js` holds some low-level changes: using a newer version of core.js and the `plugin-proposal-class-properties` isn't needed anymore.

So, boring, but all good stuff! Commit that recipe. And because we just updated `package.json`, in the other tab, hit Control+C to stop `yarn`. Then run

```
yarn install
```

to get the latest node dependencies... and

```
yarn watch
```

to restart the process. Hey, we're now building with Encore 4! Go team!

Upgrading WebpackEncoreBundle to v2

The biggest change in the Encore world was really the introduction of StimulusBundle. Related to this, in `composer.json`, `symfony/webpack-encore-bundle` has a new major version. Change this to `^2.0`.

Then spin over and, on your main terminal tab, run:

```
composer up
```

By the way, this will fail at the bottom with something related to SensioFrameworkExtraBundle. We... kinda broke our app in the previous chapter while upgrading the framework bundle recipe. We'll fix this in the next chapter, but it's not hurting anything right now.

So what changed between version 1 and 2 of WebpackEncoreBundle? Just one thing: the Twig `stimulus_` helper functions - like `stimulus_controller()` - were removed and moved into the new StimulusBundle. No big deal.

The *real* tricky part is what I mentioned earlier: as a result of the new bundle, a bunch of recipe parts were rearranged between packages. In addition to the `encore_entry()` Twig functions moving to WebpackEncoreBundle's recipe, certain files - like `assets/controllers.json` - were moved from WebpackEncoreBundle's recipe to StimulusBundle's recipe.

This is *all* good: the new situation is cleaner with Stimulus-related files living in *that* bundle's recipe. But... it makes for a bit of a mess when upgrading the recipes.

So let's walk through that. Run:

```
git status
```

Commit these changes... then run

```
composer recipes
```

again. Surprise! There are two new updates! Where did those come from? Well, we just upgraded StimulusBundle and WebpackEncoreBundle and *those* new versions have new *recipe* versions.

symfony/stimulus-bundle Recipe Update

Update `symfony/stimulus-bundle`. This... is where the weird starts. Run:

```
git status
```

We have a conflict in `assets/controllers.json`. This file already existed and the recipe tried to add it. That's because StimulusBundle is now responsible for adding this file... and it's confused because it's already here. Fix this by keeping our `controllers.json` file exactly how it was.

Add that, then `git diff` to see the other changes. Ok, it added an import line to `app.js`. That's also not something we want because... we already have it down here! It's another example of the recipe doing something that's already done. Remove that from the top... then `git add` that file.

And... everything else is fine. It gave us a new `hello_controller.js`, which you can keep or remove, and `symfony.lock`. All good.

symfony/webpack-encore-bundle V2 Recipe Update

Commit that... then onto our final update for WebpackEncoreBundle. This one is particularly strange. Run:

```
git status
```

Two conflicts. Many of the files here *used* to live in WebpackEncoreBundle's recipe, but were moved out of it. So when we upgrade the recipe, it *looks* like a bunch of stuff should be *deleted*. In `assets/app.js`, this file wasn't deleted, but it's trying to remove its guts. Keep it how it was before. Then add it to `git`.

Next up is `package.json`. It's... kind of the same thing: it's trying to delete stuff. Don't let it get away with that! Keep our code... then add this file to `git` too.

Ok, let's see how things look. It wants to delete `assets/bootstrap.js` - we don't want that - and it also wants to delete `controllers.json`. We *also* don't want that. We don't want any of these changes... especially not the letter "G" that I apparently just typed into `package.json`! There's really only one change we care about: in `base.html.twig`. Tada! It's adding back `encore_entry_link_tags()` and `encore_entry_script_tags()`.

That *is* a good change. For the final file - `webpack.config.js` - it wants to remove `enableStimulusBridge()`. Because we *are* using Stimulus, we *do* still want that. Run

```
git reset HEAD
```

to move everything out of the staging area of git, then

```
git checkout assets webpack.config.js
```

to undo those changes. Perfect. We're left with `symfony.lock` and `base.html.twig`. Commit those.

And we are good! We're rocking the latest version of WebpackEncoreBundle with the latest version of WebpackEncore *and* we've gone through that weird, one-time recipe update.

Unfortunately, earlier, we busted our app. So next up: remove SensioFrameworkExtraBundle.

Chapter 4: Goodbye SensioFrameworkExtraBundle

Our app is busted: something about SensioFrameworkExtraBundle. This happened while we were upgrading recipes. In `framework.yaml`, it's the `annotations: false`.

config/packages/framework.yaml

```yaml
// ... line 1
framework:
// ... lines 3 - 4
    annotations: false
// ... lines 6 - 33
```

SensioFrameworkExtraBundle gave us all kinds of features like the `@Route` annotation, security annotation, and something called the param converter. These all relied on the annotation system, which has been replaced by core PHP attributes. When we flipped them to false... the bundle didn't like it.

But hey, that's fine! All those nifty features found a new home in the core of Symfony. So it's time to say a fond farewell to SensioFrameworkExtraBundle.

Uninstalling it

At your terminal run:

```
composer remove sensio/framework-extra-bundle
```

So long, and thanks for all the annotated fish. When it finishes... and we refresh, the site works again!

Checking for SensioFrameworkExtraBundle Features

But... were we using any of its features? I don't know! An easy way to check is by running:

```
git grep FrameworkExtra
```

Nope! It doesn't look like we're referencing any `use` statements directly. If you *are*, it's just a matter of figuring out what new attribute from Symfony replaces that feature.

To help, Symfony has a great documentation page called Symfony Attributes Overview. This shows every PHP attribute from Symfony. For example, SensioFrameworkExtraBundle had a `Security` annotation. Now Symfony has an `IsGranted` attribute that you can use instead.

So if you *are* using something from the old system, find the new way and update.

The New "Param Converter"

Though... there *is* one feature of SensioFrameworkExtraBundle that *didn't* require an annotation... so you may have been using it without realizing. Click into one of the mixes. Notice the URL has a `slug`. The controller for this is `src/Controller/MixController.php`. Down here, the route *does* have a `{slug}` wildcard... but then a `$mix` argument, which is a Doctrine entity.

src/Controller/MixController.php

```php
// ... lines 1 - 12
class MixController extends AbstractController
{
// ... lines 15 - 35
    #[Route('/mix/{slug}', name: 'app_mix_show')]
    public function show(VinylMix $mix): Response
    {
// ... lines 39 - 41
    }
// ... lines 43 - 60
}
```

Behind the scenes, the param converter would automatically query for a `VinylMix` where `slug` equals the `{slug}` in the URL. No annotation needed: it just worked.

The good news is that, as you can see, this magic *still* works! The feature now lives in core. And in most cases, it will silently keep doing its thing, just like before.

If you add an extra letter to the end of the slug to get a 404, we see that the system behind this is `EntityValueResolver`. If you *do* need some extra control, you can configure this with the `#[MapEntity]` attribute.

Next up: I want to upgrade to Symfony 7! But to do that, we need to remove all these deprecations.

Chapter 5: Finding & Eliminating Deprecations

Symfony's deprecation system is a unicorn on the Internet: there's nothing I know of that matches it. It's one of the things that makes Symfony so special and a lot of work goes into!

How Symfony Changes / Deprecates Features

Suppose we, in Symfony, want to change something: like the name of a method. We can't just rename the method, because that would break your code. Instead, we add the new method name, keep the old one, but add a little deprecation code function *in* that old method. We release that on a minor version, like 6.3 or 6.4. Then, you upgrade to that version and, since your code is calling the old method, it hits that deprecation, which triggers a deprecation warning. These warnings are collected, and we can see them in various ways - like on the Web Debug Toolbar.

Your job is to read these and update your code to call the *new* method name. Once all the deprecations are gone, you can safely upgrade to Symfony 7.0. Because, remember, the only difference between Symfony 6.4 and Symfony 7.0 is that the deprecated code paths are gone. In our example, it means that the old method name is finally removed in 7.0.

I *love* this process. It means that Symfony can change things and keep modernizing, *and* we can update our apps in a safe and predictable way. It's the best.

Hunting Down Deprecations

So today, we're deprecation detectives: on a mission to hunt these down and eliminate them. To get started, I'll manually clear my cache directory

```
rm -rf var/cache/*
```

so that when we go over and refresh the homepage, this will *build* the cache. Some deprecation warnings only happen while your cache is being built. Ok, it looks like we have three warnings left after updating our recipes. Nice!

Removing symfony/templating

Open those up. The first is related to some templating helper class being deprecated. That's not something I remember using in my code. Look at the trace. Not super helpful. Here's the helper class... called by a class loader.

This tells me that something is trying to use this class... and the class is entirely deprecated. In fact, this entire `symfony/templating` component is deprecated: it doesn't exist at all in Symfony 7.0! There's a pretty good chance you never used it anyway... and I'm *not* using it in *my* app. So then, who is?

To find out, go to the command line and run:

```
composer why symfony/templating
```

Ah! This is required by `knplabs/knp-time-bundle`. Click the link to jump to our installed version: 1.20.0. But that's not the latest version: it now has a version 2.2. We're way behind! The major version 2.0 modernizes the code... and there's a good chance that included removing the templating dependency. In fact, we see it down here.

This *is* a new major version of the package, so we *do* need to check the changelog or release notes for any backwards compatibility breaks that might affect us.

So the interesting thing about this first deprecation is that... it wasn't something that *we* were calling directly. It's an indirect deprecation: caused by a library we're using. And that's pretty common. To be ready for Symfony 7, we need to upgrade this bundle.

In composer.json, search for "time"... then change this to the latest `^2.2`. Spin over and run:

```
composer up
```

It upgrades the package... and removes `symfony/templating`!

DoctrineFixturesBundle False Deprecation

Ok, clear the cache again, close some tabs, and let's go to the "browse mixes" page because that connects to the database. This time, we see two deprecations. Open those up and dive in. The first has something to do with data fixtures. If we look at the trace, it's not super obvious, but it's coming from DoctrineFixturesBundle. This is a tricky one: I had to dive into the DoctrineFixturesBundle GitHub repository to find a conversation. This time, the deprecation is a false warning! The deprecation layer that was added to the bundle wasn't done *quite* correctly. The maintainer confirms that it's fine... so when we upgrade to Symfony 7, things won't break.

This is an odd situation, but it shows that hunting deprecations can be tricky!

Deprecations from Doctrine

The final deprecation is longer and follows a different format. So far, each message has included the package the deprecation lives in and the version it was deprecated in. But we don't see that down here. And, at the end, it references an issue from the `doctrine/orm` repository.

Ah! This deprecation isn't coming from Symfony: it's coming from `doctrine/orm`! This tells us that we're going to need to make a change to our code - at some point - before upgrading to the next major version of that package. We're only focused on upgrading Symfony today, so this is *also* a deprecation we can ignore.

So... yeah I think we're good. Our app is pretty small, but as I click around, the only deprecations I see are those that we just looked at.

Deprecation Log on Production

But... how can we be sure that there's not a page we forgot to check or a form submit that triggers a deprecation? The answer: logging.

In `config/packages/monolog.yaml`, at the bottom, we have the production logging config. The main handler is the `nested` handler: this logs errors on production. This logs them to stderr, or you could change that to a file.

```yaml
// config/packages/monolog.yaml
// ... lines 1 - 39
when@prod:
    monolog:
        handlers:
// ... lines 43 - 48
            nested:
                type: stream
                path: php://stderr
                level: debug
                formatter: monolog.formatter.json
// ... lines 54 - 63
```

The point is: you're hopefully collecting your production errors somewhere. At the bottom, there's another handler called `deprecation`. This logs all deprecation notices to the *same* place. So in your production error logs, you should *also* see deprecations warnings.

```yaml
// config/packages/monolog.yaml
// ... lines 1 - 39
when@prod:
    monolog:
        handlers:
// ... lines 43 - 48
            nested:
                type: stream
                path: php://stderr
                level: debug
                formatter: monolog.formatter.json
// ... lines 54 - 57
            deprecation:
                type: stream
                channels: [deprecation]
                path: php://stderr
                formatter: monolog.formatter.json
```

So: fix all the deprecations you can find, deploy to production, wait a day or two, then check your logs to see if there are any deprecations. Once there aren't, you're safe to move to Symfony 7.0. Let's do that upgrade next!

Chapter 6: Upgrading to Symfony 7

All the relevant deprecations are gone. So we are ready for Symfony 7.0!

Updating composer.json

Doing the actual upgrade is... almost disappointingly easy. In `composer.json` replace `6.4.*` with `7.0.*`.

```
composer.json
{
// ... lines 2 - 4
    "require": {
// ... lines 6 - 16
        "symfony/asset": "7.0.*",
        "symfony/console": "7.0.*",
        "symfony/dotenv": "7.0.*",
// ... line 20
        "symfony/framework-bundle": "7.0.*",
        "symfony/http-client": "7.0.*",
// ... line 23
        "symfony/proxy-manager-bridge": "7.0.*",
        "symfony/runtime": "7.0.*",
        "symfony/twig-bundle": "7.0.*",
// ... lines 27 - 28
        "symfony/yaml": "7.0.*",
// ... lines 30 - 31
    },
// ... lines 33 - 80
    "extra": {
        "symfony": {
// ... line 83
            "require": "7.0.*",
// ... line 85
        }
    },
    "require-dev": {
// ... line 89
        "symfony/debug-bundle": "7.0.*",
// ... line 91
        "symfony/stopwatch": "7.0.*",
        "symfony/web-profiler-bundle": "7.0.*",
// ... line 94
    }
}
```

That's it. Spin over and run:

```
composer up
```

Finding Blocking Packages

Brace yourselves because this might not work. Yup! Something is blocking the update! The first thing I see is `babdev/pagerfanta-bundle`. Apparently, it works with Symfony 4, 5 and 6, but not 7.

There's a good chance that I probably need to upgrade this to a new version that *does* support Symfony 7. Run:

```
composer outdated
```

Sure enough: there are three pagerfanta packages that all have a new major version. In `composer.json`, search for pagerfanta. Change all of these to `^4.0` to get that new major version.

composer.json

```
{
// ... lines 2 - 4
    "require": {
// ... lines 6 - 8
        "babdev/pagerfanta-bundle": "^4.0",
// ... lines 10 - 13
        "pagerfanta/doctrine-orm-adapter": "^4.0",
        "pagerfanta/twig": "^4.0",
// ... lines 16 - 31
    },
// ... lines 33 - 95
}
```

And because this *is* a major version upgrade, I won't do it, but you should check the repository for each package and find the changelog or release notes that talk about any backwards compatibility breaks between version 3 to 4.

Ok, try the update again:

```
composer up
```

And... *still* no dice! Hmm: it says that the *root* `composer.json` - meaning *our* `composer.json` - requires `symfony/proxy-manager-bridge` `7.0.*` but it didn't find a version 7.

Sure enough, this package lives directly in *our* `composer.json` file. Proxies are something that Doctrine uses behind the scenes to load lazy relationships. Recently, Symfony added its *own* version of proxies called "ghost objects". They're spooky cool. Anyway, this proxy package isn't needed anymore. It was originally added to our app *way* back when we installed Doctrine: it *used* to be part of the `orm-pack`.

Give it the boot! Then try `composer up` again:

```
composer up
```

This time... it works! Look at all those beautiful Symfony 7 updates! And best of all, when we go to the site, it works too! Of course it does! We handled the deprecations, so there are no surprises when we finally get to 7.0.

Any other Packages to Update

At this point, I like to check to see what *other*, non-Symfony packages are outdated. Run `composer outdated` again:

```
composer outdated
```

Woh! Just two! `doctrine/lexer` and a `php-parser`. To find out why this didn't go to version 3, copy that package name, and run

```
composer why-not doctrine/lexer 3.0
```

Hmm: our version of `doctrine/orm` requires `doctrine/lexer` version 2. And since we didn't see `doctrine/orm` as an outdated package, it means that there simply isn't a version of `doctrine/orm` yet that works with `doctrine/lexer` 3. And that's fine! That's a low-level package, and we're in no rush.

The other package - `php-parser` - I can tell you, without even looking, that this is required by `symfony/maker-bundle`. In its next release, version 5 will be allowed.

New Version Recipes

Because we just updated some packages, run:

```
composer recipes
```

Hey! There are two new recipe updates available! To upgrade, first commit our changes... complete with an emoji to celebrate... then run:

```
composer recipes:update
```

And `git diff --cached` to see the changes. This is *cool*: a bunch of lines gone. These were removed because they are now the *default* values. The `session` key no longer needs this stuff - they're all the default values... and same for `php_errors` and `handle_all_throwables`. It's just a nice config clean up.

Commit that, then run `recipes:update` one more time:

```
composer recipes:update
```

Check the changes. Same thing: it removes a config option that is now the default. Commit that. Our project is now a *little* bit cleaner.

So we're on Symfony 7, our app is working and our recipes are updated!

Changing the Namespace for #[Route]!

While we're here, inside a controller, it's highlighting the `Route` attribute:

> *“Symfony Annotation namespace will be deprecated in Symfony 6.4 /7.0.”*

Look at the `use` statement: it has `Annotation` in the namespace! This class isn't *yet* deprecated, but it will be soon. And fixing it is simple. Delete the `use` statement, go down here, click on the class, hit Alt+Enter, Import Class, then get the one from the `Attribute` namespace.

Copy that... then repeat in the other two controller files. This will save us a deprecation in the future.

Now that we're on Symfony 7, I want to do something *optional*, but really cool: I want to remove Webpack Encore and replace it with AssetMapper.

Chapter 7: Migrating Encore -> AssetMapper

Symfony 6.3 came with a new component called AssetMapper... and I love it! Okay, I work on it... so I'm totally *not* objective... but trust me it's amazing! It lets us write modern JavaScript and css with *no* build system. We have an Asset Mapper tutorial and a more recent LAST Stack tutorial where we build cool stuff with it.

AssetMapper Vs Webpack Encore?

AssetMapper is a replacement for Webpack Encore. Encore isn't going to die super soon, but I definitely caught it browsing some retirement brochures!

So I know what you're wondering:

> *"Should I convert my app from Webpack Encore to AssetMapper?"*

The short, but not satisfying answer is... it's up to you. AssetMapper is more modern, it's easier to use and if you're frustrated with slow builds from Encore, that's a great reason to switch. But if Encore is working fine, there's no *huge* reason to do all the work of converting to AssetMapper. Also, if you use React or Vue, you'll want to stay with Encore because those *do* still require a build step.

Removing Webpack Encore

But let's convert! Head over to your terminal and find that tab where `yarn watch` is doing its thing. Stop that with Ctrl+C and close that tab. We do *not* need a build system - so that second tab is *not* coming back.

Then run:

```
composer remove symfony/webpack-encore-bundle
```

This will remove that package... but more important: its recipe will *uninstall* itself! It feels great: `package.json` gone, `webpack.config.js` gone, the `encore_entry_` functions in `base.html.twig` gone.

But... it also deleted `app.js` and `app.css`. We *do* want those files, so run

```
git checkout assets/
```

to get them back. But everything else looks good! Run:

```
git diff
```

In the old `package.json`, the dependencies here were related to Webpack Encore and we will *not* need those. But some of these are for our frontend, and we *will* re-add those via AssetMapper.

Ok, lock in those changes with a commit... then throw a party by removing `node_modules`, `public/build/` and the yarn error file. Oh, we can also remove `yarn.lock`. Gorgeous!

Installing AssetMapper

Now let's install AssetMapper:

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

Its recipe does a bunch of interesting things. We won't go too deep into how AssetMapper works - we have other tutorials for that - but let's explore. In `.gitignore`:

```
.gitignore
// ... lines 1 - 10
###> symfony/asset-mapper ###
/public/assets/
/assets/vendor
###
```

it ignores the final location of the built assets and where the vendor files live. And in `templates/base.html.twig`, it added an `importmap()` function that will output CSS and JavaScript.

```
templates/base.html.twig
// ... line 1
<html>
    <head>
// ... lines 4 - 9
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
{% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}
    </head>
// ... lines 17 - 82
</html>
```

It also gave us an `importmap.php` file.

importmap.php

```php
<?php

/**
 * Returns the importmap 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,
    ],
    '@hotwired/stimulus' => [
        'version' => '3.2.2',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',
    ],
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
];
```

This is, effectively, the new `package.json`: the home for 3rd party packages. And hey! It already added Stimulus and Turbo! Those are two of the packages from `package.json` that we *do* need.

Will this work? Refresh and... kinda? We don't have Bootstrap CSS... which is why it looks terrible. But I *can* see that `assets/styles/app.css` *is* being loaded: that's giving us some basic styles. But we need to fix these imports.

```
assets/styles/app.css
@import '~bootstrap';
@import '~@fortawesome/fontawesome-free/css/all.css';
@import '~@fontsource/roboto-condensed';
// ... lines 4 - 40
```

Onwards we go! Let's roll up our sleeves and nail down the last few steps to get AssetMapper up and running next.

Chapter 8: Encore -> AssetMapper Part 2

Getting 3rd party CSS files working is one of the *trickier* things to do in AssetMapper. Importing them like this isn't going to work.

Installing Bootstrap CSS

Let's focus on Bootstrap first. This is a third-party package, and we install third-party packages by saying `bin/console importmap:require` the package name:

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

Bootstrap is especially interesting because it grabs the JavaScript package, a *dependency* of the JavaScript package, and it *also* noticed that this package commonly has a CSS file... so it grabbed that *too*. All three things were added to `importmap.php`.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 29
    'bootstrap' => [
        'version' => '5.3.2',
    ],
    '@popperjs/core' => [
        'version' => '2.11.8',
    ],
    'bootstrap/dist/css/bootstrap.min.css' => [
        'version' => '5.3.2',
        'type' => 'css',
    ],
];
```

We're not using the bootstrap *JavaScript* in this project. So we *could* delete this. But I'll leave it because it's not hurting anything. The real star, however, is this CSS file. Copy its path. And in `app.css`, remove the top line.

You *can* import third party CSS with AssetMapper, but *can't* do it from inside another CSS file. Well, you technically can, but life is easier if we do it from `app.js`. Say `import`, then paste.

```
assets/app.js
// ... lines 1 - 8
import 'bootstrap/dist/css/bootstrap.min.css';
import './styles/app.css';
// ... lines 11 - 16
```

And now... Bootstrap springs to life!

Adding FontAwesome

Next up is FontAwesome. Notice that we're grabbing a *specific* CSS file from the package. One big difference between Encore and AssetMapper is that if you need to import a specific file from a package, you need to `importmap:require` *that* file, not just the package in general. Watch `bin/console importmap:require` and paste:

```
php bin/console importmap:require @fortawesome/fontawesome-free/css/all.css
```

That grabs this *one* CSS file, downloads it into the project and adds it to `importmap.php` right here. If you're curious, these files are downloaded into an `assets/vendor/` directory.

Head into `app.css`, remove that line and add another import for that path.

```
assets/app.js
// ... lines 1 - 8
import 'bootstrap/dist/css/bootstrap.min.css';
import '@fortawesome/fontawesome-free/css/all.css';
import './styles/app.css';
// ... lines 12 - 17
```

And *that* works! Though, on the topic of FontAwesome, I don't recommend using FontAwesome like this anymore. Instead, use FontAwesome kits. *Or*, better, render an inline SVG. Hopefully we'll have an icon package soon from Symfony UX to make that easier.

Adding CSS Fonts

The last item in `app.css` is a font. This is trickier. If we run `importmap:require` followed by *only* a package name - no path - it will always download the package's main JavaScript file. You only get a CSS file if you `importmap:require` a path *to* a CSS file, like we just did.

Ok, I know earlier we ran `import:require bootstrap` and that *did* give us a CSS file. So, let me be more clear. If you run `importmap:require packageName`, you'll get the *JavaScript* for that package. In *some* cases, like Bootstrap, the package advertises that it has a CSS file. When that happens, AssetMapper sees that and, effectively, runs `importmap:require bootstrap/dist/css/bootstrap.min.css` automatically... just to be helpful.

Anyway, I know we need a CSS file. With Encore, if you imported a package from inside a CSS file, Encore would try to find the CSS file in the package and import that. This doesn't happen with AssetMapper: we need to figure out *what* the path is to the CSS file then require *that*.

I like to do this at jsDelivr.com. This is the CDN that AssetMapper uses behind the scenes to fetch packages. Search for the package. It shows up, but there's one below from `@fontsource-variable`. Variable fonts can be a bit more efficient, so let's change to that. Inside, hey! It advertises the main CSS file! If you wanted a different file, you could click the Files tab and navigate to find what you need.

Copy this path all the way down to the package name, then spin over and run `importmap:require` and paste. But we don't need the version: just the package, then the path:

```
php bin/console importmap:require @fontsource-variable/roboto-condensed/index.m
```

Copy that and hit enter. It downloads the CSS file and adds an entry to `importmap.php`.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 43
    '@fontsource-variable/roboto-condensed/index.min.css' => [
        'version' => '5.0.1',
        'type' => 'css',
    ],
];
```

Finally, remove the import from `app.css` and import it from `app.js`.

```
assets/app.js
// ... lines 1 - 8
import 'bootstrap/dist/css/bootstrap.min.css';
import '@fortawesome/fontawesome-free/css/all.css';
import '@fontsource-variable/roboto-condensed/index.min.css';
import './styles/app.css';
// ... lines 13 - 18
```

Oh, and because we changed to the variable font, in `app.css`, update the font family to `Roboto Condensed Variable`.

Over on the site, watch the font when I refresh. Got it! Grabbing those third party CSS files might be the *trickiest* thing you'll do in AssetMapper.

Oh, and if you're using Sass or Tailwind, there are Symfonycasts bundles to support both of those in AssetMapper.

Adding the .js Extension

Now that styling is working, let's look into our JavaScript. In the console, we have an error: a 404 for something called `bootstrap`. That's coming from `app.js`: from this import line. To fix this, open `app.js` and add `.js` to the end.

```
assets/app.js
// ... lines 1 - 13
// start the Stimulus application
import './bootstrap.js';
// ... lines 16 - 18
```

With Webpack Encore, we're running inside a Node environment. And Node lets you cheat: if the file you're importing ends in `.js`, you don't need to include the `.js`. But in a *real* JavaScript environment, like your browser, you *can't* do that: the `.js` *is* needed.

This is probably the biggest change you'll need to make when converting.

stimulus-bridge -> stimulus-bundle

Try the page now. Next error! And it's important:

> *"Failed to resolve module specifier `@symfony/stimulus-bridge`."*

This means that, somewhere, we're importing this package... but the package doesn't exist in `importmap.php`.

There are two types of imports. First, if an import starts with `./` or `../`, it's a relative import. Those are simple: you're importing a file next to this file. The second type is called a *bare* import. This is when you're importing a package or a file in a package. For these, the string inside the import must *exactly* exist in `importmap.php`. If it doesn't, you'll see this error.

The source of our error is `bootstrap.js`. See this `@symfony/stimulus-bridge`? That does *not* exist in `importmap.php`. The solution, usually, is to install this.

But in this case, the package is specific to Webpack Encore and the fix is related to our migration. Change this to `@symfony/stimulus-bundle`.

assets/bootstrap.js

```
import { startStimulusApp } from '@symfony/stimulus-bundle';
// ... lines 2 - 8
```

And lo and behold: that string *does* live inside `importmap.php`! Below, the next line simplifies.

assets/bootstrap.js

```
import { startStimulusApp } from '@symfony/stimulus-bundle';

// Registers Stimulus controllers from controllers.json and in the controllers/ directory
export const app = startStimulusApp();
// ... lines 5 - 8
```

But it does the same thing as before: starts the Stimulus app and load our controllers. If you start a new Symfony app, you get all this with the recipe. But since we're converting, we need to do a bit more work.

Installing Missing Packages

Refresh now. We get the exact same error but with a different package: `axios`. You know the drill: somewhere, we're importing this... but it doesn't live in `importmap.php`. In this case, it's coming from `song-controls_controller.js`.

And *this* time, the fix *is* to install this package! Spin over and run

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

That adds `axios` to `importmap.php` and now... our app is alive! This is powered by AssetMapper! We have a performant, modern frontend all with no build system.

Downgrading a Dependency

Oh, but look at the footer: the text is darker than it used to be. Before, I was using bootstrap 5.1. But when we installed `bootstrap` with AssetMapper, it grabbed the latest 5.3. And apparently something changed!

I could figure out *what* changed and fix this... But we can also downgrade. Update the version in `importmap.php` to 5.1.3.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 29
    'bootstrap' => [
        'version' => '5.1.3',
    ],
// ... lines 33 - 35
    'bootstrap/dist/css/bootstrap.min.css' => [
        'version' => '5.1.3',
        'type' => 'css',
    ],
// ... lines 40 - 50
];
```

If we just did that and refreshed, nothing would change: the newer version is *still* downloaded into `assets/vendor/`. To sync that directory with `importmap.php`, run:

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

Think of this is as the `composer install` of the AssetMapper world. It noticed that we changed two packages and downloaded those. And just like that, we've crossed the finish line! We're running AssetMapper!

Next up, let's take three minutes to modernize & simplify our JavaScript.

Chapter 9: Modernizing with fetch() and await

This chapter isn't related to upgrading Symfony. But the rest of our code - including JavaScript - deserves to be modernized too!

Using fetch() instead of axios

Inside `song-controls_controller.js`, I originally used `axios` to make Ajax calls.

```
assets/controllers/song-controls_controller.js
// ... lines 1 - 11
import axios from 'axios';
// ... line 13
export default class extends Controller {
// ... lines 15 - 18
    play(event) {
// ... lines 20 - 21
        axios.get(this.infoUrlValue)
// ... lines 23 - 26
    }
}
```

I don't do that anymore. Instead, use the built-in `fetch()` function.

Remove `axios` with:

```
php bin/console importmap:remove axios
```

It's gone from `importmap.php`. Then delete the `import`... and this comment while we're here. Replace `axios.get()` with just `fetch()`. Then, to see if this is working, `console.log(response)`.

```
assets/controllers/song-controls_controller.js
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 7
    play(event) {
// ... lines 9 - 10
        fetch(this.infoUrlValue)
            .then((response) => {
                console.log(response);
                const audio = new Audio(response.data.url);
                audio.play();
            });
    }
}
```

Over in browser-land, smash that play button to trigger the method. Cool! The last two lines aren't working, but we see the response! It *did* make an Ajax call.

When I originally wrote this, I used `.then()` to handle the Promise. I don't often use that anymore to handle asynchronous code. Instead, I use the simpler `await`.

Using await & async

In front of `fetch`, say `const response = await fetch()`. Then copy the inside of the callback and put it right after.

```
assets/controllers/song-controls_controller.js
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 7
    async play(event) {
// ... lines 9 - 10
        const response = await fetch(this.infoUrlValue);
        console.log(response);
        //const audio = new Audio(response.data.url);
        //audio.play();
    }
}
```

This says: make the `fetch()` call, wait for it to finish, and *then* run this code. It's much simpler to read and write.

Though, you probably noticed my angry editor:

> *“the await operator can only be used in an async function.”*

To use `await`, we need to add `async` before the function that we're directly inside. I won't go into the details, but this advertises that *our* function is now asynchronous. If you called it and wanted the return value, you'd need to `await` that call as well.

But in our case, Stimulus is calling this method... and it definitely does *not* care about our return value. So adding `async` doesn't change anything.

When we try it... the *same* result, without the callback.

So let's finish this: `const data = await response.json()`.

This takes the JSON from the response of our API endpoint and converts it into an object. And yea, it's *also* an asynchronous function, so `await` comes in handy again! Below, pass `data.url` to `Audio`.

```
assets/controllers/song-controls_controller.js
// ... lines 1 - 2
export default class extends Controller {
// ... lines 4 - 7
    async play(event) {
        event.preventDefault();

        const response = await fetch(this.infoUrlValue);
        const data = await response.json();
        const audio = new Audio(data.url);
        audio.play();
    }
}
```

Then celebrate, that sweet, sweet Rickroll. Modern code, no build system: life is good.

Now that we're upgraded, let's take a tour into some of my favorite new features, starting with autowiring goodies that might mean you'll never edit `services.yaml` again.

Chapter 10: New Autowiring Attributes

So what's new in Symfony 7? Nothing! The real question is, what's new in Symfony 6.4? Or maybe, what's new in 6.3 or 6.2 that... maybe we missed?

Quick Tour of New Features

The best place to find this stuff... is the Symfony blog. Javier does a fantastic job with every release, uncovering the most important features.

I've pulled up a few of my favorites, like the workflow profiler. If you use the workflow component, you can now see a crazy-cool visualization of your workflow inside the profiler.

There are also some changes to the logout system - just to make life simpler... some new constraints, like `PasswordStrengthConstraint` and another that prevents suspicious characters, like zero width space characters. This can be used to prevent someone from creating a username that *looks* like someone else's.

If you're building an API, there's an excellent `debug:serializer` command to see all the metadata for a class.

And finally, the new `Webhook` and `RemoteEvent` components, which deserve their own tutorial. So we'll save that for another time.

These are just a few of my favorite features, but you can look at *everything* by going to the "Living on the Edge" section of the blog and filtering by the version. A great way to nerd out.

The Autowire Attribute

But I *do* want to walk through a few new features together, starting with improvements to the autowiring system. These happened over the last several versions of Symfony and... they do a lot of things. The overall effect is that you'll probably never need to go into `services.yaml` again.

Let's dive in! In an old tutorial, I added this `bind` for an `$isDebug` argument.

config/services.yaml

```yaml
// ... lines 1 - 12
services:
// ... line 14
    _defaults:
// ... lines 16 - 17
        bind:
            'bool $isDebug': '%kernel.debug%'
// ... lines 20 - 32
```

The reason I did that lives in `src/Controller/VinylController.php`: I gave this controller an `$isDebug` argument... which isn't autowirable.

src/Controller/VinylController.php

```php
// ... lines 1 - 13
class VinylController extends AbstractController
{
    public function __construct(
        private bool $isDebug
    )
    {}
// ... lines 20 - 56
}
```

In `services.yaml`, remove the `bind`.

When we refresh, error! It says:

> *“Hey you silly person: you have an `$isDebug` argument on a service, but I have no idea what to pass to that.”*

Hence, why we had the `bind`. Starting a few Symfony versions ago, we now have an `Autowire` attribute. If you have an argument that can't be autowired, this is your friend. Add it before the arg and define what you want. This can be a service, an expression, an environment variable, a parameter, a kitten, whatever. We want a param: `kernel.debug`.

```
src/Controller/VinylController.php
// ... lines 1 - 8
use Symfony\Component\DependencyInjection\Attribute\Autowire;
// ... lines 10 - 14
class VinylController extends AbstractController
{
    public function __construct(
        #[Autowire('%kernel.debug%')]
        private bool $isDebug
    )
    {}
// ... lines 22 - 58
}
```

Inside, `dump($this->isDebug)` to make sure it's working.

And... it is! Autowire is my new favorite attribute. But if you hold command or control to open this class... then double-click on the `Attribute` directory, we see a whole list of cool, dependency-injection related attributes. `Exclude` is a way to exclude a class from being auto-registered as a service. `Autoconfigure` and `AutoconfigureTag` are both ways to configure *options* on your service. Put this above your class - or even above an interface - and the options will apply to the service or services that implement that interface.

There's also `AutowireIterator` and `AutowireLocator`. If you have a set of services that implement a tag, you can use `AutowireIterator` to get those services passed to you as an iterator, or `AutowireLocator` to get them passed to you as a locator, basically an associative array of services.

Trying AutowireIterator

For example, pretend that, in `VinylController`, we want to get an iterable of *every* console command in our app. Say `private iterable $commands`. And to prove this is working, foreach over `$this->commands` as `$command`... then dump the object.

```
src/Controller/VinylController.php
// ... lines 1 - 15
class VinylController extends AbstractController
{
    public function __construct(
// ... lines 19 - 21
        private iterable $commands,
    )
    {
        foreach ($this->commands as $command) {
            dump($command);
        }
    }
// ... lines 29 - 65
}
```

If we stopped now, we'd get the classic error that says:

> *"I have no idea what to pass for this `$commands` argument!"*

We want an iterable of every services that implement a specific tag. Grab those with `#[AutowireIterator]`, then the tag we want: `console.command`.

```
src/Controller/VinylController.php
// ... lines 1 - 9
use Symfony\Component\DependencyInjection\Attribute\AutowireIterator;
// ... lines 11 - 15
class VinylController extends AbstractController
{
    public function __construct(
// ... lines 19 - 20
        #[AutowireIterator('console.command')]
        private iterable $commands,
    )
    {
        foreach ($this->commands as $command) {
            dump($command);
        }
    }
// ... lines 29 - 65
}
```

And just like that, we got them! We see all 102 console commands in my app. I know, it's a silly example, but isn't that cool?

Back in the controller, undo that.

```
src/Controller/VinylController.php
// ... lines 1 - 14
class VinylController extends AbstractController
{
    public function __construct(
        #[Autowire('%kernel.debug%')]
        private bool $isDebug,
    )
    {}
// ... lines 22 - 58
}
```

Next up: let's talk about a few subtle, but powerful new ways to fetch request data like query parameters and the request payload.

Chapter 11: MapQueryParameter & Request Payload

The next new stuffs I want to talk about are related to grabbing data from the request. That's normally... kind of boring work. But the new features are pretty darn cool.

The MapQueryParameter Attribute

For example, add a `?query=banana` to the URL. To fetch that in our controller, we would historically type-hint an argument with `Request` then grab it from there. And while that still works, we can now add a `?string $query` argument. To tell Symfony that this is something it should grab from a query parameter, add an attribute in front: `#[MapQueryParameter]`.

That's it! Dump `$query` to prove it works.

src/Controller/VinylController.php

```php
// ... lines 1 - 11
use Symfony\Component\HttpKernel\Attribute\MapQueryParameter;
// ... lines 13 - 15
class VinylController extends AbstractController
{
// ... lines 18 - 24
    public function homepage(
        #[MapQueryParameter] string $query = '',
    ): Response
    {
        dump($query);
// ... lines 30 - 42
    }
// ... lines 44 - 62
}
```

Back in the web browser world, refresh. In the web debug toolbar... got it!

Validation from the Type-Hint

The attribute *does* also have some options. For example, if your query parameter is called something different from your argument, you could put that here.

And beyond just grabbing the value from the request, this system *also* performs validation. Watch: duplicate this and add an `int $page = 1` argument. Oh, and I meant to make the `$query` argument optional so it doesn't *need* to be on the URL. Below, dump `$page`.

```
src/Controller/VinylController.php
// ... lines 1 - 15
class VinylController extends AbstractController
{
// ... lines 18 - 24
    public function homepage(
        #[MapQueryParameter] string $query = '',
        #[MapQueryParameter] int $page = 1,
    ): Response
    {
        dump($query, $page);
// ... lines 31 - 43
    }
// ... lines 45 - 63
}
```

Ok, if we add `?page=3` to the URL... no surprise: it dumps `3`. But it *is* nice that we get an *integer* 3: not a string. Now try `page=banana`. A 404! The system sees we have an `int` type and performs validation.

The filter_var() Function

This entire system is handled by something called the `QueryParameterValueResolver`. So if you really want to dig in, check that class. Internally, it uses a PHP function called `filter_var()` to do the validation. This is not a function I'm very familiar with, but it's quite powerful. You pass it a value, one or more filters... and it tells you whether that value *satisfies* those filters. You can also pass options to control the filters.

If you don't do anything extra, the system reads our `int` type-hint, and passes a filter to `filter_var()` that requires it to be an `int`. That's why this fails.

Validating an int is in a Range

But we *can* get fancier. Add an argument called `$limit` that defaults to 10. Dump this below. But I want the limit to be between 1 and 10. To force that, pass two options special to `filter_var`: `min_range` set to 1 and `max_range` set to 10.

```
src/Controller/VinylController.php
```

```php
// ... lines 1 - 15
class VinylController extends AbstractController
{
// ... lines 18 - 24
    public function homepage(
        #[MapQueryParameter] string $query = '',
        #[MapQueryParameter] int $page = 1,
        #[MapQueryParameter(options: ['min_range' => 1, 'max_range' => 10])] int $limit = 10,
    ): Response
    {
        dump($query, $page, $limit);
// ... lines 32 - 44
    }
// ... lines 46 - 64
}
```

Let's try it! Say `?limit=3`. That works like we expect. But when we try `limit=13`. `filter_var()` fails and we get a 404! I love that!

Grabbing Array Query Parameters

This can even be used to handle arrays. Copy and create one more argument: an array of `$filters` that defaults to an empty array. Dump that.

```
src/Controller/VinylController.php
// ... lines 1 - 15
class VinylController extends AbstractController
{
// ... lines 18 - 24
    public function homepage(
        #[MapQueryParameter] string $query = '',
        #[MapQueryParameter] int $page = 1,
        #[MapQueryParameter(options: ['min_range' => 1, 'max_range' =>
10])] int $limit = 10,
        #[MapQueryParameter] array $filters = [],
    ): Response
    {
        dump($query, $page, $limit, $filters);
// ... lines 33 - 45
    }
// ... lines 47 - 65
}
```

At the browser, add `?filters[]` equals banana, `&filters[]` equals apple. Check out that array in the web debug toolbar! It also works for associative arrays: add `foo` and `bar` between the `[]`. Yup! An associative array.

It's just a really well-designed feature for fetching query parameters.

Request Body

Also, if you need to fetch the *body* of a request, in Symfony 6.3, there's a new method called `$request->getPayload()`. Building an API? When your client sends JSON in the body, use `$request->getPayload()` to decode that into an associative array. That's nice! But also, if your user submits a normal HTML form, `$request->getPayload()` works there too. It detects that an HTML form is being submitted and decodes the `$_POST` data to an array. So no matter if you're using an API or a normal form, we have a uniform method to fetch the payload of the request. Small, but nice.

MapRequestPayload

Speaking of JSON, it's also common to use the serializer to deserialize the payload into an object. That relates to another new feature called `#[MapRequestPayload]`.

In this case, `__invoke` is the controller action. This says: take the JSON from the request and deserialize it into a `ProductReviewDto`, which is the example class above. After sending the JSON through the serializer, it even performs validation. So another well-thought-out feature.

Ok, that's enough for request stuff! Next up, let's test drive a new feature in 6.4: the ability to profile console commands.

Chapter 12: Profiling Commands

In the dev environment on our site, we get the web debug toolbar. And more importantly, the profiler, which is packed full of goodies. Even if our app is entirely an API, we can go directly to `/_profiler` to check out the profiler for any API request.

This is one of Symfony's killer features. And for 6.4, Symfony contributor Jules Pietri wondered: why can't we have this for console commands?

And now, we do! It's meant to be used for your custom console commands that might be big or complex, but we can also use it with *core* commands.

Triggering a Profile: --profile

Spin over and run:

```
php bin/console debug:container
```

If you run a normal command, it won't activate the profiler system and collect info. To trigger that, you need to run the command with `--profile`.

```
php bin/console debug:container --profile
```

Nothing *looks* different, but that *did* just activate the profiler... which collected info and stored it... somewhere. But... it's not obvious where we can go to see it!

So what you really want to do is pass `-v`:

```
php bin/console debug:container --profile -v
```

Now, at the bottom, it includes the unique token that can be used in the profiler URL. But, really, be lazier and run with `-vvv`:

```
php bin/console debug:container --profile -vvv
```

This time, we get a *link* - and even details about memory and time. I'll click the link and... it doesn't work. It's *almost* the right URL, but my terminal doesn't know what port my local web server is using. Copy that token, then... go to the profiler for any request, paste the token in the URL and... so cool!

Exploring the Profiler

We see info about the command, the input, output... and most importantly, we have the normal profiler sections! One interesting one is events: showing the actual events that were dispatched and the listeners for each one. These are *totally* different from the events that are triggered during a request, so it's cool to see them.

Now, you probably noticed that most of the profiler sections are grayed out. But if you *did* render a Twig template... or make an HTTP request or make a database query, these *would* be activated.

Even with this simple command, we unlock the performance section. Not a lot here in *this* case, but it makes me feel dangerous.

So that's it! Another, cool, well-thought-out feature. I'd love to see how people end up using this.

Ok, on to our final topic: let's experiment with one of Symfony's best new components: Scheduler.

Chapter 13: New Component: Scheduler

One of the coolest new components is Scheduler, which came from Symfony 6.3. If you need to trigger a recurring task, like generate a weekly report, send some sort of heartbeat every 10 minutes, perform routine maintenance... or even something custom and weird, this component is for you. It's really neat! It deserves its own tutorial, but we'll worry about that later. Let's take it for a test drive.

Installing Scheduler

At your command line, install it with:

```
composer require symfony/scheduler symfony/messenger
```

Scheduler relies on Messenger: they work together! The process looks like this. You create a message class and handler, like you normally would with Messenger. Then you tell Symfony:

> *"Yo! I want you to send this message to be handled every seven days, or every one hour... or something weirder."*

Creating the Message Class & Handler

This means that step one is to generate a Messenger message. Run:

```
php bin/console make:message
```

Call it `LogHello`. Cool! Over here, it created the message class - `LogHello`

```
src/Message/LogHello.php
// ... lines 1 - 2
namespace App\Message;

final class LogHello
{
    public function __construct()
    {
    }
}
```

and its handler, whose `__invoke()` method will be called when `LogHello` is dispatched through Messenger.

```
src/MessageHandler/LogHelloHandler.php
// ... lines 1 - 2
namespace App\MessageHandler;

use App\Message\LogHello;
use Symfony\Component\Messenger\Attribute\AsMessageHandler;

#[AsMessageHandler]
final class LogHelloHandler
{
    public function __construct()
    {
    }

    public function __invoke(LogHello $message)
    {
    }
}
```

In `LogHello`, give it a constructor with `public int $length`.

```
src/Message/LogHello.php
// ... lines 1 - 2
namespace App\Message;

final class LogHello
{
    public function __construct(public int $length)
    {
    }
}
```

This will help us figure out which message is being handled and when. In the handler, *also* add a constructor so we can autowire `LoggerInterface $logger`.

```
src/MessageHandler/LogHelloHandler.php
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 9
final class LogHelloHandler
{
    public function __construct(private LoggerInterface $logger)
    {
    }
// ... lines 15 - 19
}
```

Down in the method, use `$this->logger->warning()` - just so these log entries are easy to see - then `str_repeat()` to log a guitar icon `$message->length` times. I'll also log that number at the end.

```
src/MessageHandler/LogHelloHandler.php
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 9
final class LogHelloHandler
{
    public function __construct(private LoggerInterface $logger)
    {
    }

    public function __invoke(LogHello $message)
    {
        $this->logger->warning(str_repeat('🎸', $message->length).' '.$message->length);
    }
}
```

Message & handler check!

Creating the Schedule

Next up is to create a *schedule* that tells Symfony:

> *"Yo, me again. Please dispatch a `LogHello` message through messenger every 7 days."*

Or in our case, every few seconds because I don't think you want to watch this screencast for the next week!

In `src/`, I don't have to do this, but I'll create a `Scheduler` directory. And inside, a PHP class called, how about, `MainSchedule`. Make this implement `ScheduleProviderInterface`.

```
src/Scheduler/MainSchedule.php
// ... lines 1 - 2
namespace App\Scheduler;
// ... lines 4 - 6
use Symfony\Component\Scheduler\ScheduleProviderInterface;
// ... lines 8 - 9
class MainSchedule implements ScheduleProviderInterface
{
// ... lines 12 - 14
}
```

You can have multiple of these schedule providers in your system... or you can have one class that sets up *all* your recurring messages. Your call.

This class also needs an attribute called `#[AsSchedule]`. This has one optional argument: the schedule name, which, creatively, defaults to `default`. We'll see why that name is important soon. I'll use `default`.

```
src/Scheduler/MainSchedule.php
// ... lines 1 - 2
namespace App\Scheduler;

use Symfony\Component\Scheduler\Attribute\AsSchedule;
// ... line 6
use Symfony\Component\Scheduler\ScheduleProviderInterface;
// ... line 8
#[AsSchedule]
class MainSchedule implements ScheduleProviderInterface
{
// ... lines 12 - 14
}
```

Creating the Recurring Messages

Ok, go to Code -> Generate, or command+N on a Mac - to implement the one method we need: `getSchedule()`.

```
src/Scheduler/MainSchedule.php
// ... lines 1 - 2
namespace App\Scheduler;

use Symfony\Component\Scheduler\Attribute\AsSchedule;
use Symfony\Component\Scheduler\Schedule;
use Symfony\Component\Scheduler\ScheduleProviderInterface;

#[AsSchedule]
class MainSchedule implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
    }
}
```

The code in here is beautifully simple and expressive. Return a `new Schedule()`, then add things to this by calling `->add()`. Inside, for each "thing" you need to schedule, say `RecurringMessage::`. There are several ways to create these recurring messages. The easiest is `every()`, like every `7 days` or every `5 minutes`. You can also pass a `cron` syntax, or call `trigger()`. In that case, you would define your *own* logic for *exactly* when you want your weird message to be triggered.

Use `every()` and pass `4 seconds`. Every 4 seconds, we want this new `LogHello` message to be dispatched to Messenger. Copy that, then create another for every `3 seconds`.

```
src/Scheduler/MainSchedule.php
// ... lines 1 - 4
use App\Message\LogHello;
// ... line 6
use Symfony\Component\Scheduler\RecurringMessage;
// ... lines 8 - 11
class MainSchedule implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
        return (new Schedule())->add(
            RecurringMessage::every('4 seconds', new LogHello(4)),
            RecurringMessage::every('3 seconds', new LogHello(3)),
        );
    }
}
```

We're done!

Consuming the Scheduler Transport

The *result* of creating a schedule provider is that a new Messenger *transport* is created. To get your recurring messages to process, you need to have a worker that's running the `messenger:consume` command.

At your terminal, run `bin/console messenger:consume` with a `-v` so we can see the log messages from our handler. Then pass the name of the new, automatically-added transport: `scheduler_default`... where `default` is the *name* we used in the `#[AsSchedule]` attribute.

```
php bin/console messenger:consume -v scheduler_default
```

Hit it, wait about 3 seconds... there it is! Four! Then the 3 one comes up again, and four, then three. After 12 seconds, they should execute, yep, at almost the exact same moment. Technically, this one was dispatched first, and *then* that one was dispatched immediately after.

But, let me stop nerding out and back up: it's working! It's beautiful!

How does Scheduler Work?

How is it working? I wondered that same thing. When the worker command starts, it loops over every `RecurringMessage`, calculates the next runtime of each, and uses that to create a list - called the "heap" - of upcoming messages. Then it loops forever. As soon as the current time matches - or is later than - the scheduled runtime of the next message in the heap, it takes that message and dispatches it through Messenger. It *then* asks this recurring message for its *next* runtime and puts *that* inside the heap.

And this process just... continues forever.

Make your Schedule Stateful

Though there is one problem hiding in plain sight: if we restart the command, it creates the schedule from scratch. That means that it waits a fresh *new* three seconds and four seconds before it dispatches the messages.

In a real app, this will be a problem. Imagine you have a message that runs every seven days. For some reason, after 5 days, your `messenger:consume` command exits and is restarted. Because of this, your recurring message will now run seven days *after* this restart: so it will run on day 12. If it keeps getting restarted, your message may *never* run!

This is *not* workable. And so, in the real world, we always make our schedule *stateful*. And this easy. Create a `__construct` method and autowire a `private CacheInterface`: the one from Symfony cache.

src/Scheduler/MainSchedule.php

```php
// ... lines 1 - 9
use Symfony\Contracts\Cache\CacheInterface;
// ... lines 11 - 12
class MainSchedule implements ScheduleProviderInterface
{
    public function __construct(
        private CacheInterface $cache,
    )
    {

    }
// ... lines 21 - 29
}
```

Down below, call `->stateful()` and pass `$this->cache`.

```
src/Scheduler/MainSchedule.php
// ... lines 1 - 9
use Symfony\Contracts\Cache\CacheInterface;
// ... lines 11 - 12
class MainSchedule implements ScheduleProviderInterface
{
    public function __construct(
        private CacheInterface $cache,
    )
    {

    }
// ... line 21
    public function getSchedule(): Schedule
    {
        return (new Schedule())->add(
// ... lines 25 - 26
        )
            ->stateful($this->cache);
    }
}
```

Also, open `services.yaml`. In an earlier tutorial, I added some config that effectively disabled the cache in the `dev` environment. Remove that so we have a proper cache.

Ok, stop the worker and restart it. The first time we do this, it's going to have the same behavior as before: wait three seconds and four seconds. There we go.

But now, stop this, wait a few seconds and watch what happens when I restart. It catches up! Those messages happened immediately!

The state keeps track of the last time Scheduler checked for messages. And so, if your worker gets turned off for a bit, when it restarts, it *reads* that time and uses it as its starting time so it can catch up with all the messages that it missed.

It does mean that you may have some messages that are executed multiple times immediately, but it won't miss anything.

Multiple Workers: Lock your Schedule

Oh, and if you plan to have multiple workers for your scheduler transport, you'll also need to add a *lock* to the schedule. This is easy and covered in the docs: autowire the lock factory, then call

`->lock()` to pass in a new lock. This will make sure that two workers don't grab the *same* recurring message at the same time and *both* process it.

All right team, that's all I've got! Thanks for hanging out. If you have any questions about upgrading or hit a problem we didn't mention, we're here for you down in the comments. And let us know if you have a victory: we love hearing success.

All right, friends. See you next time!

With <3 from SymfonyCasts