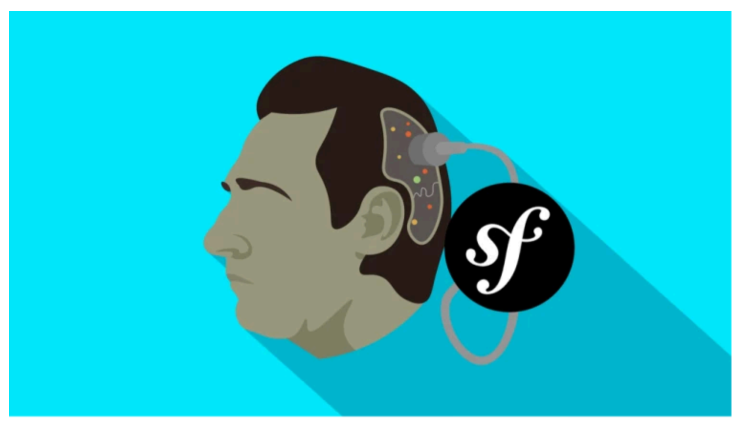
Upgrading to Symfony 8



Chapter 1: PHP 8.4 & Recipe Updates

Hey friends! Welcome to a new course on upgrading a Symfony app from Symfony 7 to Symfony 8. Luckily, newer versions of Symfony make this upgrade process a breeze. But there are still some things to make the upgrade process smoother.

To follow along, download the course code from the top of this page, open the `start` directory in your IDE, and you'll see what I have here. Follow the README to get everything set up, when you're ready, in your terminal, run:

```
symfony server:start -d
```

(the `-d` flag runs the server in the background). Click the link in the output to open the app in your browser.

Welcome to the Starshop! If you've followed along with our previous Symfony 7 courses, this should look familiar.

Down in the web debug toolbar, you can see we're on Symfony 7.3.5 and PHP 8.3.30.

Our goal? Update this bad boy to Symfony 8!

Since Symfony 8 requires PHP 8.4, a good first step is to upgrade this app to require PHP 8.4. At the same time, we'll also upgrade to the latest version of Symfony 7.3.

If you've done these major upgrades before, you might be thinking we should upgrade to 7.4, the last Symfony 7 version. We will, but I like to take small steps to avoid getting overwhelmed with too many changes at once.

In your IDE, open `composer.json`. In the `require` section, change `>=8.3` to `>=8.4`:

```
composer.json
1  {
↕  // ... lines 2 - 5
6      "require": {
7          "php": ">=8.4",
↕  // ... lines 8 - 40
41     },
↕  // ... lines 42 - 103
104 }
```

This tells composer at least PHP 8.4 is required for this project.

Scroll down and find the `config.platform.php` section. This is what the Symfony CLI uses to determine which local version of PHP to use when you run `symfony console` or `symfony php`. Change this to `8.4`:

```
composer.json
1  {
↕  // ... lines 2 - 41
42     "config": {
43         "platform": {
44             "php": "8.4"
45         },
↕  // ... lines 46 - 51
52     },
↕  // ... lines 53 - 103
104 }
```

This step isn't required, but scroll down to the `replace` section. This is a list of packages that composer will ignore when installing dependencies. You can see it's all *polyfill* packages. These are packages that provide features from newer versions of PHP to older versions. Since we're on PHP 8.4, we can add the polyfill packages for 8.3 and 8.4 to this list:

```
composer.json
1  {
↕  // ... lines 2 - 62
63     "replace": {
↕  // ... lines 64 - 71
72         "symfony/polyfill-php83": "*",
73         "symfony/polyfill-php84": "*"
74     },
↕  // ... lines 75 - 103
104 }
```

Saves a few bytes of disk space and a few milliseconds of install time...

Over in our terminal, let's confirm we're using PHP 8.4 now. Run:

```
symfony php --version
```

Nice, 8.4.20!

Now, let's do a composer update to capture all the changes we made in our `composer.json` file and upgrade to the latest allowed dependencies. Run:

```
symfony composer update
```

No errors, great! Jump to the browser and refresh the page...

Gross, this is a nasty error. It doesn't look like the standard Symfony errors because it couldn't even load Symfony. This is an error at the composer autoloader level.

It's saying we're running PHP 8.3 but our dependencies require PHP 8.4. Remember we ran the Symfony server in the background earlier? We ran it with PHP 8.3. Simple fix, just restart the server with:

```
symfony server:stop symfony server:start -d
```

Now refresh the page... Sweet! We're back in business. Notice in the web debug toolbar we're now on Symfony 7.3.11 and PHP 8.4.20.

Now for deprecations! When Symfony, or PHP itself wants to remove or change a feature, they don't just make the change. This could break tons of apps and cause a lot of problems. Instead, they mark the feature as *deprecated* for a while, which means it still works as expected, but it will trigger a warning. Usually, the warning explains how to fix the deprecation, and upgrade to the new way of doing things.

Symfony has a really great deprecation policy. They guarantee that nothing will break when upgrading to a new minor version, like 7.3 to 7.4. The breaking changes happen in the major version, like going from 7 to 8. In 7.4, all breaking changes that will happen in 8 are marked as deprecations. This gives you time and guidance to fix them. You only upgrade to 8 when no deprecations remain.

So how do we find deprecations! This deprecation logging panel is going to be your best friend when upgrading Symfony.

Looks like we have 3. This first one is a PHP deprecation. The `addDroid()` method in our `Starship` entity needs a parameter to be explicitly marked as nullable. Let's fix that first.

Open `src/Entity/Starship.php` and scroll down to the `addDroid()` method. Ahh, even PhpStorm is warning me about this. The fix is simple, add a `?` before `DateTimeImmutable`:

```
src/Entity/Starship.php
     // ... lines 1 - 15
16   class Starship
17   {
     // ... lines 18 - 207
208      public function addDroid(Droid $droid, ?\DateTimeImmutable $assignedAt = null): static
     // ... lines 209 - 263
264  }
```

Back to the browser, refresh the homepage, and... still 3 deprecations... It didn't get cleared. Sometimes when you fix deprecations, you need to clear the Symfony cache manually for it to be picked up. This is because some deprecations are detected at compile time when the container is being built. So in your terminal, run:

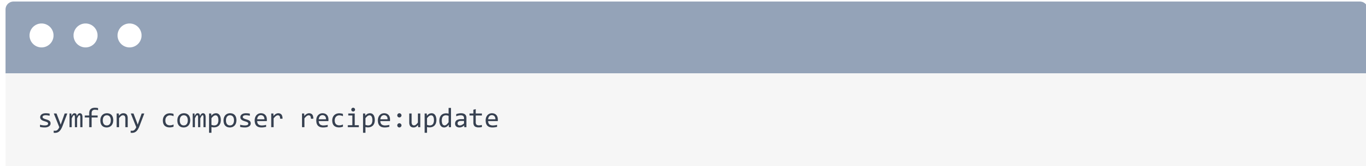
```
symfony console cache:clear
```

Refresh the homepage again... Whoa, now we have 4 deprecations. Open the panel. Our `addDroid()` deprecation is gone, so we did fix that one. Sometimes when you clear the cache manually, it triggers other deprecations. Ok, 3 of these are from Doctrine. We'll have a whole chapter on upgrading Doctrine a little later, so we'll ignore these for now.

The fourth one is a deprecated config option from zenstruck/foundry.

Before manually fixing this, let's first update our Symfony Flex recipes. Sometimes, just updating these fixes deprecations for you.

To update the recipes, over in your terminal, run:



```
symfony composer recipe:update
```

Hmm, that didn't work. It says we have uncommitted changes. The recipe update command needs to be run on a clean slate - no pending changes. Run:

```
git status
```

To see our changes. `composer.json`, `composer.lock`, and our `Starship.php`. Makes sense, let's commit these with:

```
git commit -a -m "composer update"
```

The `-a` adds all modified files to the changelist before committing and `-m` lets us set a commit message.

Clean slate, so run the recipe update command again:

```
symfony composer recipe:update
```

Ooo, we have a bunch to update. We'll go through them one-by-one. Hitting enter will use the first one in the list, `doctrine/deprecations`.

"No file changed...". Hmm, run `git status` so see what's up. Just the `symfony.lock` file was modified. This is the internal config file Symfony Flex uses to keep track of your recipes. We can just commit this and move on:

```
git commit -a -m "update recipes"
```

Onward! Run the recipe update command again:

```
symfony composer recipe:update
```

`asset-mapper` is next... Just the `symfony.lock` file again.

When I update recipes, I like to keep all the updates in a single commit. So we'll amend the new changes with the previous commit with:

```
git commit -a --amend
```

This opens a text editor to optionally update the commit message. I'll just exit to keep it the same.

Now update the `framework-bundle`'s recipe... Ooo, this one has some real changes. A cool thing about this command is it shows the CHANGELOG from the symfony/recipe repository, so you can easily dig in to understand why a change was made.

Run `git status` to see the changes. Aside from the `symfony.lock` file, `public/index.php` was modified. Let's check it out! Open `public/index.php` and see the change... Looks like the `static` keyword was added. I think this is a helpful micro-optimization, we'll keep it!

This is a good time to mention that you should never blindly update recipes. Always check the changes and if you're unsure about a change, follow that changelog back to the symfony/recipe repository to find the reasoning.

Amend the commit and move onto the next recipe update. The `monolog-bundle` also has some changes. Looks like package config, so open up `config/packages/monolog.yaml` to see them. Just some comments removed. No big deal... amend the commit.

Next, update the `stimulus-bundle` recipe and see the changes. Open `assets/stimulus_bootstrap.js`. Just added some boilerplate stuff. Amend the commit and

move onto updating the `twig-bundle` recipe. Our base template was modified, so open `templates/base.html.twig`. Some FrankenPHP stuff was added. We're not using FrankenPHP at the moment but doesn't hurt to leave it in case we do in the future!

Amend the commit and update the last recipe: `zenstruck/foundry`. The config file was modified, so open `config/packages/zenstruck_foundry.yaml`. Just a comment was updated, so no big deal.

Amend the commit and we should be done! Run the recipe update command again to confirm. Nice! "All packages appear to be up to date."

Before we check the app, clear our cache again...

Now refresh the homepage. 2 deprecations. The Doctrine autoloader one is still here, but we'll ignore that until we upgrade Doctrine. The Foundry one is still here too, so the recipe update didn't fix it. Let's fix it ourselves. It's saying this `enable_auto_refresh_with_lazy_objects` config is going to be forced to `true` in 3.0, so let's set it to `true` now. Copy the option and open `config/packages/zenstruck_foundry.yaml`. Under `zenstruck_foundry`, paste, and set to `true`:

config/packages/zenstruck_foundry.yaml

```
1  when@dev: &dev
   // ... line 2
3      zenstruck_foundry:
4          enable_auto_refresh_with_lazy_objects: true
   // ... lines 5 - 18
```

Totally an aside, but if you're curious what this weird `&dev` and `*dev` syntax is, this is a YAML anchor and alias. The `&dev` marks this config as an anchor named `dev`, and then the `*dev` is an alias that references that anchor. So this is saying, "for the `test` environment, use the same config as `dev`". It's a cool way to avoid duplication in your YAML files.

Ok, back to the browser and refresh the homepage. There are no deprecations now!

But... let's clear the cache... and refresh again... 3 now but these are the Doctrine ones we'll fix later.

Ok, we've successfully upgraded our app to PHP 8.4. Next, we'll upgrade to Symfony 7.4, the last Symfony 7 version.

Chapter 2: Upgrading to Symfony 7.4

Alright, let's dive right into upgrading to Symfony 7.4. Open your `composer.json` file. Notice how the Symfony packages use a `7.3.*` format. This is slightly different from the rest of our packages, which typically use the `^` prefix format. This makes it super easy to find and update the `symfony` packages.

If you're using PhpStorm, go to edit... find... replace. Search for `7.3.*` and replace with `7.4.*`. We can see we have 19 occurences to replace. Hit replace all... and... boom!

composer.json

```
1   {
    // ... lines 2 - 5
6       "require": {
    // ... lines 7 - 18
19          "symfony/asset": "7.4.*",
20          "symfony/asset-mapper": "7.4.*",
21          "symfony/console": "7.4.*",
22          "symfony/dotenv": "7.4.*",
    // ... line 23
24          "symfony/form": "7.4.*",
25          "symfony/framework-bundle": "7.4.*",
26          "symfony/http-client": "7.4.*",
    // ... line 27
28          "symfony/property-access": "7.4.*",
29          "symfony/property-info": "7.4.*",
30          "symfony/runtime": "7.4.*",
31          "symfony/security-csrf": "7.4.*",
32          "symfony/serializer": "7.4.*",
    // ... line 33
34          "symfony/twig-bundle": "7.4.*",
    // ... line 35
36          "symfony/validator": "7.4.*",
37          "symfony/yaml": "7.4.*",
    // ... lines 38 - 40
41      },
    // ... lines 42 - 103
104 }
```

Let's just do a double check to make sure we didn't miss any. Under `require`... yep, looks good. Now down to `require-dev`...

composer.json

```
1  {
// ... lines 2 - 96
97     "require-dev": {
98         "symfony/debug-bundle": "7.4.*",
// ... line 99
100        "symfony/stopwatch": "7.4.*",
101        "symfony/web-profiler-bundle": "7.4.*",
// ... line 102
103    }
104 }
```

Yep, looks good there too. The `maker-bundle` has a different versioning strategy than the Symfony core components and bundles. That's why it looks different.

Notice under the `extra` section, we have this `symfony` `require` config.

composer.json

```
1  {
// ... lines 2 - 90
91     "extra": {
92         "symfony": {
// ... line 93
94             "require": "7.4.*"
95         }
96     },
// ... lines 97 - 103
104 }
```

This tells Symfony Flex which version of Symfony to use when installing Symfony components. Some of our required Symfony components require other Symfony components as dependencies. These are called *transitive* dependencies (fancy word!), and can allow a wide range of Symfony versions. Like Symfony 6, 7, or 8. This config ensures that only the `7.4` versions are installed. So when upgrading Symfony, it's important to also update this config.

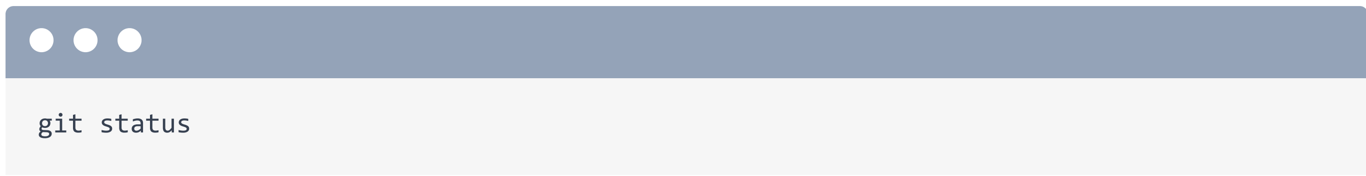
Great! Now that we've updated our `composer.json`, let's run our composer update:

```
symfony composer update
```

Verifying the Upgrade

Perfect, it looks like it worked! Let's hop over to our app and refresh. Yep, we're now on 7.4.8, the latest 7.4 version.

Back in our terminal, run:



```
git status
```

Our `composer.json` and `composer.lock` files are modified, this is expected. But we also have this new `config/reference.php`.

Open that up in your editor. This is an auto-generated file Symfony creates when building the container. Symfony now has a PHP array-based config format - an alternative to YAML. This file is generated to provide better auto-completion when using that format. YAML is still the recommended format, and what we're using in this app, so this file isn't important for us right now. If Symfony changes it's recommendation in the future, we'll be ready! Check out this blog post to learn more about it.

You can either add this file to your `.gitignore` or commit it. The best practice right now is to commit it, so let's do that. In your terminal, run:

```
git add config/reference.php
```

After running `git status` to confirm we're good, we can commit with:

```
git commit -a -m "update composer"
```

(Yeah, we should really use a better commit message here)

Updating Recipes

Let's see if there are any recipe updates for 7.4! Run:

```
symfony composer recipe:update
```

Just two, start with the `framework-bundle`. Run `git status` to see the changes. `.env` and `config/services.yaml` have been modified.

Open `.env` first. A new environment variable was added: `APP_SHARE_DIR`. When running Symfony in a multiserver architecture, this is a directory that should be shared between the servers. Previously, you had to share the entire cache directory, which isn't idea. This new config allows you to have more fined-grained control over what is shared between servers. If you're interested in learning more about this, check out this blog post.

Open our second modified file, `config/services.yaml`. Just the comments at the top were modified. But it does provide a cool new feature! See this `yaml-language-server $schema` stuff? This configures a JSON schema for this file. Wait, JSON? This is YAML. Since YAML is JSON-compatible, we can use JSON schemas to validate our YAML files. This is cool, but the better part, is that it gives us auto-completion in our IDEs if supported. And PhpStorm does support this! Here's a blog post if you want to learn more about it!

Ok, let's commit these changes at our terminal with:

```
git commit -a -m "update recipes"
```

Run the recipe update command again, and update the `routing` package. Run `git status` to see the changes. `config/routes.yaml`: open that up. At the top, a YAML JSON schema config was added.

Down below, check out the new simplified config. With the previous config, only controllers in `src/Controller` would be loaded. With this new config, any class with `#[Route]` attributes in it will be loaded as controllers. It's still the best practice to put all your controllers in `src/Controller`. But if you have a more complex app with multiple domains and their own controllers, these would be loaded no matter where they're located.

Let's commit these changes with:

```
git commit -a --amend
```

Testing the Upgrade

Perfect! Pop over to our app and refresh the homepage to make sure everything's still running smoothly. Nice, 7.4 upgrade was a success!

Next, we're going to update Doctrine and explore a cool improvement in the latest version of the ORM. Stay tuned!

Chapter 3: Upgrading Doctrine & Native Lazy Objects

We need to upgrade the doctrine-bundle... but before we do, I want to give a refresher on a really cool Doctrine feature: lazy objects.

Open up `src/Entity/StarshipPart.php`. This entity has a `Starship` property with a many-to-one relationship to our `Starship` entity. Each `StarshipPart` has one `Starship`, and each `Starship` can have many `StarshipParts`. When fetching entities with relationships, Doctrine uses some cool magic to avoid unnecessary database queries. Let's see how it works in action.

Setting Up a Temporary Controller

Over in your terminal, create a new controller:

```
symfony console make:controller
```

Name it `LazyController`, and no need for tests.

Open the new controller in `src/Controller/LazyController.php`. Ok, we have this `index()` method whose route is set to `/lazy`. Inject `StarshipPartRepository $repository` into it. Then, grab the first part from the repository with `$part = $repository->find(1)`. Dump it with `dump($part)`:

```
src/Controller/LazyController.php
    // ... lines 1 - 9
10  final class LazyController extends AbstractController
11  {
12      #[Route('/lazy', name: 'app_lazy')]
13      public function index(StarshipPartRepository $repository): Response
14      {
15          $part = $repository->find(1);
16
17          dump($part);
    // ... lines 18 - 21
22      }
23  }
```

Now, head back to our app and manually navigate to the `/lazy` url.

Exploring Doctrine's Lazy Objects

Looking at the web debug toolbar, we see a single query. That's the one fetching the `StarshipPart`. If we open the dump profiler panel, we see the fetched `StarshipPart` object. Note the `starship` property, it's type is this funky Proxy CG thing. This is a Doctrine lazy object. Look inside. All the properties are unset except for the ID. The ID is the only thing Doctrine knows about the `Starship` until it queries the database for the rest of the data. Only when accessing a property of the `Starship` does Doctrine trigger a second query to fetch the rest.

Let's trigger this second query! In `LazyController::index()`, add `$part->getStarship()->getName()` as the first argument to the `dump()`:

```
src/Controller/LazyController.php
    // ... lines 1 - 9
10  final class LazyController extends AbstractController
11  {
    // ... line 12
13      public function index(StarshipPartRepository $repository): Response
14      {
    // ... lines 15 - 16
17          dump($part->getStarship()->getName(), $part);
    // ... lines 18 - 21
22      }
23  }
```

Refresh the `/lazy` page... There are now two queries. The first one fetches the `StarshipPart`, and the second one fetches the `Starship` because we accessed its name.

In the dump panel, we can see the `Starship` is now fully loaded with all its properties.

So this CG Proxy thing is a real class Doctrine generates on the fly. It contains all the logic to fetch the data when needed. The key is, it *extends* the real `Starship` entity. That's how it can be used as a stand-in for the `Starship` until the actual data is needed. That's also why entities cannot be final, they need to be extendable by these proxy classes.

As you might imagine, the logic to do all this is pretty complex and difficult to maintain. But... that all changed in PHP 8.4, which introduced *native* lazy objects to PHP itself.

And Doctrine Bundle 3, allows our Symfony apps to take advantage! So let's upgrade!

Upgrading Doctrine Bundle

First, open our `composer.json`. Look for where we're requiring the `doctrine-bundle`. Change its version to `^3.0`.

Now, in the terminal, run:

```
symfony composer update
```

Oooo, a composer error. Our dependencies couldn't be resolved... `composer.json` requires `doctrine-bundle` 3. Yeah... `doctrine-bundle` 3 requires `doctrine/dbal` 4 but this conflicts with our requirement for `doctrine/dbal` 3.

For reference, `doctrine/dbal` is the *database abstraction layer* that Doctrine uses to communicate with different databases.

Let's check our `composer.json`. We're requiring just `dbal` version 3, but the `doctrine-bundle` needs version 4 according to that error.

Ok, this is a bit of a legacy issue. Previous versions of `doctrine-bundle` didn't support `dbal` 4, so we had to ensure version 3 was used. This is no longer required, so we can just remove it entirely, and let `doctrine-bundle` decide which version to use.

Try the update again...

Another error, but a different one. Scroll up a bit... we can see `doctrine-bundle` 3 and `dbal` 4 were successfully installed.

The error was caused when attempting to clear the cache. Looks like our `doctrine` configuration is using some options that are no longer supported.

We could fix these manually, but I'm pretty sure upgrading the Flex recipe will resolve this.

We need a clean git state before we upgrade the recipe, so let's check our `git status`. We have some modified changes and some untracked files. Run:

```
git add .
```

To track everything and run `git status` again. Now that everything is tracked, we can commit it with:

```
git commit -a -m "update doctrine-bundle"
```

`git status` again to confirm we're clean. Finally, upgrade the recipe with:

```
symfony composer recipe:update
```

Sure enough, it found an update for `doctrine-bundle`. Apply it!

Run a `git status` to see the changes. Perfect, it updated the `doctrine.yaml` file.

Checking the Changes

Back in our code, open up `config/packages/doctrine.yaml`. First of all, it removed 3 config options. These were the ones that caused that error when clearing the cache.

Next, it looks like it changed the default naming strategy... and removed some controller resolver config.

Down under the production config, it removed the `auto_generate_proxy_classes` and `proxy_dir` options. With the original lazy object system, in production, Doctrine generated files for the proxy classes to improve performance.

None of that is needed anymore with *native* lazy objects!

Ok, let's see what our lazy objects look like now. First, head back to `LazyController::index()` and remove the first argument from the `dump()`:

src/Controller/LazyController.php

```
// ... lines 1 - 9
10 final class LazyController extends AbstractController
11 {
// ... line 12
13     public function index(StarshipPartRepository $repository): Response
14     {
// ... lines 15 - 16
17         dump($part);
// ... lines 18 - 21
22     }
23 }
```

This should be dumping the part with a starship instance that isn't fully loaded.

Refresh the `/lazy` page in your browser. Ok, just one query, which is expected. Open the debug panel and check the `starship` property. This is now our normal Starship entity - not that generated proxy class.

But look inside! All the properties are unset except for the ID. This looks just like we saw in the old proxy class. This is *native* lazy objects in action!

Back in `LazyController::index()`, re-add the `$part->getStarship()->getName()` to the `dump()`...

```
src/Controller/LazyController.php
// ... lines 1 - 9
10 final class LazyController extends AbstractController
11 {
// ... line 12
13     public function index(StarshipPartRepository $repository): Response
14     {
// ... lines 15 - 16
17         dump($part->getStarship()->getName(), $part);
// ... lines 18 - 21
22     }
23 }
```

and refresh the `/lazy` page again. Two queries, and if we look at the `starship` property in the dump panel. Still just our normal `Starship` entity... and if we expand it... all its properties are loaded!

Finalizing Entities

Ok... this is cool, but what does it really mean for me as a developer? Well, there probably is some performance improvement with *native* lazy objects.

But the primary takeaway is we can now, finally, mark our entities as `final`. So, yeah, not super life-changing, but it's nice we don't need a hacky workaround anymore.

So... let's mark our entities as final!

`src/Entity/Droid.php`: final:

```
src/Entity/Droid.php
// ... lines 1 - 10
11 final class Droid
// ... lines 12 - 121
```

`Starship.php`: final:

```
src/Entity/StarshipPart.php
// ... lines 1 - 11
12 final class StarshipPart
// ... lines 13 - 90
```

`StarshipDroid.php`: final:

```
src/Entity/StarshipDroid.php
// ... lines 1 - 8
9 final class StarshipDroid
// ... lines 10 - 73
```

and *finally* `StarshipPart.php`: final:

```
src/Entity/StarshipPart.php
// ... lines 1 - 11
12 final class StarshipPart
// ... lines 13 - 90
```

Refresh our lazy page... and... it all still works!

We're almost ready to make the push to Symfony 8, but before we do, let's revisit deprecations.

Chapter 4: Tracking & Fixing Deprecations

Let's revisit deprecations as these are important to fix. As mentioned earlier, before you can safely jump to Symfony 8.0, you need to ensure you're running deprecation-free Symfony 7.4. This means that there should be zero deprecations across your entire app.

We know we can find deprecations in the Web Debug Toolbar, but... this just shows the deprecations for the current page... None here... but click on this starship. This page does have a deprecation.

Digging into the Deprecations Panel

Let's explore the deprecation panel in more detail. Click the icon in the toolbar to open it. This deprecation looks like it's coming from Doctrine, something about raw field value access... But look at this! A link to a pull request right in the message! Very helpful, I love it! Copy the link and paste it in your browser...

This is the Doctrine PR that modified some behavior and triggered this deprecation. It looks like it fixes some memory issues. There's a huge discussion to help give us more context if we want. But how do we fix it? In the PR description, there is a migration path section. The new behavior is opt-in, and can be enabled by passing `accessRawFieldValues: true` to the `Criteria` constructor.

Ok... how do we find where our app needs this fix? Deprecation panel to the rescue!

Click "show trace" under the deprecation message. This is the stack trace of how your app got to this deprecation. You read this from the top down and can mostly ignore the lines that aren't in your `src` directory. The first line in our `src` directory is likely what's triggering the deprecation. In this case, it's `StarshipPartRepository` line 23. We can even expand this to see the actual code that's triggering the deprecation. Neat!

Also in this panel is this "show context" link. This shows the actual exception that was thrown, but this mostly shows the same information. The message... and the stack trace. I usually only ever use the stack trace link.

So... to find all the deprecations with the web debug toolbar, we'd have to locally go to every page, every form submit, every API endpoint, and so on. That's not very efficient. So, what's the better way to find deprecations?

There's two primary alternative ways. First is your test suite. PHPUnit can be configured to show deprecations. But... unless you have 100% test coverage, this won't catch everything. Oh man, I don't even have tests in this app...

Logging Deprecations

The final way is a surefire catch-all method. You deploy your Symfony 7.4 app to production for a while, a few weeks or a month. During that time, you log all the deprecations that are happening in production. Remember, deprecations aren't errors, so they won't break your app. Either as they happen, or at the end of the time period, you review those logs and fix the deprecations. Do this cycle a few times until you have no more deprecations in production. Now you're ready to safely upgrade to Symfony 8.0.

Let's take a look at how logging works and can be improved. Open `config/packages/monolog.yaml`. Scroll down to the `deprecation` handler under the production config. This handler writes just the deprecations to the standard error stream. Of course, you can customize this to your liking. Spamming my team's Slack channel with them is my favorite!

Let's add this handler to our dev environment so we can see it in action. Copy the `deprecation` handler and paste it in the `when@dev` section... I don't want to stream these to standard error here, so copy the path from the handler above and paste it here. Suffix the file with `deprecations.json`:

config/packages/monolog.yaml

```
// ... lines 1 - 4
5  when@dev:
6      monolog:
7          handlers:
// ... lines 8 - 16
17             deprecation:
18                 type: stream
19                 channels: [ deprecation ]
20                 path: "%kernel.logs_dir%/%kernel.environment%.deprecations.json"
21                 formatter: monolog.formatter.json
// ... lines 22 - 61
```

This is already using the JSON formatter and using that extension will help me make it pretty in PhpStorm.

Back in our app, refresh the page that causes the deprecation. Now, open `var/log`... Hmm, I don't see the file. Maybe PhpStorm hasn't picked it up yet. I'll reload from disk... and there it is!

`dev.deprecations.json`. Open that up. I'll format the code to make it easier to read.

Including Stack Traces in the Logs

We see the same deprecation message as we saw in the profiler panel... and a bunch of other things... but... it's missing the stack trace. By default, it's hard to know where the deprecation is being triggered.

Luckily, we can include the stack trace! I'll clear this log file so we have a fresh start.

Back in `monolog.yaml`, in our dev deprecation handler config, add `include_stacktraces: true`:

```
config/packages/monolog.yaml
// ... lines 1 - 4
5  when@dev:
6      monolog:
7          handlers:
// ... lines 8 - 16
17             deprecation:
// ... lines 18 - 21
22                 include_stacktraces: true
// ... lines 23 - 62
```

Refresh the page... check the log file... format it... and there we go! The stack trace looks just like it did in the profiler panel. And sure enough, we can see that `StarshipPartRepository` line 23 is triggering the deprecation.

Log Processors

There's one thing I think that's still missing... It would be nice to know what URL triggered it. This would help duplicating the issue locally and confirming the fix. Monolog has something called *processors* that can add extra data to the log entries. There's a bunch of built-in processors, one

to add authentication details, console command details, and additional debug information. The one I want to enable is the `WebProcessor`. This adds the request details, like the URL, HTTP method, and IP.

The built-in processors aren't registered as services by default, but it's super easy to do it ourselves.

I'll clear this log file again.

Open `config/services.yaml` and at the bottom, under `services`, add `monolog.processor.web` with the class `WebProcessor`. Choose the one from the Monolog Bridge, as it's ready to go with Symfony:

```
config/services.yaml
   // ... lines 1 - 10
11 services:
   // ... lines 12 - 24
25     monolog.processor.web:
26         class: Symfony\Bridge\Monolog\Processor\WebProcessor
```

That's it! It's autowired and autoconfigured, so we don't need to do anything else.

Refresh the page again... check the log file... format it... and there we go! We have the message, the stack trace, and down at the bottom, under `extra`, we have the URL, the IP, and the HTTP method.

Just modify our setup for production in your apps and deploy. Finding, duplicating, and fixing deprecations will be a breeze!

Fixing the Deprecation

Ok, now to actually fix this deprecation! It was triggered in `StarshipPartRepository` on line 23, right? Open that up... and find line 23... Remember, we have to pass `true` to the `Criteria` constructor. This `Criteria::create()` isn't the real constructor, but it's a *named constructor*. When we jump to that method... sure enough, it has the `accessRawFieldValues` parameter... but it's commented out. What's up with that?! It's part of the deprecation process. They can't just add the new parameter. This class and method aren't final, so to maintain backwards compatibility, they have to keep the current signature. They comment out the new parameter

and trigger the deprecation if it's not passed. This `func_num_args()` stuff is a way to check if the new parameter was passed without actually adding it to the method signature.

The next major version of this package will have the real parameter in the method signature, and the old way will be removed. That's the last step of the deprecation cycle.

Enough talk, let's fix it! Back in `StarshipPartRepository`, pass `true` to `Criteria::create()`:

src/Repository/StarshipPartRepository.php

```
// ... lines 1 - 13
14 class StarshipPartRepository extends ServiceEntityRepository
15 {
// ... lines 16 - 20
21     public static function createExpensiveCriteria(): Criteria
22     {
23         return Criteria::create(true)->andWhere(Criteria::expr()->gt('price',
   50000));
24     }
// ... lines 25 - 56
57 }
```

Back to the browser. When we refresh, this deprecation should be gone... And it is!

We're finally ready to move to Symfony 8! That's next!

Chapter 5: Upgrading to Symfony 8.0!

Symfony 7.4? Check. Deprecation-free? Check.

Nice, it's finally time to upgrade to Symfony 8! Thankfully, we've done most of the hard work already, so the upgrade process should be straightforward. Let's get started by opening our `composer.json` file.

We'll do the same thing we did when upgrading to 7.4. Edit... Find... Replace... `7.4.*` and replace it with `8.0.*`. Replace all... Nice.

I'll take a moment to verify. `require` section looks good... Ah, great, our `extra` `symfony` `require` option was updated. And `require-dev` looks good too!

```
composer.json
1  {
   // ... lines 2 - 5
6      "require": {
   // ... lines 7 - 17
18         "symfony/asset": "8.0.*",
19         "symfony/asset-mapper": "8.0.*",
20         "symfony/console": "8.0.*",
21         "symfony/dotenv": "8.0.*",
   // ... line 22
23         "symfony/form": "8.0.*",
24         "symfony/framework-bundle": "8.0.*",
25         "symfony/http-client": "8.0.*",
   // ... line 26
27         "symfony/property-access": "8.0.*",
28         "symfony/property-info": "8.0.*",
29         "symfony/runtime": "8.0.*",
30         "symfony/security-csrf": "8.0.*",
31         "symfony/serializer": "8.0.*",
   // ... line 32
33         "symfony/twig-bundle": "8.0.*",
   // ... line 34
35         "symfony/validator": "8.0.*",
36         "symfony/yaml": "8.0.*",
   // ... lines 37 - 39
40     },
   // ... lines 41 - 89
90     "extra": {
91         "symfony": {
   // ... line 92
93             "require": "8.0.*"
94         }
95     },
96     "require-dev": {
97         "symfony/debug-bundle": "8.0.*",
   // ... line 98
99         "symfony/stopwatch": "8.0.*",
100        "symfony/web-profiler-bundle": "8.0.*",
   // ... line 101
102    }
103 }
```

Head over to the terminal and run:

```
symfony composer update
```

Resolving Errors

Ooo, an error, let's scroll up a bit. Ok, our root composer.json requires `symfonycasts/tailwind-bundle` `^0.9.0`. This bundle requires some Symfony packages, but unfortunately, Symfony 8 versions aren't supported. Let's see if there's an updated version.

Over in the browser, go to packagist.org. Search for `tailwind-bundle`... Here it is. On the right here, are all the versions, select `v0.9.0`, the version we're on. Sure enough, Symfony packages it requires only support up to Symfony 7. Here's the newest version: `v0.12.0`, and sweet, this one does support Symfony 8!

Back in our `composer.json`... find the `tailwind-bundle`... and change its version to `^0.12.0`:

```
composer.json
1   {
↕   // ... lines 2 - 5
6       "require": {
↕   // ... lines 7 - 36
37          "symfonycasts/tailwind-bundle": "^0.12.0",
↕   // ... lines 38 - 39
40      },
↕   // ... lines 41 - 102
103 }
```

Let's try the update again.

```
symfony composer update
```

Resolving More Errors

Huh, another error. Let's see what's going on. Our `composer.json` requires `monolog-bundle ^3.0`, but it seems `monolog-bundle 3` doesn't support Symfony 8. Let's

quickly jump back to Packagist and search for `monolog-bundle`.

Yep, version 3 doesn't support Symfony 8... but version 4 does!

Back to `composer.json`, find the `monolog-bundle`... and change its version to `^4.0`:

composer.json

```
1   {
    // ... lines 2 - 5
6       "require": {
    // ... lines 7 - 25
26          "symfony/monolog-bundle": "^4.0",
    // ... lines 27 - 39
40      },
    // ... lines 41 - 102
103 }
```

Note

If you're wondering why the `monolog-bundle` uses a different versioning strategy than the rest of Symfony, it's because it has its own repository and release cycle. It isn't part of the core Symfony monorepo.

Will the third update be the charm?

```
symfony composer update
```

Perfect! It worked this time! Let's confirm by running:

```
symfony console --version
```

Excellent! Symfony 8.0.10!

Checking for Outdated Dependencies

Now let me show you a handy composer command that checks if you have any outdated dependencies. That's dependencies that have newer versions available.

```
symfony composer outdated -D
```

The `-D` option stands for *direct* dependencies, which means, only the packages that are in our app's `composer.json` file. Not *transitive* dependencies, which are the dependencies of our dependencies. I only really care about the direct ones.

Looks like we have 3 packages that have newer versions available. `phpdocumentor/reflection-docblock`, `symfony/stimulus-bundle` and `symfony/ux-turbo` can all be upgraded to the next major version. Let's go ahead and do that in our `composer.json` file.

Find `phpdocumentor/reflection-docblock` and change to `^6.0`. Next, `stimulus-bundle`, change to `^3.0`. Finally, `ux-turbo`, change to `^3.0`:

```
composer.json
1   {
    // ... lines 2 - 5
6       "require": {
    // ... lines 7 - 14
15          "phpdocumentor/reflection-docblock": "^6.0",
    // ... lines 16 - 31
32          "symfony/stimulus-bundle": "^3.0",
    // ... line 33
34          "symfony/ux-turbo": "^3.0",
    // ... lines 35 - 39
40      },
    // ... lines 41 - 102
103 }
```

Let's update!

```
symfony composer update
```

Excellent! No errors means there are no conflicts with the upgraded packages, so we're good to go. Running `git status` confirms that it's only our `composer.lock`, `composer.json`, and

`reference.php` files that have changed. Remember, that `reference.php` file helps your IDE with array-based Symfony configuration and is auto-generated. We can ignore it as we use YAML-based config.

Commit our changes with:

```
git commit -a -m "upgrade to Symfony 8!"
```

Hmm, the terminal didn't like my enthusiasm with the exclamation point... I'll cancel this command and try again without it:

```
git commit -a -m "upgrade to Symfony 8"
```

There we go!

Checking for Recipe Updates

Now that we're on a clean slate, check for recipe updates with:

```
symfony composer recipe:update
```

Nope, no new recipes!

Verifying the Upgrade

Head back to the browser and refresh the homepage. Sweet! We're on Symfony 8 with no errors!

Since we upgraded `ux-turbo`, I'll just confirm that it's working by clicking a link... Turbo requests happen via AJAX, and sure enough, our web debug toolbar shows an AJAX request was made.

We also upgraded `stimulus-bundle`, so let's check that it's working correctly. Open the developer tools and check the console tab... These logs show that Stimulus and our Stimulus controllers are being initialized correctly!

That's it folks! I hope you learned something valuable from this tutorial and manage to upgrade *your* apps to Symfony 8 smoothly.

Til next time, Happy coding!

Chapter 6: Composer Audit and Security Updates

One last thing! A bonus topic: security updates. This isn't necassarily related to upgrading to Symfony 8, but it's an important topic, now more than ever.

Recently, the Symfony ecosystem underwent a large security review that uncovered many vulnerabilities across various packages. The reason? AI has changed the game.

Modern AI-assisted auditing tools can help security researchers analyze large codebases and identify potential vulnerabilities much faster than before. That's leading to more discoveries, more disclosures, and ultimately more fixes.

And that's a good thing.

Finding these vulnerabilities doesn't mean Symfony is becoming less secure. If anything, it's the opposite. Better tooling means issues are being identified and fixed faster.

As developers, that means it's more important than ever to stay on top of dependency updates and pay attention when security advisories are published.

Thankfully, Composer has a built-in tool that makes this super easy.

Composer Audit

To see it in action, at your terminal, run:

```
symfony composer audit
```

This checks the packages currently installed in our project against known security advisories.

And... looks like I have a few! This output is pretty gross on this small screen, so I'll run the command again with `--format=plain`.

For each vulnerability, Composer shows the affected package, a severity level, and links where we can learn more about the issue.

Most of the time, the solution is simple: update your dependencies.

Composer will often upgrade the vulnerable package to a patched version and the problem disappears.

But not always.

Sometimes the fix exists only in a newer major version that you're not ready to upgrade to yet. Sometimes the update introduces changes that break your application.

If you fall into either of these cases, then it's important to understand the vulnerability and the risk it poses to your application.

Following the Advisory

Let's start with the last vulnerability in my list.

Each advisory includes a URL with more information. If we open this one, we're redirected to a blog post on the Symfony website with all the details.

This is usually the best place to start our investigation.

The post explains:

- what the vulnerability is;
- which versions are affected;
- under what circumstances it can be exploited;
- and which versions contain the fix.

Not all packages will use a blog post to share the details of the vulnerability. Back in our terminal, we can follow the *Advisory ID* link, and it takes us to the advisory on packagist.org. This shows us the package-related details and, if it was created via GitHub, the link to the GitHub advisory.

That's this GHSA link here. If you follow that, we can see a standardized vulnerability page. Here we have the package name, the affected versions, the patched versions, the severity, and

a description of the issue and how the fix resolves it.

To see all the published security advisories for a GitHub package, from its repository page, click "Security and Quality". Here is the full listing. Clicking one goes to its advisory page.

What is a CVE?

You've probably noticed each vulnerability has a CVE identifier. What's that?

CVE stands for **Common Vulnerabilities and Exposures**.

It's a public database of security vulnerabilities where each issue is assigned a unique identifier. The goal is simple: give everyone a common way to refer to the same vulnerability.

And this isn't specific to PHP, Composer, or even open-source software. CVEs are used across the entire software industry. Operating systems, web browsers, databases, cloud platforms, hardware devices, and open- source libraries can all have CVEs assigned to them.

For example, if someone mentions `CVE-2026-46634`, developers, security researchers, package maintainers, and security tools all know they're talking about the exact same issue.

CVEs are assigned by organizations called **CVE Numbering Authorities**, or *CNAs*.

GitHub is a CNA, which is great news for open-source maintainers because they can create security advisories and request CVE identifiers directly through GitHub's tooling.

You can view the official CVE records on cve.org.

For vulnerabilities created via GitHub, I find their interface a bit easier to navigate than cve.org, but it's good to know both places.

Ignoring a Vulnerability

If you cannot upgrade to a patched version right away, and you understand the vulnerability and the risk it poses, you have the option to ignore it with Composer.

Copy the *Advisory ID* (the CVE identifier works too) and open your `composer.json` file.

Under the `config` key, add an `audit` section with an `ignore` key. Use the copied ID as the key... and for the value, provide a reason for ignoring. This is important for future you and your teammates to understand why you accepted the risk. I also like to include *when* we can remove it.

composer.json

```
1   {
    // ... lines 2 - 40
41      "config": {
    // ... lines 42 - 50
51          "audit": {
52              "ignore": {
53                  "PKSA-21g2-dzjv-sky5": "Risk acceptable, remove when upgrading to Twig 4"
54              }
55          }
56      },
    // ... lines 57 - 107
108 }
```

Now if I run:

```
symfony composer audit
```

The vulnerability hasn't disappeared. Instead, it's been moved into an **Ignored Advisories** section - waaaay up top. This shows the reason as well.

It's important that we don't hide it completely. It should simply record that we've reviewed it, understand the risk, and are temporarily choosing not to address it.

Of course, ignoring a vulnerability should be the exception, not the rule. Whenever possible, upgrading to a patched version is the safest option.

Let's remove this ignore configuration and fix the issue properly.

Applying the Fixes

Update our dependencies with:

```
symfony composer update
```

> *“No security vulnerability advisories found.”*

Sweet, that seemed to have done it. Double check by running our the audit command again.

```
symfony composer audit
```

Beautiful! No more vulnerabilities.

Automating Security Audits

Finally, don't rely on remembering to run audits manually. Automate them.

If you're using GitHub to host your code, a great solution is a scheduled GitHub Action that runs `composer audit` on a regular basis. Running that command will cause the action to fail if vulnerabilities are found, which can alert you and your team to the issue.

I'll show you the one I use! Create a new file: `.github/workflows/composer-audit.yaml`. I'll paste the definition, but you can find it in the script below:

```
.github/workflows/composer-audit.yaml
```

```yaml
name: Composer Security Audit

on:
    schedule:
        - cron: '0 12 * * 1' # Every Monday at 12:00 PM (Noon) UTC

permissions:
    contents: read

jobs:
    composer-audit:
        runs-on: ubuntu-latest

        steps:
            - name: Checkout code
              uses: actions/checkout@v6

            - name: Setup Composer
              uses: shivammathur/setup-php@v2
              with:
                  tools: composer

            - name: Run Composer audit
              run: composer audit --locked

#       - name: Notify Slack on failure
#         if: failure()
#         uses: slackapi/slack-github-action@v3.0.3
#         with:
#           method: chat.postMessage
#           token: ${{ secrets.SLACK_TOKEN }}
#           payload: |
#             channel: <CHANNEL_ID>
#             username: "Composer Audit Failed"
#             icon_emoji: ":rotating_light:"
#             text: |
#                 Repo: ${{ github.repository }}
#                 Branch: ${{ github.ref_name }}
#
#                 <${{ github.server_url }}/${{ github.repository }}/actions/runs/${{ github.run_id }}|View details>
```

Let's go through this workflow. First, it's triggered on a schedule - every Monday at noon UTC.

Note

> Note that scheduled jobs run on your *default* branch.

Ok, onto the actual job. First, we checkout the code. Then we setup PHP and Composer using a popular action. Finally, we run `composer audit`. Notice this `--locked` flag. This tells Composer to check the installed versions in `composer.lock` against the advisories. This prevents us from having to run `composer install`, saving some action minutes. It also means we don't have to specify the PHP version in the step above, keeping it simpler.

This job will fail if any non-ignored vulnerabilities are found.

If you and your team use Slack, this commented out section is an example of how you can post an alert there on `failure()`.

> **Tip**
>
> One final tip: when possible, create dedicated pull requests for security updates. Keeping them separate from other work makes reviews easier and helps ensure security fixes get merged quickly.

And there you have it.

Upgrading your application is only part of the story. Ongoing maintenance means keeping an eye on your dependencies, understanding security advisories, and applying updates when needed.

Fortunately, Composer gives us excellent tools to make that process much easier.

> **Go Deeper!**
>
> For more information, check out our blog post on Composer security advisories.

Until next time, Happy, vulnerable-free, coding!

With <3 from SymfonyCasts