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

```
{
// ... lines 2 - 5
    "require": {
        "php": ">=8.4",
// ... lines 8 - 40
    },
// ... lines 42 - 103
}
```

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

```
{
// ... lines 2 - 41
    "config": {
        "platform": {
            "php": "8.4"
        },
// ... lines 46 - 51
    },
// ... lines 53 - 103
}
```

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

```
{
// ... lines 2 - 62
    "replace": {
// ... lines 64 - 71
        "symfony/polyfill-php83": "*",
        "symfony/polyfill-php84": "*"
    },
// ... lines 75 - 103
}
```

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
final class LazyController extends AbstractController
{
    #[Route('/lazy', name: 'app_lazy')]
    public function index(StarshipPartRepository $repository): Response
    {
        $part = $repository->find(1);

        dump($part);
// ... lines 18 - 21
    }
}
```

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
final class LazyController extends AbstractController
{
// ... line 12
    public function index(StarshipPartRepository $repository): Response
    {
// ... lines 15 - 16
        dump($part->getStarship()->getName(), $part);
// ... lines 18 - 21
    }
}
```

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
final class LazyController extends AbstractController
{
// ... line 12
    public function index(StarshipPartRepository $repository): Response
    {
// ... lines 15 - 16
        dump($part->getStarship()->getName(), $part);
// ... lines 18 - 21
    }
}
```

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
final class Droid
// ... lines 12 - 121
```

`Starship.php`: final:

src/Entity/StarshipPart.php

```
// ... lines 1 - 11
final class StarshipPart
// ... lines 13 - 90
```

`StarshipDroid.php`: final:

src/Entity/StarshipDroid.php

```
// ... lines 1 - 8
final class StarshipDroid
// ... lines 10 - 73
```

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

src/Entity/StarshipPart.php

```
// ... lines 1 - 11
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

Coming soon...

Chapter 5: Upgrading to Symfony 8.0!

Coming soon...

With <3 from SymfonyCasts