

Upgrading & What's New in Symfony 5!



With <3 from SymfonyCasts

Chapter 1: Flex, Versioning & extra.symfony.require

Hi friends! Today we get to explore the, strange, mysterious, shiny world of Symfony 5. Duh, duh, duh!

Well... first we'll cover how to *upgrade* to Symfony 5 - which is its own fancy process - and *then* we'll chat about some of my favorite features.

[Symfony 5: So What's New?](#)

As far as upgrading goes, Symfony 5 doesn't make any *huge* changes: there isn't a totally new directory structure or some earth-shattering new paradigm like Symfony Flex and the recipe system. That's really because Symfony is in a great place right now.

So if you were looking forward to countless hours of work and *huge* changes to your app in order to upgrade it. Well... you're going to be disappointed. But if you're hoping for a smooth update process - and to learn about what's changed since Symfony 4.0 was released - welcome! After we're done, we should have some time left over to go eat cake.

[The Release Cycle](#)

But the fact that nothing *crazy* changed does *not* mean that nothing has been happening. Really... phew! The last 2 years since Symfony 4 was released have been *huge*... with the introduction of the Messenger component, Mailer and many, *many* other things.

Symfony releases a new "minor" version every 6 months months: 4.0 in November 2017, 4.1 in May 2018, 4.2 in November 2018, 4.3 in May 2019 and 4.4 in November 2019. It's the most *boring* release cycle ever... and I *love* it! symfony.com even [has a roadmap page](#) where you can check the timing of any past or future versions. Each of these minor versions comes packed with new features.

Will there be a Symfony 4.5? Nope! The .4 - like 3.4 or 4.4 - is always the last one. In fact, on the *same* day that a .4 minor is released, Symfony *also* releases the next *major* version. Yep, Symfony 4.4 and 5.0 were released on the same day. The *reason* deals with how upgrades work in Symfony. But more on that later.

[Project Setup](#)

So let's get to work! To composer update your upgrading skills - sorry... I couldn't help it - you should *definitely* download the course code from this page and code along with me. When you unzip the file, you'll find a start/ directory that holds the same code you see here.

This is a Symfony 4.3 project... but the app *originally* started on 4.0. So it has a decent amount of old... "stuff" that we'll need to upgrade. Open the README.md file for all the setup instructions. The last step will be to find a terminal, move into the project and use the [Symfony binary](#) to start a local web server:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top left corner. The terminal prompt is a dollar sign followed by the command 'symfony serve'.

That starts the server at <https://127.0.0.1:8000>. Find your browser and go there. Say hello to an application that will be *very* familiar to many of you: The SpaceBar: an alien news site that we've been working on since Symfony 4 was released two years ago.

[What Does Upgrading Mean?](#)

So, I have... kind of a silly question: what does it *mean* to upgrade Symfony? Because Symfony isn't just one big library: it's a *huge* number of smaller components.

Go to the project and open composer.json. Our app has grown pretty big: it has a lot of dependencies... and about half of these start with symfony/:

```

104 lines | composer.json
1  {
    ... lines 2 - 3
4  "require": {
    ... lines 5 - 20
21  "symfony/asset": "^4.0",
22  "symfony/console": "^4.0",
23  "symfony/flex": "^1.0",
24  "symfony/form": "^4.0",
25  "symfony/framework-bundle": "^4.0",
26  "symfony/mailer": "4.3.*",
27  "symfony/messenger": "4.3.*",
28  "symfony/orm-pack": "^1.0",
29  "symfony/security-bundle": "^4.0",
30  "symfony/sendgrid-mailer": "4.3.*",
31  "symfony/serializer-pack": "^1.0",
32  "symfony/twig-bundle": "^4.0",
33  "symfony/twig-pack": "^1.0",
34  "symfony/validator": "^4.0",
35  "symfony/web-server-bundle": "^4.0",
36  "symfony/webpack-encore-bundle": "^1.4",
37  "symfony/yaml": "^4.0",
    ... lines 38 - 40
41  },
42  "require-dev": {
    ... lines 43 - 45
46  "symfony/browser-kit": "4.3.*",
47  "symfony/debug-bundle": "^3.3|^4.0",
48  "symfony/dotenv": "^4.0",
49  "symfony/maker-bundle": "^1.0",
50  "symfony/monolog-bundle": "^3.0",
51  "symfony/phpunit-bridge": "^3.3|^4.0",
52  "symfony/profiler-pack": "^1.0",
53  "symfony/var-dumper": "^3.3|^4.0"
54  },
    ... lines 55 - 102
103 }

```

When we talk about upgrading Symfony, we're *really* talking about upgrading all of the libraries that start with symfony/. Well, not *all* of the libraries: a few packages - like symfony/webpack-encore-bundle - are *not* part of the main Symfony code and follow their own versioning strategy:

```

104 lines | composer.json
1  {
    ... lines 2 - 3
4  "require": {
    ... lines 5 - 35
36  "symfony/webpack-encore-bundle": "^1.4",
    ... lines 37 - 40
41  },
    ... lines 42 - 102
103 }

```

You can upgrade those whenever you want.

But the vast majority of the symfony/ packages are part of the main Symfony library and we *usually* upgrade them all at the

same time. You don't *have* to, but it keeps life simpler.

[Removing symfony/lts](#)

Before we begin, if you started your project on Symfony 4.0, then inside of your composer.json file, you *might* have a package called symfony/lts. If you do, remove it with `composer remove symfony/lts`. I already removed it from this app.

symfony/lts was a, sort of "fake" package that helped you keep all of your many symfony packages at the same version. But this package was deprecated in favor of something different.

[extra.symfony.require](#)

Look inside your composer.json file for a key called extra and make sure it has a symfony key below it and another called require:

```
104 lines | composer.json
1  {
  ... lines 2 - 95
96  "extra": {
97    "symfony": {
98      "id": "01C1TW989CK77ZA7B2H4HC9WAG",
99      "allow-contrib": true,
100     "require": "4.3.*"
101   }
102 }
103 }
```

This is a special piece of config that's used by Symfony Flex. Remember, Flex is the Composer plugin that give us the recipe system and a few other goodies. Flex reads extra.symfony.require and does two things. First, behind the scenes, it tells Composer that all the symfony/ repositories should be locked at version 4.3.*.

Scroll back up to the require section. See how symfony/form is set to ^4.0?

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4   "require": {
  ... lines 5 - 23
24   "symfony/form": "^4.0",
  ... lines 25 - 40
41  },
  ... lines 42 - 102
103 }
```

In Composer land, that format *effectively* means 4.*. If we ran `composer update`, it would upgrade it to the latest "4" version. So, 4.4.

But thanks to Symfony Flex and the extra.symfony.require 4.3.* config, symfony/form would *actually* only be updated to the latest 4.3 version.

The *second* thing this config does - and this is the *true* reason it exists - is optimize performance. When you run `composer update` or `composer require`, Flex filters out all versions of symfony/ packages that don't match 4.3.*. That actually makes Composer *much* faster as it has less versions to think about. If you've ever wondered why you *used* to run out of memory with Composer a few years ago... but don't now... this is why.

Let's see this 4.3.* require thing in action. Spin over to the terminal, open up a new tab and run:

```
$ composer update "symfony/*"
```

If we did *not* have Symfony Flex installed, we would expect that symfony/form would be updated to 4.4. But... yea! It says:

Restricting packages listed in symfony/symfony to 4.3.*

And... when we find symfony/form, it *did* upgrade it, but only to the latest 4.3 release - *not* 4.4. You can also see that it updated a few other libraries that start with symfony/* but that aren't part of the main Symfony code. Flex has no effect on these: they upgrade normally, and that's fine.

So upgrading the "patch" version of Symfony to get bug fixes and security releases is *just* as simple as running composer update "symfony/*". But to upgrade to the next *minor* version, we need to change the extra.symfony.require key. Except... there will be one other trick. Let's see what it is next.

Chapter 2: Managing Flex, extra.symfony.require & Version Constraints

We just ran:

```
$ composer update "symfony/*"
```

Thanks to the extra.symfony.require key in our composer.json file:

```
104 lines | composer.json
1  {
    ... lines 2 - 95
96  "extra": {
97      "symfony": {
    ... lines 98 - 99
100      "require": "4.3.*"
101      }
102  }
103 }
```

Which is currently set to 4.3.*, it only upgraded things to the latest 4.3 version - not 4.4. Let's change this to 4.4.*:

```
104 lines | composer.json
1  {
    ... lines 2 - 95
96  "extra": {
97      "symfony": {
    ... lines 98 - 99
100      "require": "4.4.*"
101      }
102  }
103 }
```

But wait... why are we upgrading to Symfony 4.4? Isn't this a tutorial about upgrading to Symfony 5? Why not just go straight there? The reason is due to Symfony's, honestly, *incredible* upgrade policy. Symfony *never* breaks backwards compatibility for a minor release - like from 4.3 to 4.4. Instead, it *deprecates* code... and you can *see* what deprecated code you're using with some special tools. By upgrading to 4.4, we'll be able to see the *full* list of deprecated things we need to fix. Then we can fix them before upgrading to Symfony 5. We'll see this later.

Anyways, find your terminal and, once again, run:

```
$ composer update "symfony/*"
```

Yea! This time it *is* updating the Symfony packages to 4.4. That was easy!

[composer.json Version Constraints for symfony/ Packages](#)

Except... come on... it's never *quite* that easy. In fact, *some* Symfony packages did *not* upgrade. Check it out. Run:

```
$ composer show symfony/mailer
```

Scroll up: woh! This is still on version 4.3! Why?

Open up the composer.json file and find symfony/mailer:

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4  "require": {
  ... lines 5 - 24
25  "symfony/framework-bundle": "^4.0",
26  "symfony/mailer": "4.3.*",
  ... lines 27 - 40
41  },
  ... lines 42 - 102
103 }
```

Interesting: some packages - like symfony/form or symfony/framework-bundle are set to ^4.0 - which more or less means 4.*. But the symfony/mailer version is 4.3.*.

[Symfony Flex: composer.json Version Formatting](#)

There are two things I need to say about this. First, *usually* when you run `composer require some/package`, when Composer updates your `composer.json` file, it uses the "caret" (^) format. That's why you see ^3.0 and ^1.1.

But, when you use Symfony Flex and `composer require` a Symfony package, it *changes* that to use the * format - like 4.3.*. That's not a huge deal. In fact, it's almost an accidental feature - but it *is* nice because the best-practice is typically to control the "minor" version of your Symfony packages - that's the middle number - so that you can upgrade them all at the same time.

But... Flex didn't *always* do this. That's why, in my project, you see a mixture: some libraries like symfony/form have the "caret" format and other libraries - that were installed more recently like symfony/mailer - use the "star" format.

[Symfony Flex: symfony.extra.require is a "Soft" Requirement](#)

The *second* thing I need to tell you is that the `extra.symfony.require` config - set to 4.4.* now - is... more of a "suggestion". It doesn't *force* all Symfony packages to this version. More accurately it says:

When any symfony/ package is updated, its upgrade will be restricted to a version matching 4.4.*

But if you have a package that is *specifically* locked to 4.3.*, it won't *override* that and *force* it to 4.4.*. *That is why* symfony/mailer didn't upgrade.

[Changing symfony/ composer.json Versions](#)

If all this explanation doesn't make total sense... or you just don't care - Hey, that's ok! Here is what you need to know: whenever you upgrade Symfony to a new minor version - like 4.3 to 4.4, you need to do two things: (1) update the `extra.symfony.require` value *and* (2) update *all* the package versions to 4.4.*.

If that seems a bit redundant, it... kinda is! But changing the version next to the package to 4.4.* gives you *clear* control of what's going on... and it's how Composer *normally* works. And then, the `extra.symfony.require` config gives us a big performance boost in the background.

Let's do this next, upgrade to Symfony 4.4 and fix a few packages that ended up inside our "dev" dependencies incorrectly.

Chapter 3: Upgrading to Symfony 4.4

To upgrade from Symfony 4.3 to 4.4 - that's a "minor" version upgrade - we need to change the `extra.symfony.require` value to `4.4.*` - done! - *and* update each Symfony package version to that same value.

Updating Versions of Individual Symfony Packages

Let's get to work! I'll start with `symfony/asset`: change it to `4.4.*`. Copy that and start repeating it:

```
104 lines | composer.json
1  {
    ... lines 2 - 3
4  "require": {
    ... lines 5 - 20
21  "symfony/asset": "4.4.*",
22  "symfony/console": "4.4.*",
    ... line 23
24  "symfony/form": "4.4.*",
25  "symfony/framework-bundle": "4.4.*",
26  "symfony/mailer": "4.4.*",
27  "symfony/messenger": "4.4.*",
    ... line 28
29  "symfony/security-bundle": "4.4.*",
30  "symfony/sendgrid-mailer": "4.4.*",
    ... line 31
32  "symfony/twig-bundle": "4.4.*",
    ... line 33
34  "symfony/validator": "4.4.*",
35  "symfony/web-server-bundle": "4.4.*",
    ... line 36
37  "symfony/yaml": "4.4.*",
    ... lines 38 - 40
41  },
    ... lines 42 - 95
96  "extra": {
97  "symfony": {
    ... lines 98 - 99
100  "require": "4.4.*"
101  }
102  }
103 }
```

I *will* skip a *few* packages that start with `symfony/` because they are *not* part of the main Symfony repository - like `symfony/flex`:

104 lines | [composer.json](#)

```
1  {  
  ... lines 2 - 3  
4    "require": {  
  ... lines 5 - 22  
23      "symfony/flex": "^1.0",  
  ... lines 24 - 40  
41    },  
  ... lines 42 - 102  
103 }
```

These follow their own release schedules... so they usually have a version that's very different than everything else.

All "packs" - those are the, sort of, "fake" packages that just require other packages for convenience - are another example:

104 lines | [composer.json](#)

```
1  {  
  ... lines 2 - 3  
4    "require": {  
  ... lines 5 - 27  
28      "symfony/orm-pack": "^1.0",  
  ... lines 29 - 30  
31      "symfony/serializer-pack": "^1.0",  
  ... line 32  
33      "symfony/twig-pack": "^1.0",  
  ... lines 34 - 40  
41    },  
  ... lines 42 - 102  
103 }
```

These usually allow pretty much any version of the libraries inside of them - so any Symfony packages *will* update correctly. If you want more control over the versions, remember that you can run:

```
$ composer unpack symfony/orm-pack
```

When you do that, Flex will remove this line and replace it with the individual packages so you can manage their versions. That's not required, but also not a bad idea.

WebpackEncoreBundle is another example of a package that isn't part of the main repository - you can see that its version is totally different:

104 lines | [composer.json](#)

```
1  {  
  ... lines 2 - 3  
4    "require": {  
  ... lines 5 - 35  
36      "symfony/webpack-encore-bundle": "^1.4",  
  ... lines 37 - 40  
41    },  
  ... lines 42 - 102  
103 }
```

Don't forget to also check the require-dev section: there are a bunch here:

```

104 lines | composer.json
1  {
    ... lines 2 - 41
42  "require-dev": {
    ... lines 43 - 45
46      "symfony/browser-kit": "4.4.*",
47      "symfony/debug-bundle": "4.4.*",
48      "symfony/dotenv": "4.4.*",
    ... lines 49 - 50
51      "symfony/phpunit-bridge": "4.4.*",
    ... line 52
53      "symfony/var-dumper": "4.4.*"
54  },
    ... lines 55 - 102
103 }

```

Including symfony/debug-bundle, which has a funny-looking version because I unpacked it from a debug-pack in one of our courses. And both MakerBundle and MonologBundle are not in the main repository:

```

104 lines | composer.json
1  {
    ... lines 2 - 41
42  "require-dev": {
    ... lines 43 - 48
49      "symfony/maker-bundle": "^1.0",
50      "symfony/monolog-bundle": "^3.0",
    ... lines 51 - 53
54  },
    ... lines 55 - 102
103 }

```

If you're not sure, you can search Packagist.org for symfony/symfony. That package lists *all* of the packages that make up this "main" repository I keep talking about.

Update phpunit-bridge, leave the profile-pack version and update var-dumper:

```

104 lines | composer.json
1  {
    ... lines 2 - 41
42  "require-dev": {
    ... lines 43 - 50
51      "symfony/phpunit-bridge": "4.4.*",
    ... line 52
53      "symfony/var-dumper": "4.4.*"
54  },
    ... lines 55 - 102
103 }

```

Perfect! We have 4.4.* everywhere up here *and* 4.4.* for extra.symfony.require so that everything matches *and* we get that performance boost in Composer.

Let's do this! Find your terminal and run:

```
$ composer update "symfony/*"
```

And... yea! It's upgrading the last few libraries that were *previously* locked to 4.3. Congratulations! You just upgraded *all*

Symfony packages to 4.4.

Fixing some require-dev Packages

Before we move on, I noticed a small problem in composer.json: the symfony/dotenv package is in my require-dev section:

```
104 lines | composer.json
1  {
  ... lines 2 - 41
42  "require-dev": {
  ... lines 43 - 47
48      "symfony/dotenv": "4.4.*",
  ... lines 49 - 53
54  },
  ... lines 55 - 102
103 }
```

When we put something in require-dev, we're saying:

This package is *not* needed when I run my code on production.

It was true that when Symfony 4.0 was released, the DotEnv component was used in the development environment only - as a way to help set environment variables more easily. That's not true anymore: Symfony apps now *always* load the .env files.

The symfony/monolog-bundle package - which gives us the logger service - should *also* live under require - along with its supporting package: easy-log-handler:

```
104 lines | composer.json
1  {
  ... lines 2 - 41
42  "require-dev": {
  ... line 43
44      "easycorp/easy-log-handler": "^1.0.2",
  ... lines 45 - 49
50      "symfony/monolog-bundle": "^3.0",
  ... lines 51 - 53
54  },
  ... lines 55 - 102
103 }
```

Logging is something we *always* want.

Let's fix these. Copy the symfony/dotenv package name, find your terminal, and *remove* these three packages:

```
$ composer remove --dev symfony/dotenv symfony/monolog-bundle easycorp/easy-log-handler
```

An easy way to move a package from require-dev to require and make sure that Composer notices, is to remove the package and re-add it.

When we do that... our code explodes! No problem: our app *totally* needs the DotEnv component... so it's temporarily freaking out. You'll also notice that, if you run:

```
$ git status
```

Removing these packages *also* removed their recipes. Re-add the libraries by using that same command, but replacing remove with require and getting rid of the --dev flag:

```
$ composer require symfony/dotenv symfony/monolog-bundle easycorp/easy-log-handler
```

Tip

The easycorp/easy-log-handler package is abandoned, so it's probably even better to remove it from this list and leave it out of your app

This should add those back under the require section - yep, here is one - *and* it will reinstall the *latest* version of their recipes... which means that the recipe *could* be slightly newer than the one we had before:

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4    "require": {
  ... lines 5 - 7
8      "easycorp/easy-log-handler": "^1.0",
  ... lines 9 - 23
24     "symfony/dotenv": "4.4.*",
  ... lines 25 - 29
30     "symfony/monolog-bundle": "^3.5",
  ... lines 31 - 43
44   },
  ... lines 45 - 102
103 }
```

This is... accidentally... the first example of upgrading a recipe. Run:

```
$ git status
```

Cool. Should we commit all of these changes? Not so fast. When a recipe is updated, you need to *selectively* add each change. Let's learn how next.

Chapter 4: Selectively Committing Recipe Updates

We just... sort of... *accidentally* updated the MonologBundle recipe by removing that package and reinstalling it. Doing that modified *several* files.

Let's add the changes we *know* we want to keep to git:

```
$ git add composer.json composer.lock symfony.lock
```

For the *other* changes - the ones the recipe made - we need to be *very* careful. Why? Because recipes don't really "upgrade" in some clean way. Nope, when we removed the packages, some config files we *deleted*... which means any custom code we had in those was *removed*. When we reinstalled the packages, the recipes re-added these files... but any custom code we had is *gone*.

So, we *do* want to "accept" any new, cool changes to these files that a newer version of the recipe may have added. But we *do not* want our custom code to disappear.

[Changeset Swiss Army Knife: git add -p](#)

My favorite way to sort all of this out is to run:

```
$ git add -p
```

This interactive command looks at *every* change one-by-one and asks whether or not you want to add it. For bundles.php... this isn't a file that we usually add custom code to - so it should be safe. It *looks* like it's *removing* MonologBundle, but it actually just *moved* this line. Hit y to add this change. And... yep! Here is the line being added-back. Hit y again. That change was meaningless.

The next change is in config/packages/dev/monolog.yaml: it wants to remove a markdown_logging handler. Hey! No! This is *our* custom code. Say n to *not* add this change.

[Updated Recipes Show New Features](#)

Finally, in the production monolog.yaml file, it changed excluded_404s to excluded_http_codes. This is *awesome*. The excluded_http_codes - which is basically a way to help you log errors... but not things like 404 errors - is a relatively new feature that didn't exist when we originally installed MonologBundle. The updated recipes is telling us about a feature that we *may* not know about it!

Should we accept this change? It's up to you. Do you like this new way of filtering logs? I do: because I don't like having 405 errors in my logs: that's when someone, for example, makes a GET request to a URL that only allows POST requests. Sometimes a bot will do that. Let's hit "y" to add this change.

And... it's done! Run:

```
$ git status
```


to see what it did. Cool. All the changes we *do* want are up in the "staged" area and ready for commit. The one change that is *not* staged - down here in red - was the one change that we did *not* want to commit.

[Removing Unwanted Changes](#)

To *undo* that change - so it goes back to the way it was before - run:

```
$ git checkout config/packages/dev/monolog.yaml
```

Now you can safely commit these changes however you want, like:



```
$ git commit -m "moving packages into require"
```

I'll let you make that commit.

Congrats! You just got your first experience *upgrading* a recipe. Was it *necessary*? Not really. The newer version of MonologBundle would have worked *fine* if we had kept our existing config files *exactly* like they were. But it *did* teach us about a new feature... which was kind of awesome.

Next: let's start updating our recipes for *real*. We'll learn about some new commands that Flex adds to composer to help with this.

Chapter 5: Upgrading Recipes: New Commands!

Fun fact time! When you start a brand new Symfony project, behind the scenes, what you're *actually* doing is cloning this repository: symfony/skeleton. Yep, your app *literally* starts as a single composer.json file. But as *soon* as Composer installs your dependencies, the app is suddenly filled with a few directories and about 15 files.

All Config Files Come from a Recipe

All of those things are added by different *recipes*. So even the most "core" files - for example, public/index.php, the file that our web server executes, is added by a recipe!

```
28 lines | public/index.php
... lines 1 - 2
3  use App\Kernel;
4  use Symfony\Component\Debug\Debug;
5  use Symfony\Component\HttpFoundation\Request;
6
7  require dirname(__DIR__).'/config/bootstrap.php';
8
9  if ($_SERVER['APP_DEBUG']) {
10     umask(0000);
11
12     Debug::enable();
13 }
14
15 if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ?? $_ENV['TRUSTED_PROXIES'] ?? false) {
16     Request::setTrustedProxies(explode(',', $trustedProxies), Request::HEADER_X_FORWARDED_ALL ^ Request::HEADER_X_FORWARDED_FOR);
17 }
18
19 if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? $_ENV['TRUSTED_HOSTS'] ?? false) {
20     Request::setTrustedHosts([$trustedHosts]);
21 }
22
23 $kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
24 $request = Request::createFromGlobals();
25 $response = $kernel->handle($request);
26 $response->send();
27 $kernel->terminate($request, $response);
```

We pretty much *never* need to look inside here or do anything, even though it's *critical* to our app working.

Another example is config/bootstrap.php:

52 lines | [config/bootstrap.php](#)

... lines 1 - 2

```
3 use Symfony\Component\Dotenv\Dotenv;
4
5 require dirname(__DIR__).'./vendor/autoload.php';
6
7 // Load cached env vars if the .env.local.php file exists
8 // Run "composer dump-env prod" to create it (requires symfony/flex >=1.2)
9 if (is_array($env = @include dirname(__DIR__).'./env.local.php')) {
10     $_SERVER += $env;
11     $_ENV += $env;
12 } elseif (!class_exists(Dotenv::class)) {
13     throw new RuntimeException("Please run \"composer require symfony/dotenv\" to load the \".env\" files configuring the application.");
14 } else {
15     $path = dirname(__DIR__).'./env';
16     $dotenv = new Dotenv();
17
18     // load all the .env files
19     if (method_exists($dotenv, 'loadEnv')) {
20         $dotenv->loadEnv($path);
21     } else {
22         // fallback code in case your Dotenv component is not 4.2 or higher (when loadEnv() was added)
23
24         if (file_exists($path) || !file_exists($p = "$path.dist")) {
25             $dotenv->load($path);
26         } else {
27             $dotenv->load($p);
28         }
29
30         if (null === $env = $_SERVER['APP_ENV'] ?? $_ENV['APP_ENV'] ?? null) {
31             $dotenv->populate(array('APP_ENV' => $env = 'dev'));
32         }
33
34         if ('test' !== $env && file_exists($p = "$path.local")) {
35             $dotenv->load($p);
36             $env = $_SERVER['APP_ENV'] ?? $_ENV['APP_ENV'] ?? $env;
37         }
38
39         if (file_exists($p = "$path.$env")) {
40             $dotenv->load($p);
41         }
42
43         if (file_exists($p = "$path.$env.local")) {
44             $dotenv->load($p);
45         }
46     }
47 }
48
49 $_SERVER['APP_ENV'] = $_ENV['APP_ENV'] = ($_SERVER['APP_ENV'] ?? $_ENV['APP_ENV'] ?? null) ?: 'dev';
50 $_SERVER['APP_DEBUG'] = $_SERVER['APP_DEBUG'] ?? $_ENV['APP_DEBUG'] ?? 'prod' !== $_SERVER['APP_ENV'];
51 $_SERVER['APP_DEBUG'] = $_ENV['APP_DEBUG'] = (int) $_SERVER['APP_DEBUG'] || filter_var($_SERVER['APP_DEBUG'], FILTER
```

the boring, low-level file that initializes and normalizes environment variables. It's important that all Symfony projects have the same version of this file. If they didn't, some apps might work different than others... even if they have the same version of Symfony. Think of trying to write documentation for thousands of projects that all work a *little* bit differently. It's literally my

nightmare.

All of the configuration files were *also* originally added by recipes. For example, `cache.yaml` comes from the recipe for `symfony/framework-bundle`:

```
21 lines | config/packages/cache.yaml
1  framework:
2    cache:
  ... lines 3 - 13
14    # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
15    app: '%cache_adapter%'
16
17    # Namespaced pools use the above "app" backend by default
18    pools:
19      cache.flysystem.psr6:
20        adapter: cache.app
```

Why Recipes Update

Over time, the recipes *themselves* tend to change. If we installed the `symfony/framework-bundle` today, it *might* give us a slightly *different* `cache.yaml` file.

There are three reasons that a recipe might change. First, someone might update a recipe *just* because they want to add more examples or add some documentation comments to a config file. Those changes... aren't super important.

Or, second, someone might update a configuration file inside a recipe to activate a new feature that's probably from a new version of that library. These changes aren't *critical* to know about... but it *is* nice to know if a great new feature is suddenly available. We saw that a few minutes ago when the updated `MonologBundle` recipe told us about a cool option for filtering logs by status code.

The third reason a recipe might update is because something needs to be fixed, or we decide that we want to change some significant behavior. These changes *are* important.

Let me give you an example: during the first year after Symfony 4.0, several small but meaningful tweaks were made to the `bootstrap.php` file to make sure that environment variables have *just* the right behavior. If you started your project on Symfony 4.0 and never "updated" the `bootstrap.php` file, your app will be handling environment variables in a different way than other apps. That's... not great: we want our `bootstrap.php` file to look *exactly* like it should.

New Recipe Commands!

A few minutes ago, when we did all the composer updating stuff, one of the packages that we upgraded was `symfony/flex` itself: we upgraded it to 1.6.0:

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4    "require": {
  ... lines 5 - 24
25    "symfony/flex": "^1.0",
  ... lines 26 - 43
44  },
  ... lines 45 - 102
103 }
```

Well guess what?! Starting in Flex 1.6.0, there are some brand new fancy, amazing, incredible commands inside Composer to help inspect & upgrade recipes. It still takes a little bit of work and care, but the process is now very possible. A big thanks to community member and friend [maxhelias](#) who really helped to get this done.

Let's go check them out! Move over to your terminal and run:



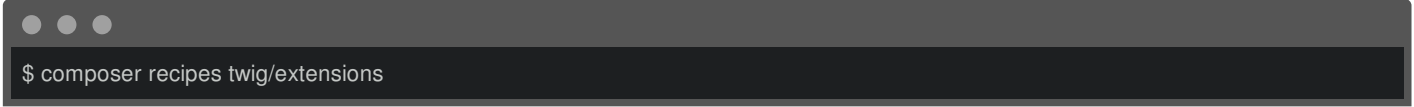
```
$ composer recipes
```

Cool! This lists *every* recipe that we have installed and whether or not there is an update available. Heck, it will even show you if a package that's installed *has* a recipe that you're missing - maybe because it was added later.

Because my project was originally created on Symfony 4.0, it's fairly old and a *lot* of recipes have updates. The recipe system is *also* relatively new, so I think there were more updates during the first 2 years of that system than there will be in the *next* two years. We've got some work to do. Of course, we could just *ignore* these recipe updates... but we're risking our app not working quite right or missing info about new features.

[Inspecting the twig/extension Recipe](#)

Let's look at one of these more closely. How about twig/extensions. This is not a particularly important library, but it's a nice one to start with. Run:



```
$ composer recipes twig/extensions
```

to see more details. Interesting: it has a link to the installed version of the recipe. Let's go check that out in the browser. Paste and... this is what the recipe looked like the *moment* we installed it. We can also go grab the URL to see what the *latest* version of the recipe looks like.


Check out the commit history. The version of the recipe *we* have installed has a commit hash starting with c986. Back on the history, hey! That commit is right here! So this recipe *is* out of date, but the *only* change that's been made is this *one* commit. Inside it... search for twig/extensions to find its changes. Ha! It's totally superficial: we changed from using tilde (~) to null... but just for clarity: those are equivalent in YAML.

Yep! The update to twig/extension is not important *at all*. We *could* still update it - and I'll show you how next. But I'm going to skip it for now. Because... this is a tutorial about upgrading Symfony! So I want to focus on upgrading the recipes for everything that starts with symfony/.

Let's start that process next by focusing on, surprisingly, one of the *most* important recipes: symfony/console.

Chapter 6: Recipe Upgrade: symfony/console & bootstrap.php

The first recipe I want to update is symfony/console. Let's get some more info about it:



```
$ composer recipes symfony/console
```

Just like before, it shows us some links: a link to what the recipe looked like at the moment we installed it - I'll paste that into my browser - and also a version of what the recipe looks like right now. Let's open up that one too.

[Recipe Organization & History](#)

We're not going to *always* study the history of a recipe like this, but I need to show you something. Because of the way that recipes are organized, if you want see what changed in a recipe since you installed it, it's not *always* as easy as looking at the history.

This is what the recipe looks like today. Click to look at the commit history. You *might* think:

Okay, I'll just go back and see what commits have been made to this recipe since my version.

That's a great idea, because sometimes knowing *why* something changed can be a huge help. But... the installed version of my recipe lives in a symfony/console/3.3/ directory. The new one lives in symfony/console/4.4/.

Let's look at this *entire* symfony/console directory. Oh, and make sure that you're looking at the master branch - the latest commits. Each recipe is allowed to have multiple versions. If you installed symfony/console version 3.3, Flex would install the 3.3 recipe. If you installed 4.2, you would get the 4.2 recipe. What if you installed symfony/console 4.1? You would get the 3.3 recipe. A new directory is created *only* when a recipe needs to be updated to show off a feature that's only available in a newer version.

So, it's kind of a strange versioning mechanism. When we installed this recipe, we installed the 3.3 version. Updating it would install the 4.4 version. So if you want to see the full history... it's tricky: you kind of need to look at the history of what commits have been made to the 4.4 branch... and also maybe the history of what's been done to the 4.2 branch.... and also maybe the commit history for the 3.3 branch.


But, it's not as bad as it sounds - I'm trying to deliver the bad news first. Most of the time, the *reason* a file was updated in a recipe will be pretty obvious. And when it's not, with a little digging, we can find the reason.

[Let's update a Recipe](#)

Okay, so how do we *actually* upgrade a recipe to the latest version? You can see the answer down here:


```
composer recipes:install symfony/console --force -v.
```

But, it's not *really* a smart "update" system. That command tells Flex to completely re-install the symfony/console recipe using the latest version. Try it:



```
$ composer recipes:install symfony/console --force -v
```


Nice! Thanks to the -v flag, it tells us what files it worked on. It says:



```
$ Created bin/console
$ Created config/bootstrap.php
```


Well, really, it *modified* those files... but at least in the version of Flex I'm using, it always says "created".

Let's see how things look. Run:



```
$ git status
```

Cool! 3 changed files. Just like with the MonologBundle recipe, we need to add these changes *carefully*: if we had any custom code in the files, the update process just *replaced* it. Run:



```
$ git add -p
```

The bin/console changes

The first file is bin/console... and it's a namespace change from Debug to ErrorHandler. This is updating our code to use a new ErrorHandler component - some features of the Debug component were moved there. So that's a good change.

It also... I don't know, added some if statement that prints a warning... it looks like it's just making sure we don't try to run this from inside a web server. Enter "y" to add both changes.

Investigating a Change

Next, woh! It removed a huge block of code from the bottom and tweaked an if statement further up. These changes are on config/bootstrap.php.

This... *looks* like some low-level, edge-case normalization of environment variables. So *probably* we want this. But let's pretend we don't know: we want to find out *why* this change was made.

How? By doing some digging! Go back to symfony/recipes/console. Start by looking in the 4.4/ directory - the version we're installing. Find config/bootstrap.php.

Wait... see the config/ directory? What does that little arrow mean? It means that this is actually a *symbolic link* to another directory: the 4.4/config directory is identical to 4.2/config.

Ok, let's go look there! Head into the 4.2/ directory, then config/. Woh! Another arrow! This time the bootstrap.php file is a symlink - pointing to, wow! A totally different recipe - a bootstrap.php file in framework-bundle.

The bootstrap.php file is *the* most complex file in the recipe system and it's shared across several recipes. Yep, I'm showing you the ugliest case.

Let's go find that: symfony/framework-bundle/4.2/config/bootstrap.php. *Here* is the file. To find the change, use the "blame" feature. Ok, the block of code we're looking at - lines 9 through 12 - have two different commits. Let's look at just one of them:

```
Allow correct environment to be loaded when .env.local.php exists
```

And we can even click to see the pull request: [#647](#) if you want *really* dive into the details. In this case, the change fixes a bug if you use the .env.php.local file.

But, *really*, config/bootstrap.php is a low-level file that should almost *always* be identical in every project. So unless you're doing something super advanced, you will probably want to accept *all* of these changes.

This big removal of code? That was because in an earlier version of this recipe, your project may or may *not* have had this loadEnv() method on DotEnv: it was added in Symfony 4.2. If your app did *not* have that method, it added a *bunch* of code to "imitate" its behavior. We don't need that anymore. Thank you recipe update!

The last change is for the symfony.lock file. We don't even need to look at this: always accept these changes. This marks the recipe as *updated* and sometimes saves extra debugging info that might be useful later.

That *may* have seemed like a small step. But other than trying to figure out the *reasons* a file changed, this was a home run! We were able to update two low-level files, which will help make sure our app continues to work like we expect.

Go ahead and commit these changes. Then let's keep going. Next, we'll update the biggest and most important recipe: the one for symfony/framework-bundle.

Chapter 7: Upgrading the FrameworkBundle Recipe (Part 1)

Run:

```
$ composer recipes
```

[Updating symfony/flex](#)

Our goal is to update all of the recipes starting with symfony/. The hardest ones are at the beginning: symfony/console and symfony/framework-bundle. But right now, let's update symfony/flex itself. Run:

```
$ composer recipes symfony/flex
```

... because that's an easy way to get the update command. Run it:

```
$ composer recipes:install symfony/flex --force -v
```

Hmm, it looks like it only modified one file: .env. Take a look with:

```
$ git status
```

Yup! Just that one. Check it out:

```
$ git diff
```

Ok: two changes. The first one is a fix for a typo in a comment. Then... it deleted a bunch of *my* code. Rude! Ok, we expected that: this is not a true *update* process: the new .env file from the recipe *overrode* mine completely.

So this recipe update was to fix a meaningless typo. That's *super* minor, but I guess we want that change. Hit "Q" to get out of this mode. Then run:

```
$ git add -p
```

I will accept the typo change - y - but not the rest - n. Add the symfony.lock changes as usual. Ok, run:

```
$ git status
```

Two changes staged and ready to commit and one unstaged change to .env. Let's commit the staged updates:

```
$ git commit -m "updating symfony/flex"
```

Cool! Now git diff tells us that the only remaining change is the removal of the stuff that we *do* want in .env. Revert all of that by running:

```
$ git checkout .env
```

Done!

[Upgrading the symfony/framework-bundle Recipe](#)

Let's check our progress:

```
$ composer recipes
```

Another one done! Take a deep breath and move onto the *biggest*, most important recipe: symfony/framework-bundle. Run:

```
$ composer recipes symfony/framework-bundle
```

Hmm, yea, we're upgrading from version 3.3 of the recipe to 4.4: that might be a fairly big upgrade. Copy the recipes:install command and run it:

```
$ composer recipes:install symfony/framework-bundle --force -v
```

Apparently this modified *several* files. You know the drill: let's start walking through the changes by running:

```
$ get add -p
```

[Changes to .env](#)

The first change is inside .env - it updated APP_SECRET. This recipe has a special power: each time you install it, it generates a *new* unique value for APP_SECRET, which is used to generate some cryptographic stuff in your app. We don't really need or want to change this value.

[Hunting Down the Reason for a Change](#)

What about the change right below it - for TRUSTED_PROXIES? We're not using that value anyways - you can see that both the old and new code are commented out.

But, as a challenge, let's see if we can find *what* this change is all about. Go back to the homepage of the symfony/recipes repository and then navigate to symfony/framework-bundle/. We're installing the 4.4 recipe, so start there.

Most of the time, a recipe simply copies files into your project. And so we're *usually* comparing the contents of a file between two recipes.

But there are a couple of *other* things a recipe can do, like *modify* your .env or .gitignore files. In those cases, you won't see a .env or .gitignore file in the recipe: those changes are described in this manifest.json file.

Ah! A symlink - this points to the 4.2 version. I'll take a shortcut and change the URL to jump to that file.

manifest.json is the config file that describes everything the recipe does. The env key says:

```
Hey! I want you to update the .env file to add APP_ENV, APP_SECRET and these two TRUSTED comment lines.
```

Let's "blame" this file. The TRUSTED_PROXIES line was modified about three months ago. Click that commit... and jump to the pull request - 654 - to get the full details.

Ok: "Trusted proxies on private and local IPs". This links to *another* issue on the main Symfony repository where someone proposes that private IP address ranges should be trusted by default.

If you're not familiar with TRUSTED_PROXIES, then you probably don't care much about this and... you might as well just accept the update. If you *do* care, you'll understand that this PR marks *private* IP ranges as "trusted", which may or may not be useful for you. The point is: we figured out the reason for this change and - if we use this feature - we can accept or reject these changes intelligently.

Because we *don't* want the APP_SECRET change... and I don't really care about the updated comment line, I'll say "n" to skip both changes.

The next file that's modified is .gitignore. Let's talk about this next as well as changes to framework.yaml and *super* important updates to the Kernel class.

Chapter 8: FrameworkBundle Recipe Part 2: The Kernel Class

We're *right* in the middle of upgrading the FrameworkBundle recipe.

[Updates to .gitignore](#)

It apparently added a new line to our .gitignore file: some config/secrets/prod/prod.decrypt.private.php file.

We don't have that file yet. What is it? Oh, I can't *wait* to talk about it! It's part of Symfony's new secrets management system. OooOOooo. For now, say yes: we *will* want to ignore this file later when we create it.

[Updates to cache.yaml](#)

The next change is inside cache.yaml. It... yea... just updated some comments. We don't need that, but it's nice: type "y" to add them. Oh, and the next change is *also* from cache.yaml - further down. It looks like they changed the example config... but we've already customized this. So, we do *not* want this. Enter "n" to *not* add it.

[Updates to framework.yaml](#)

Next up is framework.yaml. Interesting, two changes here: it removed the default_locale and added two new cookie settings. I won't make you dig through the commit history to find the explanation behind these. default_locale was removed from here because it is *also* defined in translation.yaml... and someone realized it was pointless and a bit confusing to have it in both places.

The cookie settings are a bit more interesting: they activate two security-related features. The first is cookie_secure. The auto setting means that *if* a visitor comes to your site via HTTPS, then it will create an HTTPS-only cookie. It's a no brainer and you should exclusively be using HTTPS on your production site anyway.

The cookie_samesite option activates a feature on your cookies called... well... "samesite". It's a relatively new security-related feature that's quickly been adopted by most browsers. We talk more about it inside our [API Platform Security](#) tutorial - but this setting shouldn't cause problems in most setups and is definitely more secure.


So let's say "y" for *all* of these changes.

[Updates to services.yaml](#)

Keep going! Now we're inside services.yaml! Hmm... it looks like it's just removing a bunch of stuff! That's because we've customized *most* of this file. If you look closely, there *is* one change: the recipe apparently removed the public: false line along with the comments describing it.

Why? Because since Symfony 4.0, public: false is the *default* value. We actually *never* needed this config! It was included originally... mostly for historical reasons.

So we *do* want this change... but... we can't say "yes" to this because it would kill *all* our code. Enter "q" to get out of the git add -p system. We'll need to make this change manually. First, undo *all* of the changes by running:



```
$ git checkout config/services.yaml
```

Move back and look at the file in our editor... the custom code is back! *Now* manually take out the public: false line and the comments below it:

54 lines | [config/services.yaml](#)

... lines 1 - 12

13 services:

14 # default configuration for services in *this* file

15 _defaults:

... lines 16 - 17

18 public: false # Allows optimizing the container by removing unused services; this also means

19 # fetching services directly from the container via \$container->get() won't work.

20 # The best practice is to be explicit about your dependencies anyway.

... lines 21 - 54

Let's continue the process. Start again with:

```
$ git add -p
```

It's going to ask us about a few changes we've already said no to - say no again. And this time, for the services.yaml change, enter "y" to add it.

[Updates to public/index.php](#)

The next change is inside public/index.php. Hey! It's that namespace change from Debug to ErrorHandler. We *know* that's a good change. If we *did* skip this, we would see a deprecation warning telling us to make that change. So upgrading the recipes is... actually saving us time later!

[Updates to Kernel.php](#)

Finally, we get to the *most* important file of the recipe: src/Kernel.php. This is another file that you *probably* haven't added custom changes to. And so, it's *probably* safe to accept all these updates. But let's look carefully and I'll highlight the reason behind a few changes. It looks like a lot, but it's all minor.

For example, PHP 7.1 allows you to have *private* constants. The recipe update uses that. No big deal. The getCacheDir() and getLogDir() methods aren't needed anymore because they're implemented by the *parent* class with the same logic. Removing them is a nice cleanup.

And registerBundles() now has an iterable return type.

I'll clear my screen then answer "y" to add these changes.

Next, it *added* a getProjectDir() method. This *used* to not be needed, because Symfony determined it automatically by searching for your composer.json file. But since they didn't work correctly in some edge-cases, it's added directly in our class now. *Probably* not a super important thing for us, but we'll accept this change.

Next, configureContainer() has a void return type and some parameters got tweaked. The autowiring.strict_mode parameter was removed because it was something that made Symfony 3 behave like Symfony 4 does by default. It's not needed anymore... and never was. Clean up!

Then, inline_factories is a performance thing - cool - and there's a *slight* tweak to how the config files are loaded to make life faster in the development environment: it no longer looks recursively for files inside the environment config directories - like config/packages/dev.

At the bottom, configureRoutes() has a void return type and a similar recursive tweak. Say "y" to add all of this.

And... we're done! This is symfony.lock: definitely accept these changes.

[Adding config/routes/dev/framework.yaml](#)

Let's check out how things look:

```
$ git status
```

Oh! The recipe added a *new* file: `config/routes/dev/framework.yaml`. Interesting. Let's go open that: `config/routes/dev/framework.yaml`:

```
4 lines | config/routes/dev/framework.yaml
1  _errors:
2    resource: '@FrameworkBundle/Resources/config/routing/errors.xml'
3    prefix: /_error
```

You may or may not know, but Symfony has a feature that allows you to test what your production error pages look like. Just go to `/_error/404` to see the 404 page or `/_error/500` to see the 500 error page... though... ha, *that*, uh, never happens on production.

This file loads an `errors.xml` file that adds this route in the dev environment only.

Previously, if you open the `twig.yaml` file in the same directory, this feature came from TwigBundle:

```
4 lines | config/routes/dev/twig.yaml
1  _errors:
2    resource: '@TwigBundle/Resources/config/routing/errors.xml'
3    prefix: /_error
```

Now it lives inside FrameworkBundle. *That* is why the framework-bundle recipe added the new file.

Hmm... but since we haven't updated the TwigBundle recipe yet, we temporarily have *two* routing files that are trying to add a route to the *same* `/_error` URL. We'll update the TwigBundle recipe next to fix this.

Right now, add this file:

```
$ git add config/routes/dev/framework.yaml
```

And run:

```
$ git status
```

Hmm... yep! These last changes are the ones we do *not* want. Revert them with:

```
$ git checkout .
```

We're ready to commit the *biggest* recipe upgrade we're going to have:

```
$ git commit -m "upgrading symfony/framework-bundle recipe"
```

Phew! Next, let's update the TwigBundle recipe then keep going onto the Mailer recipe and then the rest. Home stretch people!

Chapter 9: Updating the TwigBundle Recipe

The updated framework-bundle recipe gave us this new routing file: config/routes/dev/framework.yaml:

```
4 lines | config/routes/dev/framework.yaml
1  _errors:
2    resource: '@FrameworkBundle/Resources/config/routing/errors.xml'
3    prefix: /_error
```

Which loads a `/_error/{statusCode}` route where we can test what our *production* error pages look like.

This feature *used* to live in TwigBundle... which is why twig.yaml has basically the exact same import:

```
4 lines | config/routes/dev/twig.yaml
1  _errors:
2    resource: '@TwigBundle/Resources/config/routing/errors.xml'
3    prefix: /_error
```

This is a minor problem. In your terminal, run:

```
$ php bin/console debug:router
```

and scroll up to the top. Yep! We have *two* routes for the *exact* same URL. The first one - which *by chance* is the one from FrameworkBundle - would win, but we *still* don't want the old one sitting there. Plus, it's deprecated and will disappear in Symfony 5.

We need to delete this twig.yaml file. But... we *probably* also need to update the TwigBundle recipe... which will *probably* delete it for us. Run:

```
$ composer recipes
```

Yep! The recipe for symfony/twig-bundle has an update.

[Updating symfony/twig-bundle Recipe](#)

Get some info about it:

```
$ composer recipes symfony/twig-bundle
```

Then copy the recipes:install command and run it:

```
$ composer recipes:install symfony/twig-bundle --force -v
```

Perfect! It looks like it modified three files. Let's start walking through them:

```
$ git add -p
```

The first change is inside the twig.yaml config file. If you ignore the stuff that it's removing - that's all *our* custom code - it looks

like the updated recipe *added* a line: `exception_controller: null`.

Ok, so we definitely want to keep our custom changes... and we *probably* want to keep this new line... except that we don't really know *why* it was added.

[Checking CHANGELOGs](#)

Let's go do some digging! But this time, instead of checking the recipe commit history, let's try something different. Because this is a config change for TwigBundle, let's go see if they mention this in a CHANGELOG.

Google for "GitHub TwigBundle" to find its GitHub page. Scroll down and... yea! It has a CHANGELOG.md file.

Open it up and look at the 4.4.0 changes. Actually, this `exception_controller` change *could* even be from an earlier version - but we'll start here. And... yea, it *does* talk about it:

deprecated twig.exception_controller configuration option, set it to "null" and use framework.error_controller configuration instead.

[The deprecated twig exception_controller Option](#)

This is *another* feature that was deprecated inside TwigBundle and *moved* to FrameworkBundle. The exception, or "error", controller is the controller that's responsible for rendering an error page.

To *disable* - basically "stop using" the deprecated *old* code - we need to set `exception_controller` to null. *That* is why the recipe added this change. This *is* a good change. Of course, if your config file already has an `exception_controller` option... because you're using a *custom* exception controller, you'll need to *move* that value to `framework.error_controller` and do some reading to see if your controller code needs any other updates.

So we *do* want this change... but we can't accept this patch without killing our custom code. Copy the new config, hit "q" to quit this mode, and then... let's see - undo those changes by running:

```
$ git checkout config/packages/twig.yaml
```

Oh, and I guess I should spell "checkout" right.

Now, spin back over, open that file - `config/packages/twig.yaml` - and add `exception_controller: null`:

```
11 lines | config/packages/twig.yaml
1  twig:
    ... lines 2 - 9
10 exception_controller: null
```

Nice! Let's... keep going: start the `git add -p` system again:

```
$ git add -p
```

This time we *do* want to accept the change to `twig.yaml` - "y" - and the next change is inside `symfony.lock`. Accept that too.

[base.html.twig and the new test/twig.yaml](#)

The *last* updated file is `templates/base.html.twig` and we definitely *do not* want to accept this change and kill our custom layout. Looking at the new code... I don't see anything super important that we might want to add. In fact, if you checked the recipe history, there haven't been *any* updates to this file in years. Hit "n" to ignore this.

Run:

```
$ git status
```

to see how things look. Oh! A new file: `config/packages/test/twig.yaml` - a config file that's *only* loaded in the test environment. Before we see what's inside it, let's revert the changes we don't want:

```
$ git checkout templates/base.html.twig
```

Go open the new file: `config/packages/test/twig.yaml`:

```
3 lines | config/packages/test/twig.yaml
```

```
1 twig:
2   strict_variables: true
```

Ah, super minor: it sets `strict_variables` to `true` for our tests. This settings tells Twig to throw an exception if we try to use an undefined variable in a template. If we ever did that, we probably *would* want Twig to explode in our tests so we know about it. That's a good change. Add that file:

```
$ git add config/packages/test/twig.yaml
```

[Manually Deleting `config/routes/dev/twig.yaml`](#)

We're done! But... wait a second. We *expected* that the updated recipe would delete the extra `config/routes/dev/twig.yaml` file... but it didn't. Hmm... is it *still* in the recipe for some reason? Run:

```
$ composer recipes symfony/twig-bundle
```

Copy the URL to the recipe... and paste it in your browser. Huh. No - there is *no* `config/routes` directory at *all* in here. The file *is* gone! Why wasn't it deleted?

This is a shortcoming of the recipe update system: it's not smart enough. In a perfect world, it would realize that there *used to be* a `config/routes/dev/twig.yaml` file in the *old* version of the recipe... and since it is *not* there in the new version, it would delete it. But, that does *not* happen, at least not yet.

So, we need to delete it manually. This doesn't happen very often, but it *is* something you should be aware of.

Back at the terminal, run:

```
$ git status
```

one more time - things look good - and let's commit:

```
$ git commit -m "updating symfony/twig-bundle"
```

Nice! Now run:

```
$ composer recipes
```

We're getting close! Let's do any easy one next: let's upgrade `symfony/mailer` *and* `symfony/sendgrid-mailer`.

Chapter 10: Updating the Mailer Recipe(s)

The next recipe on our list is symfony/mailer... which is an *especially* interesting one because, until Symfony 4.4, symfony/mailer was marked as "experimental". That means that there *are* some backwards-compatibility breaks between Symfony 4.3 and 4.4. The recipe update *might* help us find out about a few of these.

Run:

```
$ composer recipes symfony/mailer
```

Then copy the recipes:install command and run it:

```
$ composer recipes:install symfony/mailer --force -v
```

According to the output, this only touched *one* file. Let's see for sure. Run:

```
$ git status
```

Yep! Only .env was changed. Run:

```
$ git add -p
```

Hmm. It looks like it *removed* two comment lines, which mention that the MAILER_DSN for the null transport looks different in Symfony 4.4. And then it added an example of using the smtp transport. The top line is my custom code.

I don't *really* want these changes. I mean, I *do* still want to define a MAILER_DSN environment variable and I *do* still want to use the null transport. Except... the removed note *did* just remind me about a syntax change in the null transport for Symfony 4.4.

Hit "n" to *not* add this change... for now. Then hit "y" for the symfony.lock update.

[The Updated Null Mailer Transport Syntax](#)

Let's see how things look:

```
$ git status
```

Undo the changes:

```
$ git checkout .env
```

Open .env in our editor... and find the "mailer" section:

62 lines | .env

... lines 1 - 37

```
38 ###> symfony/mailer ###
39 MAILER_DSN=smtp://null
40 # in Symfony 4.4 and higher, the syntax is
41 # MAILER_DSN=null://default
42 ###
```

... lines 43 - 62

Even though we didn't accept the new recipe changes, we *do* need to update our syntax. Copy the example and paste. Actually, the default part can be anything - you'll sometimes see null:

61 lines | .env

... lines 1 - 37

```
38 ###> symfony/mailer ###
39 MAILER_DSN=null://null
40 # in Symfony 4.4 and higher, the syntax is
41 # MAILER_DSN=null://default
42 ###
```

... lines 43 - 61

And *now* if you wanted to delete the extra comments about Symfony 4.4, you totally could... and probably should.

So... we basically didn't use *anything* from the updated recipe, but it *did* remind us of a change we needed to make.

Checking the CHANGELOG

And because symfony/mailer may have *other* backwards-compatibility breaks, it's not a bad idea to check its CHANGELOG. I'll go to <https://github.com/symfony/mailer>... and click to see it. Yep! You can see info about the null change and a few others. We'll see one of these later.

Back at your terminal, run:

● ● ●

```
$ composer recipes
```

again. There's *one* other recipe that's relevant to symfony/mailer. It's symfony/sendgrid-mailer: a package that helps us send emails through SendGrid. Let's skip straight to updating this:

● ● ●

```
$ composer recipes:install symfony/sendgrid-mailer --force -v
```

And then step through the changes with:

● ● ●

```
$ git add -p
```


The first change is inside .env. Oh! Ha! That's the change we made, I forgot to add it. Hit "y" to add it now.

The *other* change is *also* in .env: it changed the MAILER_DSN example from something starting with smtp:// to sendgrid://. Similar to the null transport situation, symfony/mailer 4.4 *also* changed the syntax for a few *other* transports.

I'm going to say "y" to accept this change: both the old and new code were just examples anyway.


But, there is one *other* spot you need to check: we need to see if we're using the old format in the .env.local file. Go open that up. In this project, nope! I'm not overriding that. If we *did* have smtp://sendgrid in *any* env files, or configured as a *real* environment variable, maybe on production, that would need to be updated.

For the last change - to symfony.lock - hit "y" to add it. Run:



```
$ git status
```

to make sure we're not missing anything. Looks good! Commit!



```
$ git commit -m "updating symfony/mailer recipe packages"
```

Done! We're down to the *last* few recipe updates. Let's *crush* them.

Chapter 11: phpunit-bridge & routing Recipes

Let's see what recipes we have left:

```
$ composer recipes
```

[Updating symfony/phpunit-bridge](#)

Next up is phpunit-bridge. Copy its name and run:

```
$ composer recipes:install symfony/phpunit-bridge --force -v
```

It says it created - probably *updated* - 5 files. You know the process:

```
$ git add -p
```

[Updating .env.test](#)

The first is `.env.test` with two changes. This fix is a silly typo on the `APP_SECRET` variable - that's not important - and the second is a new `PANTHER_APP_ENV` variable. If you're using Symfony Panther - a cool testing tool - then this variable is used to tell Panther which environment to launch. If you're not, then you don't technically need this... but it doesn't hurt anything. You also might install Panther in the future.

As a challenge, let's see if we can *just* add the *second* change. To do that, type "s", which means "split". The tool will then *try* to ask you about each change independently. Say "n" to the first and "y" to the second.

[.phpunit.result.cache inside .gitignore](#)

The next update is inside `.gitignore`: it ignores a new `.phpunit.result.cache` file. This is why I *love* updating recipes. Since version 7.3, PHPUnit outputs this file under certain conditions as a way to remember which tests failed. It should be ignored, and this takes care of that. Enter "y".

[Updating bin/phpunit](#)

Woh! The next change looks bigger - this is `bin/phpunit`: a script that this packages adds to help execute PHPUnit. It has a number of subtle updates... and since you've *almost* definitely not made any custom tweaks to this file, let's add the change.

[Updating phpunit.xml.dist](#)

The *last* update is for `phpunit.xml.dist` - PHPUnit's configuration file. You *may* have some customizations here that you want to keep - so be careful. The recipe updates are tiny: these changed from `<env` to `<server` - a mostly meaningless change to how environment variables are added - and it looks like something about these two `SYMFONY_PHPUNIT` variables got tweaked a bit.

Hit "y" to accept this patch. The last change is for `symfony.lock` - hit "y" for this one too.

[Updating symfony/routing](#)

Done! The next recipe on the list is `symfony/routing`. Let's jump straight to update it:

```
$ composer recipes:install symfony/routing --force -v
```

And then get into:

```
$ git add -p
```

Bah! Duh! I should have committed my changes *before* starting this and then reverted the stuff we did *not* want - like this. We'll do that in a minute. Hit "n" to ignore the .env.test change.

The first *real* change is in config/packages/routing.yaml: strict_requirements is gone and utf8: true was added. If you dug into the recipe history, you could find the reason behind both of these. utf8 is a new feature in routing. By setting this to true, you're activating that feature. We may not need it, but I'm going to say yes to this.

The second change - strict_requirements - is thanks to a little reorganization of the routing config files that, actually, I am responsible for. The short story is that: you want this key to be set to a different value in different environments. I moved some config around to get that done with less files.

Hit "y" to add these changes. And... yep! This is symfony.lock, so accept this too. Phew! Let's see how things look:

```
$ git status
```

Woh! A new routing.yaml file for the prod environment! If you open that - config/packages/prod/routing.yaml - it has strict_requirements: null:

```
4 lines | config/packages/prod/routing.yaml
```

```
1 framework:
2   router:
3     strict_requirements: null
```

It's part of that reorganization I was just talking about. Add that change:

```
$ git add config/packages/prod/routing.yaml
```

The *last* change - which we need to do manually - is to delete config/packages/test/routing.yaml:

```
4 lines | config/packages/test/routing.yaml
```

```
1 framework:
2   router:
3     strict_requirements: true
```

It's *another* file with strict_requirements and it is *gone* from the new recipe. Why? It's just not needed anymore: if you followed the logic, you'd find that strict_requirements is already true in the test environment. Delete it:

```
$ git rm config/packages/test/routing.yaml
```


Oh... I apparently modified that file? Whoops! Yep, I added a "g"! Not helpful. Remove it and... delete the file:

```
$ git rm config/packages/test/routing.yaml
```

Let's see how things look:


```
$ git status
```

A *lot* of progress from those two recipe updates. Let's commit:




```
$ git commit -m "upgrading phpunit & routing recipes"
```

The *one* change left - that we decided we *didn't* care about - is in `.env.test`. Revert it with:



```
$ git checkout .env.test
```

Woo! Let's find out what recipes we have left:



```
$ composer recipes
```

Woh! Only 3 main Symfony repositories left: security-bundle, translation and validator. Let's do those next.

Chapter 12: Updating security, translation & validator Recipes

The composer recipes command tells us that we only have *three* more main Symfony recipes to update. Let's get to it! The next one is for security-bundle. Update it:

[Updating symfony/security-bundle Recipe](#)

```
$ composer recipes:install symfony/security-bundle --force -v
```

And then run:

```
$ git add -p
```

Woh! It looks like it made a *lot* of changes! But... like we've learned, what we're *really* seeing is it *replacing* all of our custom logic with the updated file from the recipe.

And... we want to keep pretty much *all* of our stuff: our custom encoder, user provider and firewall config. Let's look closely to see if there's anything interesting in the *new* code. Oh, there is *one* change: anonymous: true was changed to anonymous: lazy.

This is a new feature from Symfony 4.4. It basically means that, instead of Symfony figuring out *who* you're logged in as at the beginning of each request, it will do it *lazily*: it will *wait* until the moment your code tries to ask "who" is logged in. If your code *never* asks, then the authentication logic *never* runs. This was done to help make HTTP caching easier for pages that don't need any user info.

So, we *do* want this change. Hit "q" to exit this and... revert the changes with:

```
$ git checkout config/packages/security.yaml
```

Now, open that file in your editor, find anonymous and change it to lazy:

```
61 lines | config/packages/security.yaml
1  security:
    ... lines 2 - 16
17  firewalls:
    ... lines 18 - 20
21  main:
22      anonymous: lazy
    ... lines 23 - 61
```

Let's keep going:

```
$ git add -p
```

This time, say "y" to add the change... and "y" again for symfony.lock. Let's commit!

```
$ git commit -m "upgrading security recipe"
```

Done!

[Upgrading the symfony/translation Recipe](#)

What's next? Let's find out:

```
$ composer recipes
```

Ah, translation! Update it:

```
$ composer recipes:install symfony/translation --force -v
```

And walk through the changes:

```
$ git add -p
```

In translation.yaml, all the %locale% parameters were replaced with just en. The locale parameter is set in our config/services.yaml file:

```
51 lines | config/services.yaml
```

```
... lines 1 - 5
```

```
6 parameters:
```

```
... line 7
```

```
8 locale: 'en'
```

```
... lines 9 - 51
```

This was *originally* added by a recipe.

So... what's going on here? *Purely* for simplification, instead of setting that parameter and then using it in this *one* file, the recipe was updated to remove the parameter and set the locale directly. You don't need to make this change if you don't want to.

But I'll say "y" and then "y" again for the symfony.lock file. Back in services.yaml, manually remove the locale parameter:

```
51 lines | config/services.yaml
```

```
... lines 1 - 5
```

```
6 parameters:
```

```
... line 7
```

```
8 locale: 'en'
```

```
... lines 9 - 51
```

Why didn't the recipe remove that for me? Well, again, *removing* things - like old files or even old parameters - is not something the recipe update system *currently* handles.

Run:

```
$ git status
```

Then:

```
$ git add -p
```

And accept this *one* change. Commit!

```
$ git commit -m "updating translation recipe"
```

[Updating the symfony/validator recipe](#)

We're on a roll!

```
$ composer recipes
```

Oh, *so* close. Next is the validator recipe. Update it:

```
$ composer recipes:install symfony/validator --force -v
```

And walk through the changes:

```
$ git add -p
```

The first change is in config/packages/validator.yaml: it adds some new config that's commented out. This activates a new validation feature called auto-mapping. It's *really* cool - and we're going to talk about it later. Hit "y" to add these comments and... yep! This is the symfony.lock file. Press "y" again.

That was easy! Let's commit. Actually, I *should* have run git status, because this recipe *also* added a *new* file. We'll see it in a minute:

```
$ git commit -m "updating validator recipe"
```

[Updating webpack-encore-bundle Recipe](#)

Are we done? Run:

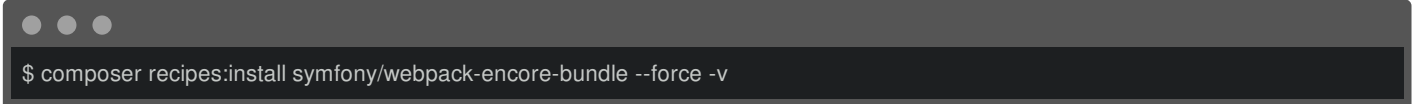
```
$ composer recipes
```

We *are*! Well, there is *one* more that starts with symfony/: webpack-encore-bundle. But that bundle *isn't* part of the main Symfony repository... so you can update it now or later. If you're interested, let's update it next. If you're not, skip ahead one chapter to start finding and fixing deprecations.

Chapter 13: Updating the webpack-encore-bundle Recipe

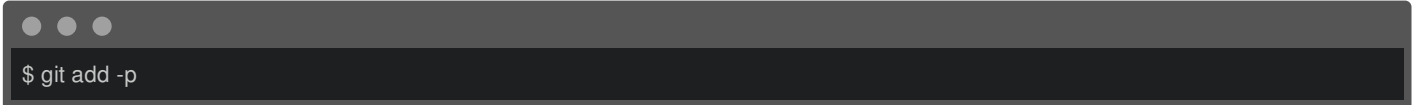
Our goal was to upgrade all of the recipes for the main Symfony packages. And... we've done it! Victory! The *last* one that starts with symfony/ is *not* part of the main repository... so if the goal is to upgrade to Symfony 5... we don't really need to do this now. But if our goal is to be an over-achiever and earn extra credit... well... then we should *crush* this last Symfony recipe update.

Let's do it:



```
$ composer recipes:install symfony/webpack-encore-bundle --force -v
```

Start checking out the updates with:



```
$ git add -p
```

Change one: it added `/public/build/` to the `.gitignore` file. We *definitely* want that... I'm not sure why it was missing. Next is `assets/js/app.js`. There are a *lot* of changes here... but we don't want *any* of them. The WebpackEncoreBundle recipe gives you an "example" `app.js` file to start with. We don't want that example to overwrite our custom code.

The next change is a missing line at the end of the file - that's meaningless - and then... let's see... this is `config/packages/webpack_encore.yaml`. It didn't actually *change* anything... it just added a lot more comments. Let's hit "y" to add it - comments are nice.

Next is `package.json`. The recipe gives us a *starting* `package.json` file. But *we* want our custom code - so hit "n". Hit "n" again to *also* keep our custom `browserslist` config at the bottom.

[Updating webpack.config.js](#)

The next file is `symfony.lock` - hit "y" to accept - and the *last* is `webpack.config.js`. This is *another* file that *we* customize. So we definitely do *not* want to accept *everything*. But, there *may* be some nice new suggestions.


This first new code looks like it helps out with some edge-case... let's accept this. But the next overwrites all of our custom entries. Definitely hit "n".

The third change adds a commented-out example of `disableSingleRuntimeChunk()` - I don't need that - and then... woh! The last "chunk" contains a *bunch* of stuff. I'll clear the screen and hit "s" to "split" this big change into smaller pieces.

Much better! The first relates to configuring Babel. You *should* now have some config that looks like this in your `webpack.config.js` but I won't go into the details *why* right now. Both the old and new code are effectively identical... but the new version is recommended, so hit "y" to add it.

Next, we don't want to change any of our sass-loader stuff... say no to that. This changes some commented-out example code - might as well say "y". And we *are* using `autoProvidejQuery()`, so keep that. Finally, we're apparently missing a new line at the end of the file - that's meaningless, but I'll hit "y".

Phew! Run:



```
$ git status
```

Oh! And there are *three* new files too!

The first - `app.css` - is an example CSS file that the Encore recipe adds. We're not using it in our app - so we don't need it. Delete it!

```
$ rm assets/css/app.css
```

The next new file - `config/packages/test/validator.yaml` is one I missed earlier from the validator recipe. Let's check it out, it's super minor:

```
4 lines | config/packages/test/validator.yaml
1  framework:
2    validation:
3      not_compromised_password: false
```

It *disables* a validator in the test environment that makes a network request and is a security-related feature that just *isn't* needed in your tests.

The last new file is in the same directory - `webpack_encore.yaml`:

```
3 lines | config/packages/test/webpack_encore.yaml
1  #webpack_encore:
2  #  strict_mode: false
```

Which... contains some commented-out example code. Let's add both of these new files:

```
$ git add config/packages/test
```

And see how things look:

```
$ git status
```

Perfect! Commit time!

```
$ git commit -m "updating webpack encore recipe + missing validator file"
```

We can revert *all* of the changes we don't want with:

```
$ git checkout .
```

Ah! I think we're done! Check out the recipes:

```
$ composer recipes
```

Gorgeous! All the symfony recipes are now up-to-date. I know that was a lot of work... but mostly because we were being extra careful and doing our research to find the *reason* a change was made.

The benefit is *huge*. Not only can we keep upgrading our app forever thanks to the smart way that Symfony handles new major versions, but by updating our recipes, we can make sure our app *truly* continues to look & act like all apps. Plus, we get to find out about new features and this gave us a head-start on fixing deprecations.

Now that we're using Symfony 4.4 with a set of up-to-date recipes, let's start finding and fixing the deprecations in our app. That's the *last* step before going to Symfony 5.

Chapter 14: Fixing the First Deprecations

We upgraded our app to Symfony 4.4 *and* updated all of its Symfony recipes. We *rock*!

And *now*, our path to Symfony 5 is clear: we need to find and fix all deprecated code that we're using. As *soon* as we've done that, it will be safe to tweak our composer.json file, go to 5.0 and celebrate with cheesecake.

[Finding Deprecated Code](#)

So how *do* we figure out what deprecated functions, config options or classes we might be using? There are two *main* ways. The best is down here on the web debug toolbar. This *literally* tells us that during this page load, we called 49 deprecated things. We'll look at these in a minute and start to eliminate them.

But *even* once you get this number down to zero... you can't *really* be sure that you've fixed *all* your deprecated code. Like, what if you're using a deprecated function *only* on some obscure AJAX call... and you forgot to check that.

That's why Symfony has a *second* way to find this stuff: a deprecations log file on production. Open up config/packages/prod/monolog.yaml. This has two handlers - deprecation and deprecation_filter - that, together, add a log entry each time your app uses deprecated code on production:

```
24 lines | config/packages/prod/monolog.yaml
1  monolog:
2    handlers:
3    ... lines 3 - 15
16  deprecation:
17    type: stream
18    path: "%kernel.logs_dir%/%kernel.environment%.deprecations.log"
19  deprecation_filter:
20    type: filter
21    handler: deprecation
22    max_level: info
23    channels: ["php"]
```

So once you *think* you've fixed all your deprecated code, deploy it to production, wait a few hours or days, and check the log to make sure it doesn't contain anything new. *Then* you'll *know* it's safe to upgrade.

By the way, Symfony Insight has a special feature to identify and fix deprecation warnings. So if you want some extra help... or an "easy" button, give it a try.

[Removing WebServerBundle](#)

Let's start crushing our deprecations. I'll refresh the homepage and open the deprecations link in a new window. Fixing deprecated code can be... well... an adventure! Because you might need to change a class name, method name, remove a bundle, upgrade a 3rd party library, tweak some config or do a secret handshake. Every deprecation is a little different.

But the first one is simple: it says that the WebServerBundle is deprecated since Symfony 4.4.

At the beginning of this tutorial, we started a local web server by running:

```
$ symfony serve
```

This symfony thing is the Symfony binary: a nice development tool that, in addition to other tricks, is able to start a development server. Before this existed, we *used* to start a local web server by running:

```
$ php bin/console server:run
```

A console command that comes from WebServerBundle. That's now deprecated... because the Symfony binary is shinier and more powerful.

So this deprecation is easy to fix. Inside `composer.json`, find the `symfony/web-server-bundle` line:

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4  "require": {
  ... lines 5 - 37
38  "symfony/web-server-bundle": "4.4.*",
  ... lines 39 - 43
44  },
  ... lines 45 - 102
103 }
```

Copy it, go to your terminal and remove it:

```
$ composer remove symfony/web-server-bundle
```

Oh! It's over-achieving! In addition to removing the bundle, it's upgrading a few related packages to the latest bug fix version. This also - importantly - *unconfigured* the recipe: it removed the bundle from `bundles.php` *and* deleted a line from `.gitignore` that we don't need anymore.

Updating a 3rd Party Bundle

Hey! One deprecation is gone! Let's go find another one! Hmm, the second is something about:

Calling the `EventDispatcher::dispatch()` method with the event name as the first argument is deprecated since Symfony 4.3.

One of the trickiest things about fixing deprecations is that you need to find out *where* this is coming from. To make things even *more*, let's say, "interesting", pretty often, a deprecation won't be triggered directly by *our* code, but by a third-party bundle that were using.


If you show the trace, it gives info about where this is coming from. It's not super obvious... but if you look closely up here, it mentions `LiipImagineBundle`.

Ok, so `LiipImagineBundle` *appears* to be calling some deprecated code, which means that our current version will *definitely* not be compatible with Symfony 5. Fortunately, there's *one* clear way to fix deprecated code that's called from a vendor library: upgrade it!

Let's do this in the *laziest* way possible. Inside `composer.json`, find that package:

```
104 lines | composer.json
1  {
  ... lines 2 - 3
4  "require": {
  ... lines 5 - 15
16  "liip/imagine-bundle": "^2.1",
  ... lines 17 - 43
44  },
  ... lines 45 - 102
103 }
```

Copy its name, and run:

A terminal window with a dark background and light gray text. The window has three small circular window control buttons (red, yellow, green) in the top-left corner. The command being entered is "\$ composer update liip/image-bundle".

```
$ composer update liip/image-bundle
```

What we're *hoping* is that this deprecation *has* been fixed and - *ideally* - that we only need to upgrade a "minor" version to get that fix, like maybe upgrading from 2.1 to 2.2 or 2.3.

And actually... yea! It *did* upgrade from 2.1 to 2.2. Did that fix the deprecation? I have no idea! Let's find out! Close the profiler tab and refresh the homepage. Good sign: the deprecations went from 48 to 29. I'll open the deprecations in a new tab and... awesome: it *does* look like that specific deprecation is gone.

Let's keep going! We're going to focus on these TreeBuilder::root() deprecations next. These are *a/so* coming from third-party libraries. But upgrading them will be a bit more complex.

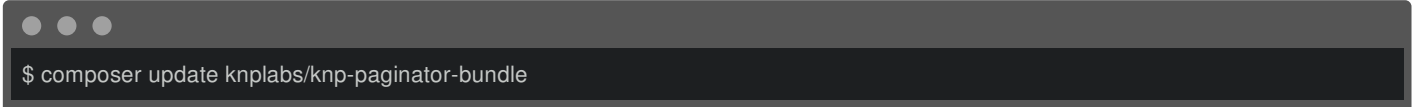
Chapter 15: Upgrading KnpPaginatorBundle & PHP Platform Version

If you look at the list of deprecations, a *bunch* of these mention the same problem: some `TreeBuilder::root()` thing. This is a low-level function that third-party bundles use. And if you dig through the list, this `stof_doctrine_extensions` comes from `StofDoctrineExtensionsBundle`... as does most of the other ones - like `orm`, and `mongodb`. The last one comes from `KnpPaginatorBundle`.

So basically... we need to upgrade *both* `KnpPaginatorBundle` and `StofDoctrineExtensionsBundle`.

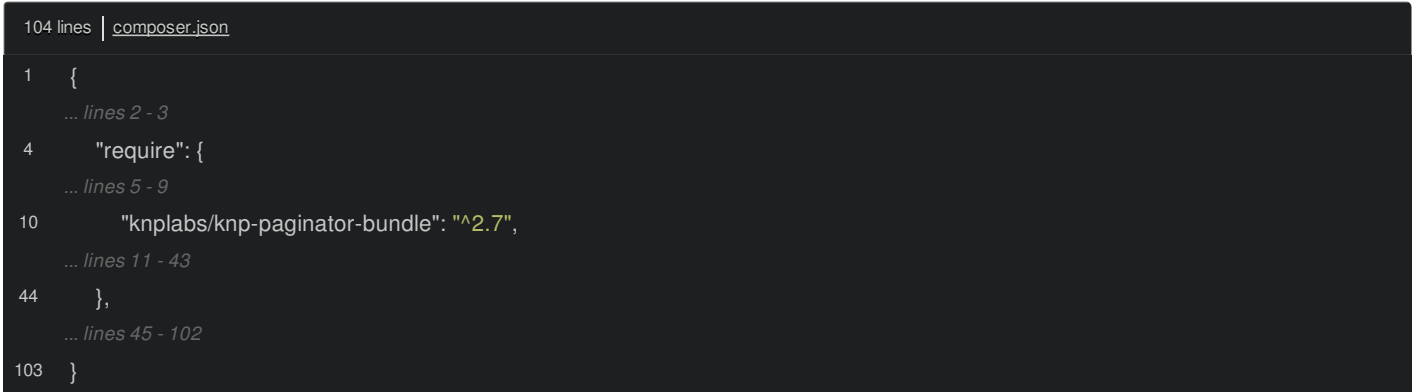
[Upgrading KnpPaginatorBundle the Lazy Way](#)

Let's start with `KnpPaginatorBundle`... and try to be as *lazy* as possible. Copy the package name, move over, and run:



```
$ composer update knplabs/knp-paginator-bundle
```

My *hope* is that a minor upgrade - something like 2.8 to 2.9, which my `composer.json` version constraint allows - will be enough to fix the deprecation:



```
104 lines | composer.json
1  {
  ... lines 2 - 3
4    "require": {
  ... lines 5 - 9
10      "knplabs/knp-paginator-bundle": "^2.7",
  ... lines 11 - 43
44    },
  ... lines 45 - 102
103 }
```

And... absolutely *nothing* happens. It didn't upgrade the library at *all*! Boo.

[Digging into a Library's Releases](#)

So much for the lazy way out: *now* we need to do some digging. Google for the bundle and find their GitHub page. PhpStorm tells me that I *currently* have version 2.8.0. Back on the GitHub page, click on "Releases".

Woh! The latest version is 5.0! And it says:

Added support for Symfony 5

That's what we want! So, to get a version of this library that works with Symfony 5, we need to upgrade to 5.0 of the bundle. Back in `composer.json`, change the version to `^5.0`:

```

103 lines | composer.json
1  {
    ... lines 2 - 3
4    "require": {
    ... lines 5 - 9
10      "knplabs/knp-paginator-bundle": "^5.0",
    ... lines 11 - 42
43    },
    ... lines 44 - 101
102  }

```

And yes, because we're upgrading to a new *major* version - heck, we're upgrading 3 new major versions - this *could* contain some backwards-incompatible changes that will break our app. Let's... worry about that in a little while.

Go update!

```

$ composer update knplabs/knp-paginator-bundle

```

Checking your config.platform.php Setting

And... this fails. Boo! Let's see: we tried to get version 5.0 of the bundle... but it requires PHP 7.2 or higher... and my PHP version is 7.3.6... but is overwritten by my config.platform.php version... which is 7.1.3.

Wow! That's a fancy way of saying that:

Version 5 of this library requires a higher version of PHP than I'm using

Except... well... that's not totally right. I'm using PHP 7.3, but in my composer.json file... if you search for config, here it is: I added a config.platform.php key set to 7.1.3:

```

103 lines | composer.json
1  {
    ... lines 2 - 53
54  "config": {
    ... lines 55 - 58
59    "platform": {
60      "php": "7.1.3"
61    }
62  },
    ... lines 63 - 101
102  }

```

This is an optional setting and you might not have this in your app... but I *do* recommend adding it. It tells Composer to *pretend* like I'm using PHP 7.1.3 when downloading dependencies *even* though I'm *actually* using PHP 7.3.

Why would I want that? By setting this value to whatever PHP version you have on production, it will make sure you don't accidentally download any packages that work on your local machine with its *higher* PHP version... by explode on production.

So if our goal is to upgrade to Symfony 5, our production server will need to *at least* be set to Symfony 5's minimum PHP version, which is 7.2.5. And for consistency... even though it doesn't affect anything, under the require key, update this too:

```

103 lines | composer.json
1  {
    ... lines 2 - 3
4    "require": {
5        "php": "^7.2.5",
    ... lines 6 - 42
43    },
    ... lines 44 - 53
54    "config": {
    ... lines 55 - 58
59        "platform": {
60            "php": "7.2.5"
61        }
62    },
    ... lines 63 - 101
102 }

```

Updating --with-dependencies

Ok *now* that Composer knows it's ok to download packages that require PHP 7.2... let's try that update command again:

```
$ composer update knplabs/knp-paginator-bundle
```

And... yay! Another error! I mean, another *fascinating* challenge that we are *totally* up to beating. Hmm... KnpPaginatorBundle requires something called knp-components and... basically 5.0 of the bundle requires version 2 of knp-components, but we're currently "locked" at version 1.3, which just means that version 1.3 is what is installed in our app right now.

This knp-components library is *not* something that we have directly in our composer.json file: it's a "transitive" dependency, which is a hipster way of saying that our app only needs it because KnpPaginatorBundle needs it.

So then... why didn't Composer just update *both* libraries? Because Composer is conservative: we told it to *only* upgrade knplabs/knp-paginator-bundle and it correctly figured out that it can't *only* upgrade that *one* package.

To fix this, run the command again but now add --with-dependencies:

```
$ composer update knplabs/knp-paginator-bundle --with-dependencies
```

This says: it's ok to upgrade knp-paginator-bundle *and* also any of *its* dependencies. This time... it did the trick: this upgrades from version 1 to 2 of knplabs/knp-components and from version 2 to 5 of knplabs/knp-paginator-bundle.

Checking Major Upgrade Changelogs

Awesome! Except... we need to be careful: these are *major* version upgrades... which means that they might contain "breaking" changes.

Go back to the GitHub homepage for KnpPaginatorBundle and look for a CHANGELOG.md file. Not every library will have this... but most do. Let's see: the breaking changes for version 3 were just removing support for old PHP versions. For version 4... it dropped support for old PHP and old Symfony versions... and for version 5, it added a return type to PaginatorAwareInterface... which is not something I'm using in my app.

So... we're good! You could repeat this for the knp-components library if you want, though since we're not using its code *directly* in our app, we should be good.

Ok, we've handled this knp_paginator deprecation. Next, let's update StofDoctrineExtensionsBundle to remove the *rest* of those pesky "tree" deprecations.

Chapter 16: Upgrading/Migrating from StofDoctrineExtensions

Let's see how our deprecation todo list is looking: refresh the homepage, open the profiler and... we still have the `TreeBuilder::root()` deprecation coming from `stof_doctrine_extension's`.

You know the drill: try to upgrade this the *lazy* way: find the package name and copy it:

```
103 lines | composer.json
1  {
    ... lines 2 - 3
4    "require": {
    ... lines 5 - 20
21    "stof/doctrine-extensions-bundle": "^1.3",
    ... lines 22 - 42
43  },
    ... lines 44 - 101
102 }
```

We're *hoping* a minor upgrade - maybe from 1.3 to 1.4 - will fix things. Update!

```
$ composer update stof/doctrine-extensions-bundle
```

And... once again... *nothing* happens.

Tip

Symfony 5 support was added in release v1.4.0 of `stof/doctrine-extensions-bundle` so you can continue using this package and skip installing `antishov/doctrine-extensions-bundle` fork as we do further.

Let's go hunting for answers! Google for `StofDoctrineExtensionsBundle` and... find its GitHub page. The *first* question I have is: what is the latest version? It's, oh: 1.3.0 - that's the version *we're* using... and it's 2 years old!

This is an example of a bundle that, at least at the time of this recording, does *not* yet support Symfony 5. So... what do we do? Panic! Ahhhh.

Now that we've accomplished that, I recommend looking at the package's issues and pull requests. *Hopefully* you will find some conversation about Symfony 5 support and, *hopefully*, it's something that's coming soon or you can help with.

But... in this case, as much as I like this bundle, you'll find that it's basically abandoned.

Hello fork: [antishov/doctrine-extensions-bundle](#)

That *does* happen sometimes. After all, most open source maintainers are volunteers. *However*, that digging into the pull requests would *also* reveal that someone in the community has done a *really* nice job of forking this library and creating some new releases.

Copy the library name, Google for it and... let's see... here is its [GitHub page](#). Click to view the releases.

Basically, someone forked the library, kept *all* the code and release history, but started fixing things and creating new releases... including a release that adds Symfony 5 support. We're saved!

Changing to [antishov/doctrine-extensions-bundle](#)

So let's switch to use this fork. Copy the stof package name again, and remove it:

```
$ composer remove stof/doctrine-extensions-bundle
```

Composer removes it and then... explodes! That's ok: it *was* removed... but because our app, *needs* this library... it's temporarily not speaking to us.

Now go back to the homepage of the fork, find the composer require line, copy it, and re-install the library:

```
$ composer require antishov/doctrine-extensions-bundle
```

This *basically* gives us the same library but at a newer version. The author *also* created an identical recipe for this package, so even the recipe gets re-installed nicely.

Commit the files we *know* we want to keep:

```
$ git add composer.json composer.lock symfony.lock
```

Now selectively-choose the changes from the updated recipe by running:

```
$ git add -p
```

For bundles.php - it *looks* like it removed the bundle... but if you hit "y", it just moved it. A meaningless change. And next, because it re-installed the recipe, it removed our custom changes. Hit "n" to skip those.

Let's commit!

```
$ git commit -m "using doctrine extensions bundle fork"
```

And then, revert the changes to the config file:

```
$ git checkout .
```

So... that update was weird. Let's close some tabs and refresh. *Yas!* The deprecations jumped from 25 to 16.

We're killing it! The next deprecations are going to uncover that we *also* need to upgrade DoctrineBundle... from version 1 to 2 - a significant jump.

Chapter 17: Upgrading to DoctrineBundle 2.0

Let's look at the latest list of deprecated code. Hmm... there's a lot of stuff related to Doctrine. Ok: two tricky things are happening in the Doctrine world right now that make upgrading to Symfony 5... a *bit* more confusing. First, DoctrineBundle has a new *major* version - 2.0. And second, Doctrine itself is being split into smaller packages. Both things are great... but it means there are some extra "moving pieces".

[DoctrineBundle & Symfony 5 Compatibility](#)

If you search this page for DoctrineBundle, there's one deprecation: some missing `getMetadataDriverClass()` method in a `DoctrineExtension` class. So far, this is nothing new: a third-party library is using some deprecated code... which means that we need to upgrade it.

Google for DoctrineBundle and find its GitHub page. If you did some digging, you'd learn that if you want Symfony 5 support, you need version 2 or higher of this bundle. There's also a version 1.12 that's being maintained... but it won't work with Symfony 5.

[Debugging the DoctrineBundle Version](#)

So let's start with our normal, lazy way of upgrading. In your terminal, run:

```
$ composer update doctrine/doctrine-bundle
```

It *does* upgrade... but only to 1.12.6. So *probably* we need to go into `composer.json` and update its version constraint. Search for `doctrine/doctrine-bundle`. Huh... it's actually *not* inside our `composer.json` file. It must be a dependency of something else. Let's find out more. Run:

```
$ composer why doctrine/doctrine-bundle
```

Ok: `doctrine-bundle` is required by both `fixtures-bundle` and `migrations-bundle`. But the *original* reason that it was installed was because of `symfony/orm-pack`, which allows version 1 *or* 2. Remember: `orm-pack` is a sort of "fake" library that requires *other* libraries. It gives us an easy way to install Doctrine and some other Doctrine-related libraries.

I want to have *more* control over `doctrine/doctrine-bundle` so that I can *force* version 2 to be used, instead of just 1 *or* 2. To do that, we can "unpack" the pack. Run:

```
$ composer unpack symfony/orm-pack
```

Very simply, this removes `symfony/orm-pack` from our `composer.json` file:

```
103 lines | composer.json
1  {
    ... lines 2 - 3
4  "require": {
    ... lines 5 - 30
31  "symfony/orm-pack": "^1.0",
    ... lines 32 - 42
43  },
    ... lines 44 - 101
102 }
```

And *replaces* it with the libraries that it requires:

```

105 lines | composer.json
1  {
    ... lines 2 - 3
4    "require": {
    ... lines 5 - 8
9      "doctrine/doctrine-bundle": "^1.6.10|^2.0",
10     "doctrine/doctrine-migrations-bundle": "^1.3|^2.0",
11     "doctrine/orm": "^2.5.11",
    ... lines 12 - 44
45   },
    ... lines 46 - 103
104  }

```

Change the doctrine-bundle version to *just* ^2.0:

```

105 lines | composer.json
1  {
    ... lines 2 - 3
4    "require": {
    ... lines 5 - 8
9      "doctrine/doctrine-bundle": "^2.0",
    ... lines 10 - 44
45   },
    ... lines 46 - 103
104  }

```

Handling Composer Update Problems

Now that we're *forcing* version 2, let's... see if it will update! Run:

```

$ composer update doctrine/doctrine-bundle

```

And... this does *not* work. I should've seen this coming. doctrine-bundle can't be updated to version 2 because our project currently has doctrine-fixtures-bundle 3.2, which requires doctrine-bundle 1.6. So apparently we *also* need to update doctrine-fixtures-bundle. Ok! Copy the library name and say:

```

$ composer update doctrine/doctrine-bundle doctrine/doctrine-fixtures-bundle

```

We *may* need to change that bundle's version constraints to allow *it* to upgrade to a new major version:

```

105 lines | composer.json
1  {
    ... lines 2 - 45
46  "require-dev": {
47    "doctrine/doctrine-fixtures-bundle": "^3.0",
    ... lines 48 - 54
55  },
    ... lines 56 - 103
104  }

```

I honestly don't really know. My hope is that a newer 3.something version will allow doctrine-bundle 2.0. But when we check on Composer... bah! It didn't work! But *this* time because of a *different* library: doctrine-migrations-bundle. That *also* needs to be updated. No problem: copy its name and add it to the end of our composer update line:

```
$ composer update doctrine/doctrine-bundle \  
$ doctrine/doctrine-fixtures-bundle \  
$ doctrine/doctrine-migrations-bundle
```

We're now allowing doctrine-bundle, doctrine-fixtures-bundle *and* doctrine-migrations-bundle all to update and... it *still* doesn't work. Sheesh. Let's see: this time it's because doctrine-migrations-bundle requires doctrine/migrations 2.2 and apparently we're locked at a lower version of that. Yea we have 2.1.1.

It's the *same* problem... again. Well, it's *slightly* different. We *could* add doctrine/migrations to the end of our composer update command - or even use doctrine/* - and try it again. That should work.

Or we can add --with-dependencies. This says: allow any of these three bundles to update *and* allow their dependencies to update. doctrine/migrations is *not* in our composer.json file: it's a dependency of doctrine-migrations-bundle. Oh, and if you *really* want the "easy" way out, we could have just ran composer update with no arguments to allow *anything* to update. But I prefer to update as *little* as possible at one time. Try the command:

```
$ composer update doctrine/doctrine-bundle \  
$ doctrine/doctrine-fixtures-bundle \  
$ doctrine/doctrine-migrations-bundle \  
$ --with-dependencies
```

And... it worked! Yeeeee! Then... exploded at the bottom. We'll talk about that in a minute. First, look back up: it upgraded doctrine-migrations-bundle and doctrine-fixtures-bundle both to new *minor* versions. So there *shouldn't* be any breaking changes in those.

The doctrine-bundle upgrade *was* over a major version - from 1 to 2 - so it shouldn't be a *huge* surprise that it made our code go bonkers: the new version *does* have some breaking changes.

[Exit: DoctrineCacheBundle](#)

One other thing I want to point out is that doctrine-cache-bundle was removed. That's no longer needed by doctrine and you shouldn't use it anymore either: use Symfony's cache.

Next, let's fix our app to work with DoctrineBundle 2.0 *and* update its recipe, which contains a few important config changes.

Chapter 18: DoctrineBundle Updates & Recipe Upgrade

We just upgraded from DoctrineBundle version 1 to version 2 and... it broke our app. That's lame! Hmm:

Cannot autowire service ApiTokenRepository: argument \$registry references interface RegistryInterface but no such service exists.

[Checking the CHANGELOG](#)

Hmm. Ya know, instead of trying to figure this out... and digging for any other breaking changes... let's just go look at the bundle's CHANGELOG. Go back to the DoctrineBundle GitHub homepage. And... ah! Even better: upgrade files! Open UPGRADE-2.0.md.

There's a lot here: dropping old versions of PHP & Symfony and changes to commands. But if you look closely, you'll find that *most* of these are pretty minor. The *most* important changes are under "Services aliases". Previously, if you wanted to get the doctrine service, you could use the RegistryInterface type-hint for autowiring. Now you should use ManagerRegistry.

[From RegistryInterface to ManagerRegistry in Repository Classes](#)

Where do we use RegistryInterface? Move over to your terminal and run:

```
$ git grep RegistryInterface
```

We use it in *every* single repository class. This is code that the make:entity command generated *for* us. The newest version of that bundle uses ManagerRegistry.

Fixing this is as simple as changing a type-hint... it's just tedious. Open up *every* repository class. And, one-by-one, I'll change RegistryInterface to ManagerRegistry in the constructor:

```
87 lines | src/Repository/UserRepository.php
... lines 1 - 14
15 class UserRepository extends ServiceEntityRepository
16 {
17     public function __construct(ManagerRegistry $registry)
18     {
... line 19
20     }
... lines 21 - 85
86 }
```

Use ManagerRegistry from Doctrine\Persistence. There is *also* one from Doctrine\Common\Persistence:

```
87 lines | src/Repository/UserRepository.php
... lines 1 - 6
7 use Doctrine\Persistence\ManagerRegistry;
... lines 8 - 14
15 class UserRepository extends ServiceEntityRepository
16 {
17     public function __construct(ManagerRegistry $registry)
18     {
... line 19
20     }
... lines 21 - 85
86 }
```

[doctrine/common Split into doctrine/persistence \(and other Packages\)](#)

That's another Doctrine change that's happening right now. Doctrine originally had a package called doctrine/common, which contained a lot of... well... "common" classes that other Doctrine libraries needed. Doctrine is now splitting doctrine/common into *smaller*, individual packages. Basically, the Persistence directory of doctrine/common is now its own package and you should use *its* classes: the old ones are deprecated.

What makes this a bit more confusing is that the UPGRADE log references the old class name. Like I said: there are a lot of moving pieces right now.

I'll also remove the old RegistryInterface use statement. Repeat this a *bunch* more times: change to ManagerRegistry, remove the use statement and keep going:

```
51 lines | src/Repository/TagRepository.php
... lines 1 - 6
7  use Doctrine\Persistence\ManagerRegistry;
... lines 8 - 14
15 class TagRepository extends ServiceEntityRepository
16 {
17     public function __construct(ManagerRegistry $registry)
18     {
... line 19
20     }
... lines 21 - 49
50 }
```

```
82 lines | src/Repository/CommentRepository.php
... lines 1 - 8
9  use Doctrine\Persistence\ManagerRegistry;
... lines 10 - 17
18 class CommentRepository extends ServiceEntityRepository
19 {
20     public function __construct(ManagerRegistry $registry)
21     {
... line 22
23     }
... lines 24 - 80
81 }
```

Do you want to see how fast I can type?

```
75 lines | src/Repository/ArticleRepository.php
... lines 1 - 8
9  use Doctrine\Persistence\ManagerRegistry;
... lines 10 - 16
17 class ArticleRepository extends ServiceEntityRepository
18 {
19     public function __construct(ManagerRegistry $registry)
20     {
... line 21
22     }
... lines 23 - 73
74 }
```

```

51 lines | src/Repository/ApiTokenRepository.php
... lines 1 - 6
7  use Doctrine\Persistence\ManagerRegistry;
... lines 8 - 14
15 class ApiTokenRepository extends ServiceEntityRepository
16 {
17     public function __construct(ManagerRegistry $registry)
18     {
... line 19
20     }
... lines 21 - 49
50 }

```

Suuuuuper faaaaaaaast. Ah! I sprained a finger.

Let's see if we're good! Spin over and just run:

```
$ php bin/console
```

Before those changes, running this would have caused an *explosion* - the same one that we saw after running composer update. So we are good: we're using a Symfony5-compatible version of DoctrineBundle.

[Upgrading the DoctrineBundle Recipe](#)

But... because this library is *so* important... and because we just did a *major* version upgrade, I also want to upgrade its recipe. Run:

```
$ composer recipes
```

Ok, yea, DoctrineBundle is one of the *few* recipes that still have an update available. Run:

```
$ composer recipes doctrine/doctrine-bundle
```

to get more info and copy the update command. Run it!

```
$ composer recipes:install doctrine/doctrine-bundle --force -v
```

Ok, it looks like this updated several files. Let's step through the changes. Clear the screen and run:

```
$ git add -p
```

[.env Changes and serverVersion](#)

The first changes are inside .env: it added a PostgreSQL example and, oh, this comment is important: it mentions that the serverVersion setting is *required* in this file or in config/packages/doctrine.yaml. That's actually not a new thing, but the new recipe now gives you a bit more info about this.

The setting tells Doctrine what *version* of your database you're using, like MySQL 5.7 or mariadb-10.2.12. Doctrine uses that to know which features are *supported* by your database.

The point is: this is something Doctrine *needs* to know and you can add that config inside your DATABASE_URL environment variable *or* in doctrine.yaml, which is what I prefer. I like to set this to my production database version and

commit it inside doctrine.yaml so that the project works the same on any machine.

So... I guess I want these new comment changes, except that I want to keep my *existing* DATABASE_URL. Copy it, hit "y" to accept the changes, then "q" to quit the patch mode.

Back in our editor... find .env, look for DATABASE_URL, and paste our original value:

```
62 lines | .env
... lines 1 - 22
23 ###> doctrine/doctrine-bundle ###
24 # Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using
25 # For an SQLite database, use: "sqlite:///kernel.project_dir%/var/data.db"
26 # For a PostgreSQL database, use: "postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&charset=utf8"
27 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
28 DATABASE_URL=mysql://root:@127.0.0.1:3306/the_spacebar
29 ###
... lines 30 - 62
```

Let's keep going!

```
$ git add -p
```

Accept the change we just made to .env. The next update is in composer.json, we definitely want this. Then... actually, hit "q". Let's add the files we *know* we want:

```
$ git add composer.json composer.lock symfony.lock src/Repository
```

Run:

```
$ git status
```

Much better! Back to:

```
$ git add -p
```

[Updates to doctrine.yaml](#)

In bundles.php, it removed DoctrineCacheBundle - that's a good change - and now we're inside of doctrine.yaml.

There are a *bunch* of interesting updates here. First, there *used* to be a parameter called env(DATABASE_URL). This was a workaround to prevent Doctrine from exploding in some edge cases. It's no longer needed. Progress!

Next, the driver setting isn't needed inside here because that part is *always* contained inside DATABASE_URL. It's just extra, so we can remove it. Oh, and server_version was just moved further down.

The recipe *also* removed these charset options, and that *is* interesting. If you use MySQL, these settings are needed to make sure that you can store unicode characters. Starting in DoctrineBundle 2.0, these values are no longer needed because they are the *default*. That's a nice cleanup.

Below, the server_version is *now* commented-out by default: you need to *choose* to put it in this file or inside your environment variable. I'll uncomment this in a minute.

Finally, this naming_strategy is a minor change: it controls how table and column names are generated from class and property names. The new setting handles situations when there is a number in the name. It's a good change... and the old setting is deprecated. However it *is* possible that this could cause Doctrine to try to rename some columns. You can run:

```
$ php bin/console doctrine:schema:update --dump-sql
```

after making this change to be sure. Enter "y" to accept *all* these changes, then "q" to quit. Find this file - config/packages/doctrine.yaml - uncomment server_version and adjust it to whatever you need for your app:

```
19 lines | config/packages/doctrine.yaml
1  doctrine:
2  dbal:
   ... lines 3 - 6
7  server_version: '5.7'
   ... lines 8 - 19
```

[Production doctrine.yaml Cache Changes](#)

Back to work!

```
$ git add -p
```

Enter "y" for our server_version change. The *last* big update is in config/packages/prod/doctrine.yaml. This file configures cache settings in the prod environment: this is stuff you *do* want. When we originally installed the bundle, the old recipe created several cache services down here under the services key... and then used them above for the different cache drivers.

Basically, in DoctrineBundle 2.0, these services are created for you. This means that the config can be *drastically* simplified. Say "y" to this change.

And... we're done! Phew! Commit this:

```
$ git commit -m "upgrading to DoctrineBundle 2.0"
```

And celebrate!

[The doctrine/persistence 1.3 Deprecations](#)

Let's go see how the deprecations look now. When I refresh the homepage... down to 11 deprecations! Check them out.

Huh: a lot of them are *still* from doctrine... they all mention a deprecation in doctrine/persistence 1.3. doctrine/persistence is one of the libraries that was extracted from doctrine/common.

Ok, but why are we getting all these deprecations? Where are they coming from?

I have 2 things to say about this. First, because this is a deprecation warning about a change in doctrine/persistence 2.0... and because we're focusing right now on upgrading to Symfony 5.0, this is *not* a deprecation we need to fix. We can save it for later.

Second, if you Google'd this deprecation, you'd find that it is *not* coming from our code: it's coming from Doctrine *itself*, specifically doctrine/orm.

There's currently a pull request open on doctrine/orm - [number 7953](#) - that fixes these. Basically, doctrine/orm is using some deprecated code from doctrine/persistence, but the fix hasn't been merged yet. The fix is targeted for version 2.8 of doctrine/orm. So hopefully when that's released in the future, you'll be able to update to it to remove this deprecation. But, as I said... it's not a problem right now: we can keep working through the Symfony-related deprecations and ignore these.

And... *that* list is getting pretty short! Time to finish them.

Chapter 19: Fixing our Deprecations: Form, Controller & Mailer

We are now *super* close to fixing *all* the deprecation warnings that block us from going to Symfony 5. Let's check out the current list for the homepage. There are *technically* 12 deprecations. But remember, we can ignore all the ones from doctrine/persistence because they're not related to Symfony.

[Form getExtendedTypes\(\) Deprecation](#)

With that in mind... if you look closely, there are really only *two* real deprecations left... and they look like the same thing: something about TextareaSizeExtension should implement a static getExtendedTypes() method.

This TextareaSizeExtension class is a "form type extension" that we built in an earlier tutorial. Let's go check it out: src/Form/TypeExtension/TextareaSizeExtension.php:

```
39 lines | src/Form/TypeExtension/TextareaSizeExtension.php
... lines 1 - 7
8  use Symfony\Component\Form\FormTypeExtensionInterface;
... lines 9 - 11
12 class TextareaSizeExtension implements FormTypeExtensionInterface
13 {
... lines 14 - 37
38 }
```

And... PhpStorm is *immediately* mad at us:

Class must be declared abstract or implement method getExtendedTypes().

This is the error you see when you have a class that implements an interface but is *missing* one of the methods that the interface requires. But in this case, that's not *technically* true. Hold command or control and click the interface to jump to that file.

In reality, there is *no* getExtendedTypes() method on this interface! It has getExtendedType() - that's the old, deprecated method - but no getExtendedTypes(). It's not actually on the interface, it's just *described* on top of the class in comments.

You're seeing Symfony's deprecation system in action. If Symfony suddenly added this new getExtendedTypes() method to the interface in Symfony 4.4, it would have broken our app when we upgraded. That would violate Symfony's backwards-compatibility promise... which basically says: we will *not* break your app on a minor upgrade.

Instead Symfony *describes* that you need this method and *warns* you to add it via the deprecation system. It *will* be added to the interface in Symfony 5.0. Our job is to add this new static getExtendedTypes() method that returns iterable.

We got this! At the bottom of our class, add public static function getExtendedTypes() with an iterable return type. Inside, return an array with the same class as the old method:

```

44 lines | src/Form/TypeExtension/TextareaSizeExtension.php
... lines 1 - 4
5 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
... lines 6 - 11
12 class TextareaSizeExtension implements FormTypeExtensionInterface
13 {
... lines 14 - 33
34     public static function getExtendedTypes(): iterable
35     {
36         return [TextareaType::class];
37     }
... lines 38 - 42
43 }

```

As soon as we do this, the old, `getExtendedType()` method won't be called anymore:

```

39 lines | src/Form/TypeExtension/TextareaSizeExtension.php
... lines 1 - 11
12 class TextareaSizeExtension implements FormTypeExtensionInterface
13 {
... lines 14 - 33
34     public function getExtendedType()
35     {
36         return TextareaType::class;
37     }
38 }

```

And it will be *gone* from the interface in Symfony 5.0. But we *do* need to keep it temporarily... because, again, for backwards compatibility, it *does* still exist on the interface in Symfony 4.4. If we removed it from our class, PHP would be super angry. I'll add a comment:

not used anymore - remove in 5.0

```

44 lines | src/Form/TypeExtension/TextareaSizeExtension.php
... lines 1 - 11
12 class TextareaSizeExtension implements FormTypeExtensionInterface
13 {
... lines 14 - 38
39     public function getExtendedType()
40     {
41         // not used anymore - remove in 5.0
42     }
43 }

```

Cool! Let's go close the profiler, refresh and open the new deprecations list. And... hey! Ignoring the doctrine/persistence stuff, our homepage is now *free* of deprecations!

Does that mean our app is ready for Symfony 5? Ah... not so fast: we still need to do a few more things to be *sure* that no deprecated code is hiding.

Clearing the Cache to Trigger Deprecations

For example, sometimes deprecations hide in the cache-building process. Find your terminal and run:

```

$ php bin/console cache:clear

```

This will force Symfony to rebuild its container, a process which *itself* can sometimes contain deprecation warnings. Refresh the homepage now: still 10 deprecation warnings but... oh! One of these is different!

CommentAdminController extends Controller that is deprecated, use AbstractController instead.

[Controller to AbstractController](#)

Let's go find this: src/Controller/CommentAdminController.php:

```
37 lines | src/Controller/CommentAdminController.php
... lines 1 - 9
10 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 11 - 14
15 class CommentAdminController extends Controller
16 {
... lines 17 - 35
36 }
```

Very simply: change extends Controller to extends AbstractController:

```
38 lines | src/Controller/CommentAdminController.php
... lines 1 - 7
8 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... lines 9 - 15
16 class CommentAdminController extends AbstractController
17 {
... lines 18 - 36
37 }
```

I'll also remove the old use statement:

```
38 lines | src/Controller/CommentAdminController.php
... lines 1 - 10
11 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 12 - 38
```

These two base-classes work *almost* the same. The only difference is that, once you use AbstractController, you can't use `$this->get()` or `$this->container->get()` to fetch services by their id.

[Mailer: NamedAddress to Address](#)

Ok! Another deprecation down and the homepage is *once* again not triggering any deprecated code. Let's surf around and see if we notice any other deprecations... how about the registration page: `SymfonyNerd@example.com`, any password, agree to the terms and... woh! That's not a deprecation... that's a huge error!

In theory, you should *never* get an error after a "minor" version upgrade - like Symfony 4.3 to 4.4. But this is coming from Symfony's Mime component, which is part of Mailer. And because Mailer was experimental until Symfony 4.4 there *were* some breaking changes from 4.3 to 4.4. We saw this one mentioned earlier when we looked at the Mailer CHANGELOG. Basically, NamedAddress is now called Address.

Where do we use NamedAddress? Let's find out! At your terminal, my favorite way to find out is to run:

```
$ git grep NamedAddress
```

It's used in SetFromListener, Mailer and MailerTest. Let's do some updating. I'll start with src/Service/Mailer.php: change the use statement from NamedAddress to Address, then search for NamedAddress and remove the Named part here and in one other place:

67 lines | [src/Service/Mailer.php](#)

```
... lines 1 - 8
9  use Symfony\Component\Mime\Address;
... lines 10 - 12
13  class Mailer
14  {
... lines 15 - 27
28  public function sendWelcomeMessage(User $user): TemplatedEmail
29  {
30      $email = (new TemplatedEmail())
31          ->to(new Address($user->getEmail(), $user->getFirstName()))
... lines 32 - 41
42  }
43
44  public function sendAuthorWeeklyReportMessage(User $author, array $articles): TemplatedEmail
45  {
... lines 46 - 51
52      $email = (new TemplatedEmail())
53          ->to(new Address($author->getEmail(), $author->getFirstName()))
... lines 54 - 64
65  }
66  }
```

Next is EventListener/SetFromListener. Make the same change on top... and below:

29 lines | [src/EventListener/SetFromListener.php](#)

```
... lines 1 - 6
7  use Symfony\Component\Mime\Address;
... lines 8 - 9
10  class SetFromListener implements EventSubscriberInterface
11  {
... lines 12 - 18
19  public function onMessage(MessageEvent $event)
20  {
... lines 21 - 25
26      $email->from(new Address('alienmailcarrier@example.com', 'The Space Bar'));
27  }
28  }
```

The last place is inside of tests/: Service/MailerTest. Let's see: remove Named from the use statement... and then it's used below in 2 places:

65 lines | [tests/Service/MailerTest.php](#)

```
... lines 1 - 11
12 use Symfony\Component\Mime\Address;
... lines 13 - 15
16 class MailerTest extends KernelTestCase
17 {
18     public function testSendWelcomeMessage()
19     {
... lines 20 - 36
37     /** @var Address[] $namedAddresses */
... line 38
39     $this->assertInstanceOf(Address::class, $namedAddresses[0]);
... lines 40 - 41
42     }
... lines 43 - 63
64 }
```

Got it! Let's try the registration page now: refresh and... validation error. Change to a new email, agree to the terms and... got it!

Ok, the deprecations are gone from the homepage and registration page at *least*. Are we done? How can we be sure?

Next, let's use a few more tricks to *really* be sure the deprecations are gone.

Chapter 20: Hunting the Final Deprecations

How can we be sure *all* our deprecated code is gone? The easiest way to catch *most* things is to surf around your site to see if you can trigger any other deprecation logs. Except... if a deprecation happens on a form submit where you redirect after... or if it happens on an AJAX call... you're not going to see those on the web debug toolbar.

Checking Deprecated Logs Locally

Fortunately, deprecations are also logged to a file. At your terminal, run:

```
$ tail -f var/log/dev.log
```

Symfony writes a *lot* of stuff to this log file... *including* any deprecation warnings: "User Deprecated". Hit Ctrl+C to exit the "tail" mode and run this again, but this time "pipe" it to grep Deprecated:

```
$ tail -f var/log/dev.log | grep Deprecated
```

We're now watching the log file for any lines that contain Deprecated. Unfortunately, because of that annoying doctrine/persistence stuff, it *does* contain extra noise. But it'll still work. You could filter that out by adding another `| grep -v persistence`.

Anyways, *now* we can try out the site: like clicking into an article... or doing anything else you can think of, like going to an admin section `/admin/comments`. Oh, duh - I'm not logged in as an admin. You get the point: use your site, then go back and check out the deprecations.

Yikes! I probably *should* have added that `| grep -v persistence` to remove all the noise. But if you look closely... yea... *every* single one of these is from doctrine/persistence!

So as *best* as we can tell, our site *is* deprecation free. But! There are a few more things to check to be sure.

Command Deprecations

For example, if you have some custom console commands, *they* might trigger some deprecated code. Open a new terminal tab and run:

```
$ php bin/console
```

This app has two custom console commands. Let's run this `article:stats` command... it just prints out a fake table:

```
$ php bin/console article:stats foo
```

It worked perfectly. But if you go back to the logs and look closely... ah! A *real* deprecation warning!

ArticleStatsCommand::execute() return value should always be of the type int since Symfony 4.4, NULL returned.

Interesting. Let's open that command: `src/Command/ArticleStatsCommand.php`:

```

51 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
14     protected static $defaultName = 'article:stats';
15
16     protected function configure()
17     {
18         ... lines 18 - 22
23     }
24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         ... lines 27 - 48
49     }
50 }

```

Since Symfony 4.4, the `execute()` method of every command *must* return an integer. At the bottom, return 0:

```

53 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
14     ... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         ... lines 27 - 49
50         return 0;
51     }
52 }

```

This ends up being the "exit code" that the command returns when you run it. Zero means successful and pretty much anything else - like 1 - means that the command *failed*.

Copy the return and open the other command class. At the bottom of `execute()`, return 0:

```

65 lines | src/Command/AuthorWeeklyReportSendCommand.php
... lines 1 - 14
15 class AuthorWeeklyReportSendCommand extends Command
16 {
17     ... lines 17 - 38
39     protected function execute(InputInterface $input, OutputInterface $output)
40     {
41         ... lines 41 - 61
62         return 0;
63     }
64 }

```

And... let's make sure that we don't have any other return statements earlier. Nope, it looks good.

[Production Deprecation Log](#)

So we've surfed the site, checked the logs and run some console commands. *Now* are we sure that all the deprecated code is gone? Maybe? There are 2 final tricks.

First, as I mentioned earlier, at this point, I would deploy my code to production and watch the `prod.deprecations.log` file for any new entries... ignoring any doctrine/persistence stuff:

24 lines | [config/packages/prod/monolog.yaml](#)

```
1  monolog:
2    handlers:
    ... lines 3 - 15
16   deprecation:
17     type: stream
18     path: "%kernel.logs_dir%/%kernel.environment%.deprecations.log"
    ... lines 19 - 24
```

If nothing new is added, it's almost definitely safe to upgrade.

[Deprecations in Tests](#)

Another easy trick is to... run your tests! You... *do* have tests, right? Run:



```
$ php bin/phpunit
```

For me, it looks like it needs to download PHPUnit... and then... cool! This *collects* all the deprecations that were hit inside our tests and prints them when it's done. There are a *lot* of doctrine/persistence things... but that's it! There are no Symfony deprecations.

I am *now* willing to say that our app is ready for Symfony 5.0. So... let's upgrade next! Thanks to all our hard work, upgrading to a new major version of Symfony is just a Composer trick.

Chapter 21: Upgrading to Symfony 5.0

We've done it! We fixed *all* the deprecations in our app... except for the doctrine/persistence stuff, which we don't need to worry about because we're not upgrading that library. That means... we are ready for Symfony5!

[Changing composer.json for Symfony 5.0](#)

How... do we actually upgrade? We already know: it's the *exact* same process we used to upgrade from 4.3 to 4.4.

Open up composer.json. Our goal is to update *all* of these symfony/ libraries to 5.0:

```
105 lines | composer.json
1  {
  ... lines 2 - 3
4  "require": {
  ... lines 5 - 24
25    "symfony/asset": "4.4.*",
26    "symfony/console": "4.4.*",
27    "symfony/dotenv": "4.4.*",
28    "symfony/flex": "^1.0",
29    "symfony/form": "4.4.*",
30    "symfony/framework-bundle": "4.4.*",
31    "symfony/mailer": "4.4.*",
32    "symfony/messenger": "4.4.*",
33    "symfony/monolog-bundle": "^3.5",
34    "symfony/security-bundle": "4.4.*",
35    "symfony/sendgrid-mailer": "4.4.*",
36    "symfony/serializer-pack": "^1.0",
37    "symfony/twig-bundle": "4.4.*",
38    "symfony/twig-pack": "^1.0",
39    "symfony/validator": "4.4.*",
40    "symfony/webpack-encore-bundle": "^1.4",
41    "symfony/yaml": "4.4.*",
  ... lines 42 - 44
45  },
46  "require-dev": {
  ... lines 47 - 48
49    "symfony/browser-kit": "4.4.*",
50    "symfony/debug-bundle": "4.4.*",
51    "symfony/maker-bundle": "^1.0",
52    "symfony/phpunit-bridge": "4.4.*",
53    "symfony/profiler-pack": "^1.0",
54    "symfony/var-dumper": "4.4.*",
55  },
  ... lines 56 - 103
104 }
```

Well, not quite *all* of them - a few are not part of the main Symfony library, like monolog-bundle. But basically, everything that has 4.4.* now needs to be 5.0.*.

We also need to update one more thing: the extra.symfony.require value:

105 lines | [composer.json](#)

```
1  {  
    ... lines 2 - 96  
97  "extra": {  
98  "symfony": {  
    ... lines 99 - 100  
101    "require": "4.4.*"  
102  }  
103 }  
104 }
```

This is primarily a performance optimization that helps Composer filter out extra Symfony versions when it's trying to resolve packages. This *also* needs to change to 5.0.*.

Let's... do it all at once: Find 4.4.*, replace it with 5.0.* and hit "Replace all":

105 lines | [composer.json](#)

```
1  {
  ... lines 2 - 3
4  "require": {
  ... lines 5 - 24
25    "symfony/asset": "5.0.*",
26    "symfony/console": "5.0.*",
27    "symfony/dotenv": "5.0.*",
  ... line 28
29    "symfony/form": "5.0.*",
30    "symfony/framework-bundle": "5.0.*",
31    "symfony/mailer": "5.0.*",
32    "symfony/messenger": "5.0.*",
  ... line 33
34    "symfony/security-bundle": "5.0.*",
35    "symfony/sendgrid-mailer": "5.0.*",
  ... line 36
37    "symfony/twig-bundle": "5.0.*",
  ... line 38
39    "symfony/validator": "5.0.*",
  ... line 40
41    "symfony/yaml": "5.0.*",
  ... lines 42 - 44
45  },
46  "require-dev": {
  ... lines 47 - 48
49    "symfony/browser-kit": "5.0.*",
50    "symfony/debug-bundle": "5.0.*",
  ... line 51
52    "symfony/phpunit-bridge": "5.0.*",
  ... line 53
54    "symfony/var-dumper": "5.0.*"
55  },
  ... lines 56 - 96
97  "extra": {
98    "symfony": {
  ... lines 99 - 100
101      "require": "5.0.*"
102    }
103  }
104 }
```

And then... make sure that this didn't accidentally replace any non-Symfony packages that may have had the same version.... looks good.

Updating Symfony Packages in Composer

We're ready! At your terminal... I'll hit Ctrl+C to stop the log tail.... and run the same command we used when upgrading from Symfony 4.3 to 4.4:

```
$ composer update "symfony/*"
```

That's it! It's that easy! We're done! Kidding - it's never *that* easy: you will almost *definitely* get some dependency errors. Probably... several. Ah, here's our first.

[Composer: Many Packages Need to Update](#)

These errors are always a *little* hard to read. This says that the current version of doctrine/orm in our project is not compatible with Symfony 5... which means that it *also* needs to be updated. Specifically we need a newer version that's compatible with symfony/console version 5.

And... it's possible that there is *not* yet a release of doctrine/orm that supports Symfony 5 - we hit that problem earlier with StofDoctrineExtensionsBundle. But... let's blindly try it! Add doctrine/orm to our update list and try again:

```
$ composer update "symfony/*" doctrine/orm
```

And... another error. Actually, the *same* error but this time for knplabs/knp-markdown-bundle. We don't *know* if this bundle has a Symfony5-compatible release either... and even if it does... it might require a *major* version upgrade. But the easiest thing to do is add it to our list and hope for the best. Try it:

```
$ composer update "symfony/*" doctrine/orm knplabs/knp-markdown-bundle
```

So... this is going to happen *several* more times - this is the same error for knplabs/knp-snappy-bundle. Little-by-little, we're discovering *all* the packages that we need to upgrade to be compatible with Symfony 5. *Instead* of doing this one-by-one, you *can* also choose the easy route: just run composer update with no arguments and allow Composer to update *everything*.

I prefer to upgrade more cautiously than that... but it's not a bad option. After all, our Composer version constraints don't allow any *major* version upgrades: so running composer update *still* won't allow any new major package versions unless you tweaked your composer.json file.

Let's keep going with my cowardly, I mean, cautious way: copy the package name and add it to the update command:

```
$ composer update "symfony/*" \  
$ doctrine/orm \  
$ knplabs/knp-markdown-bundle \  
$ knplabs/knp-snappy-bundle
```

Let's keep trying and... I'll fast-forward through a few more of these: this is for liip/imagine-bundle - add that to the update command - then oneup/flysystem-bundle... and now sensio/framework-extra-bundle: add that to our very-long update command:

```
$ composer update "symfony/*" \  
$ doctrine/orm \  
$ knplabs/knp-markdown-bundle \  
$ knplabs/knp-snappy-bundle \  
$ liip/imagine-bundle \  
$ oneup/flysystem-bundle \  
$ sensio/framework-extra-bundle
```

[Updating --with-dependencies](#)

Hmm, but this *next* error looks a bit different: it's something about doctrine/orm and doctrine/instantiator. If you look closely, this says that in order to get Symfony 5 support, we need doctrine/orm version 2.7, but version 2.7 requires doctrine/instantiator 1.3... and our project is currently *locked* at version 1.2.

Our app doesn't require doctrine/instantiator directly: it's a dependency of doctrine/orm. We saw this earlier when we were updating doctrine-migrations-bundle and we *also* needed to allow its dependency - doctrine/migrations to update.

We allow that by adding --with-dependencies to the update command:

```
$ composer update "symfony/*" \
$     doctrine/orm \
$     knplabs/knp-markdown-bundle \
$     knplabs/knp-snappy-bundle \
$     liip/imaginer-bundle \
$     oneup/flysystem-bundle \
$     sensio/framework-extra-bundle \
$     --with-dependencies
```

Updating our PHP Version

And... this gets us to our next error. Oh, interesting! Apparently nexylan/slack-bundle version 2.2.1 requires PHP 7.3! We saw a similar error earlier, which caused us to decide that our production app would *now* need to at *least* run PHP 7.2. We enforced that by adding a config.platform.php setting in composer.json to 7.2.5. This says:

Yo Composer! Pretend I'm using PHP 7.2.5 and don't let me use any packages that require a higher version of PHP.

So... hmm. Apparently the version of nexylan/slack-bundle that supports Symfony 5 *requires* PHP 7.3. Basically... unless we want to stop using that bundle, it means that *we* need to start using PHP 7.3 as well.

Fortunately, I'm already using PHP 7.3 locally: so I just need to change my config.platform.php setting to 7.3 and also make sure that we have 7.3 on production.

Inside composer.json, search for platform: there it is. Use 7.3.0. And, even though it doesn't affect anything in a project, also change the version under the require key:

```
105 lines | composer.json
1  {
  ... lines 2 - 3
4    "require": {
5      "php": "^7.3.0",
  ... lines 6 - 44
45  },
  ... lines 46 - 55
56  "config": {
  ... lines 57 - 60
61    "platform": {
62      "php": "7.3.0"
63    }
64  },
  ... lines 65 - 103
104 }
```

Ok, *now* try to update:

```
$ composer update "symfony/*" \
$     doctrine/orm \
$     knplabs/knp-markdown-bundle \
$     knplabs/knp-snappy-bundle \
$     liip/imaginer-bundle \
$     oneup/flysystem-bundle \
$     sensio/framework-extra-bundle \
$     --with-dependencies
```

Bah! I should've seen that coming: it's *still* complaining about nexylan/slack-bundle: it's reminding us that we need to *also* allow that bundle to update. Add it to our list:

```
$ composer update "symfony/*" \  
$ doctrine/orm \  
$ knplabs/knp-markdown-bundle \  
$ knplabs/knp-snappy-bundle \  
$ liip/imagique-bundle \  
$ oneup/flysystem-bundle \  
$ sensio/framework-extra-bundle \  
$ nexylan/slack-bundle \  
$ --with-dependencies
```

And try it. Surprise! Another package needs to be update. I *swear* we're almost done. Add that to our *gigantic* update command:

```
$ composer update "symfony/*" \  
$ doctrine/orm \  
$ knplabs/knp-markdown-bundle \  
$ knplabs/knp-snappy-bundle \  
$ liip/imagique-bundle \  
$ oneup/flysystem-bundle \  
$ sensio/framework-extra-bundle \  
$ nexylan/slack-bundle \  
$ knplabs/knp-time-bundle \  
$ --with-dependencies
```

Other than Symfony: (Mostly) Only Safe Minor Upgrades

And... whaaaaat? It's working! It's upgrading a *ton* of packages, including the Symfony stuff to 5.0.2. *And*, because we didn't change any other version constraints inside composer.json, we know that all of these upgrades are just *minor* version upgrades at best. For example, nexylan/slack-bundle went from 2.1 to 2.2. Even if there *were* a new version 3 of this bundle, we know that it wouldn't upgrade to it because its version constraint is ^2.1, which allows 2.1 or higher, but *not* 3:

```
105 lines | composer.json  
1  {  
  ... lines 2 - 3  
4  "require": {  
  ... lines 5 - 20  
21  "nexylan/slack-bundle": "^2.1",  
  ... lines 22 - 44  
45  },  
  ... lines 46 - 103  
104 }
```

Well, that's not *completely* true: check out nexylan/slack: it went from version 2.3 to 3: that *is* a major upgrade. That's because this is one of those *transitive* dependencies: this package isn't in our composer.json, it only lives in our project because nexylan/slack-bundle requires it. So unless we're using its code directly - which *is* possible, but less likely - the major upgrade won't affect us. If you're worried, check its CHANGELOG.

Ok, so we are now on Symfony 5. Woo! The little icon on the bottom right of the web debug toolbar shows 5.0.2.

Next, let's celebrate by trying out some new features! We'll start by talking about Symfony's new "secrets management".

Chapter 22: Secrets Management Setup

My favorite new feature in Symfony 4.4 and 5 - other than the fact that Messenger and Mailer are now stable - is probably the new secrets management system, which *is* as cool as it sounds.

Secrets?

Here's the deal: every app has a set of config values that need to be different from machine to machine, like different on my local machine versus production. In Symfony, we store these as environment variables.

One example is MAILER_DSN:

```
62 lines | .env
... lines 1 - 38
39 ###> symfony/mailer ###
40 MAILER_DSN=null://null
41 # in Symfony 4.4 and higher, the syntax is
42 # MAILER_DSN=null://default
43 ###
... lines 44 - 62
```

While developing, I want to use the null transport to *avoid* sending real emails. But on production, this value will be different, maybe pointing to my SendGrid account.

We reference environment variables with a special syntax - this one is in config/packages/mailer.yaml: %env()% with the variable name inside: MAILER_DSN:

```
4 lines | config/packages/mailer.yaml
1 framework:
2   mailer:
3     dsn: '%env(MAILER_DSN)%'
```

If you look at the full list of environment variables, you'll notice that there are two types: sensitive and non-sensitive variables.

For example, MAILER_DSN is a "sensitive" variable because the production value probably contains a username & password or API key: something that, if someone got access to it, would allow them to use our account. So, it's not something that we want to commit to our project.

But other values are *not* sensitive, like WKHTMLTOPDF_PATH:

```
62 lines | .env
... lines 1 - 44
45 ###> knplabs/knp-snappy-bundle ###
46 WKHTMLTOPDF_PATH=/usr/local/bin/wkhtmltopdf
47 WKHTMLTOIMAGE_PATH=/usr/local/bin/wkhtmltoimage
48 ###
... lines 49 - 62
```

This might need to be *different* on production, but the value is not sensitive: we don't need to keep it a secret. *We could* actually commit its production value somewhere in our app to make deployment easier if we wanted to.

So... why are we talking about this? Because, these sensitive, or "secret" environment variables make life tricky. When we deploy, we need to *somehow* set the MAILER_DSN variable to its secret production value, either as a real environment variable or probably by creating a .env.local file. Doing that safely can be tricky: do you store the secret production value in a config file in this repository or in some deploy script? You can, but then it's not very secure: the less people that can see your secrets - even people on your team - the better.

The Vault Concept

One general solution to this problem is something called a vault. The basic idea is simple: you encrypt your secrets - like the production value for MAILER_DSN - and then store the *encrypted* value. The "place" where the encrypted secrets are stored is called the "vault". The secrets inside can only be *read* if you have the decryption password or "private key".

This makes life easier because now your secrets can safely be stored in this "vault", which can just be a set of files on your filesystem or even a cloud vault service. Then, when you deploy, the only "secret" that you need to have available is the password or private key. Some vaults also allow other ways to authenticate.

Introducing Symfony's Secrets "Vault"

None of this "vault" stuff has anything to do with Symfony: it's just a cool concept and there are various services & projects that support the idea - the most famous being HashiCorp's Vault.

But, in Symfony 4.4, a new secrets system was added to let us do *all* this cool stuff out-of-the-box.

Here's the goal: instead of having MAILER_DSN as an environment variable, we're going to move this to be an "encrypted secret".

Dumping an Env Var for Debugging

To see how this all works clearly, let's add some debugging code to dump the MAILER_DSN value. Open config/services.yaml and add a new bind - \$mailerDsn set to %env(MAILER_DSN)% - so we can use this as an argument somewhere:

```
51 lines | config/services.yaml
... lines 1 - 11
12  services:
13      # default configuration for services in *this* file
14      _defaults:
... lines 15 - 17
18      # setup special, global autowiring rules
19      bind:
... lines 20 - 24
25      $mailerDsn: '%env(MAILER_DSN)%'
... lines 26 - 51
```

I forgot my closing quote... which Symfony will "gently" remind me in a minute.

Next, open src/Controller/ArticleController.php. In the homepage action, thanks to the bind, we can add a \$mailerDsn argument. Dump that and die:

```
66 lines | src/Controller/ArticleController.php
... lines 1 - 13
14  class ArticleController extends AbstractController
15  {
... lines 16 - 28
29      public function homepage(ArticleRepository $repository, $mailerDsn)
30      {
31          dump($mailerDsn);die;
... lines 32 - 36
37      }
... lines 38 - 64
65  }
```

Now, refresh the homepage. Booo. Let's go fix my missing quote in the YAML file. Refresh again and... perfect: the current value is null://null.

That's no surprise: that's the value in .env and we are *not* overriding it in .env.local:

62 lines | `.env`

... lines 1 - 38

39 `###> symfony/mailer ###`

40 `MAILER_DSN=null://null`

... lines 41 - 62

[Converting an Env Var to a Secret](#)

Ok, as *soon* as you have an environment variable that you want to *convert* to a secret, you need to fully *remove* it as an environment variable: do *not* set it as an environment variable anywhere anymore. I'll remove `MAILER_DSN` from `.env` and if we *were* overriding it in `.env.local`, I would also remove it from there:

62 lines | `.env`

... lines 1 - 38

39 `###> symfony/mailer ###`

40 `# MAILER_DSN=null://null`

... lines 41 - 62

Not surprisingly, when you refresh, we're greeted with a great big ugly error:

The environment variable is not found.

[Bootstrapping the Secrets Vault](#)

So how *do* we make `MAILER_DSN` an encrypted secret? With a fancy new console command:

```
$ php bin/console secrets:set MAILER_DSN
```

That will ask us for the value: I'll go copy `null://null` - you'll learn why I'm choosing that value in a minute - and paste it here. You don't see the pasted value because the command hides the input to be safe.

[The Public/Encryption & Private/Decryption Keys](#)

Hit enter and... awesome! Because this was the *first* time we added something to the secrets vault, Symfony needed to *create* the vault - and it did that automatically. What does that *actually* mean? It means that it created several new files in a `config/secrets/dev` directory.

Let's go check them out: `config/secrets/dev`. Ooooo.

To "create" the secrets vault, Symfony created two new files, which represent "keys": a private *decrypt* key and a public *encrypt* key. If you look inside, they're just fancy text files: they return a long key value.

The public encrypt file is something that *is* safe to commit to your repository. It's used to *add*, or "encrypt" a secret, but it can't *read* encrypted secrets. By committing it, other developers can add new secrets.

The private decrypt key - as its name suggests - is needed to decrypt and *read* secrets.

[One set of Secrets per Environment](#)

Now *normally*, the "decrypt" key is *private* and you would *not* commit it to your repository. However, as you may have noticed, Symfony maintains a different set of secrets per *environment*. The vault we created is for the dev environment only. In the next chapter, we'll create the vault for the prod environment.

Anyways, because secrets in the dev environment usually represent safe "defaults" that aren't terribly sensitive, it's ok to commit the private key for the dev environment. Plus, if you *didn't* commit it, other developers on your team wouldn't be able to run the app locally... because Symfony wouldn't be able to read the dev secrets.

[Committing the dev Keys](#)

Let's add these to git:

```
$ git status
```

Then git add config/secrets and also add .env:

```
$ git add config/secrets .env
```

This added *all* 4 files. The other two files store info about the secrets themselves: each secret will be stored in its own file and the "list" file just helps us get the full list of secrets that exist. Commit this:

```
$ git commit -m "setting up dev environment vault"
```

[%env\(\)% Automatically Looks for Secrets](#)

And *now* I have a pleasant surprise: go over and refresh the homepage. It works! That's by design: the %env()% syntax is smart:

```
51 lines | config/services.yaml
... lines 1 - 11
12  services:
13    # default configuration for services in *this* file
14    _defaults:
... lines 15 - 17
18    # setup special, global autowiring rules
19    bind:
... lines 20 - 24
25    $mailerDsn: '%env(MAILER_DSN)%'
... lines 26 - 51
```

It *first* looks for a MAILER_DSN environment variable. If it finds one, it uses it. If it does *not*, it *then* looks for a MAILER_DSN secret. *That's* why... it just works.

[bin/console secrets:list](#)

To get a list of *all* the encrypted secrets, you can run:

```
$ php bin/console secrets:list
```

Yep - just one right now. Add --reveal to see the values. By the way, this "reveal" *only* works because the decrypt file exists in our app.

Next: our app will *not* currently work in the prod environment because there is no prod vault and so no MAILER_DSN prod secret. Let's fix that and talk a bit about deployment.

Chapter 23: Production Secrets

Whenever you add a new secret, you need to make sure that you add it to the dev environment *and* the prod environment. That's because each *set* of secrets, or "vault" as I've been calling it, is specific to the *environment*. This vault of secrets, for example, will *only* be loaded in the dev environment. So, unless we *also* add MAILER_DSN to the prod vault, the prod environment will be... yep! *Totally* broken. And a busted production environment is... a bummer.

Creating the Production (prod) Vault

So, how *do* we add MAILER_DSN to the prod vault? With the same command as before: `secrets:set`, but *this* time with `--env=prod`:

```
$ php bin/console secrets:set --env=prod MAILER_DSN
```

I'll paste in my production SendGrid value... which you can't see because the command hides the input to be safe.

Cool! *Just* like last time, because this is the first time we've added a key to the prod vault, it automatically *created* the vault for us... which means that it created the decrypt and encrypt keys.

Production Encrypt & Decrypt Keys

And just like with the dev environment, the encrypt key file *is* safe to commit to your repository. Heck, you could post it onto the Internet! It only gives people the power to *add* things to your vault, which is probably something that you *do* want any developer to be able to do.

But the decrypt key file should *not* be committed to the repository. It is *incredibly* sensitive: it has the power to decrypt *all* of your production secrets! We decided that it was *probably* ok to commit the dev decrypt key... because the dev keys are probably not very sensitive. But you should *not* commit this one. Or, if you do - just realize that everyone who has access to view files in your repository will have access to all your secrets... and you might as well just commit them as plain-text values.

We'll talk more about the decrypt key in a few minutes.

Add the new vault files to git:

```
$ git add config/
```

Then:

```
$ git status
```

Oh! This did *not* add the private decrypt key. That's no accident: our `.gitignore` file is *specifically* ignoring this:

```
27 lines | .gitignore
1  ###> symfony/framework-bundle ###
   ... lines 2 - 4
5  /config/secrets/prod/prod.decrypt.private.php
   ... lines 6 - 8
9  ###
   ... lines 10 - 27
```

This line was added when we updated the symfony/framework-bundle recipe.

[Listing & Revealing prod Secrets](#)

Anyways, *just* like with the dev vault, we can list the secrets:

```
$ php bin/console secrets:list --env=prod
```

And because my app *does* have the decrypt key, we can add `--reveal` to see their values:

```
$ php bin/console secrets:list --env=prod --reveal
```

[Secrets are Committed](#)

Ok, let's commit!

```
$ git commit -m "Adding MAILER_DSN to prod vault"
```

Do you realize how awesome that was? We just *safely* committed a secret value to the repository! Secrets are version controlled, which means that we can see when a secret is added on a pull request and can even check later to see why and when a secret was added. That's a huge step!

[Deploying with the Decrypt Key](#)

Now, instead of needing to figure out how and where to securely store *all* our sensitive values so that we can add them to our app when we deploy, there is now just *one* sensitive value: the decrypt key file.

When we deploy to production, the *only* thing we need to worry about now is creating that decrypt file with this long value inside. Or, you can base64_encode the key's *value* and set it on a special environment variable called `SYMFONY_DECRYPTION_SECRET`. You can use a PHP trick to get the exact value to set on that env var:

```
$ php -r 'echo base64_encode(require "config/secrets/prod/prod.decrypt.private.php");'
```

The point is, on production you either need to re-create the `prod.decrypt.private.php` file or set the `SYMFONY_DECRYPTION_SECRET` environment variable. How? That depends completely on your deploy. For example, with SymfonyCloud - which is what we use - we set the decrypt key as a SymfonyCloud "variable".

However you deploy, whatever is responsible for deploying your app should be the *one*, um, "thing" that has access to the private key.

[Seeing the prod Secret Value](#)

Let's go make sure this whole prod vault idea works. Right now, if we refresh the page, it still shows us the null value because we are *still* in the dev environment.

Open up your `.env` file and, temporarily, change `APP_ENV` to `prod`:

```
62 lines | .env
... lines 1 - 15
16  ###> symfony/framework-bundle ###
17  APP_ENV=prod
... lines 18 - 20
21  ###
... lines 22 - 62
```

Then, find your console and clear the cache:

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

I don't need to add `--env=prod` *now* because we are *already* in the prod environment thanks to the `APP_ENV` change.

Ok, go try it! Refresh and... yes! That's the value from the prod vault! Symfony automatically used the private key to decrypt it.

[And if the Decrypt Key is Missing?](#)

What would happen if the decrypt key wasn't there? Let's find out! Temporarily delete the decrypt key - but make sure you can get it back: if you lose this key, you won't *ever* be able to decrypt your secrets and you'll need to create a *new* private key and re-add them all again. That would be... a bummer.

Refresh now to see... oh! Giant 500 page... but we can't see the error. Check out the logs:

```
$ tail var/log/prod.log
```

And... there it is:

```
Environment variable not found: "MAILER_DSN".
```

If you *don't* have the private key... bad things will happen. Let's go *undelete* that private key file. Refresh: all better. Let's also change back to the dev environment to make life nicer:

```
62 lines | .env
... lines 1 - 15
16  ###> symfony/framework-bundle ###
17  APP_ENV=dev
... lines 18 - 20
21  ###
... lines 22 - 62
```

So... that's it! You have a dev vault and a prod vault, you can commit your encrypted secrets via Git and you only need to handle one sensitive value at deploy: the private decrypt key.

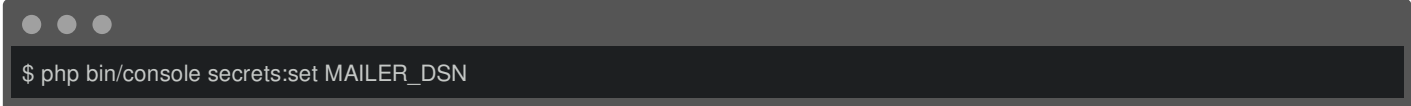
But... what if a developer needs to locally override a value in the dev environment? For example, in our dev vault, `MAILER_DSN` uses the null transport so that emails are *not* sent. What if I need to temporarily change Mailtrap so that I can test the emails?

The answer: the "local" vault... a little bit of coolness that will open up a couple of neat possibilities. That's next.

Chapter 24: Overriding Secrets Locally (Local Vault)

What if I need to override a secret value on my local machine? MAILER_DSN is a perfect example: in the dev secrets vault, it's set to use the null transport. What if I want to see what an email *looks* like... and so I need to override that value locally to send to Mailtrap?

Well, we *could* run over to the terminal and say:



```
$ php bin/console secrets:set MAILER_DSN
```

And then modify the vault value. But... ugh - then I have to be *super* careful not to commit that change... and eventually... I will on accident... and I'll look super uncool because I accidentally changed a development secret. *Fortunately*, for absent-minded committers like me, there's a built-in solution to help!

[Setting a Secret into the "Local" Vault](#)

Pretend like we're going to override the MAILER_DSN secret... but add an extra `--local` flag to the end:



```
$ php bin/console secrets:set MAILER_DSN --local
```

So far... this looks identical to before. I'll paste in my Mailtrap value... which the command hides for security reasons. And... *fascinating!* This didn't change our dev vault at all! Nope, it apparently added the secret to `.env.dev.local`.

Quick review about `.env` files: Symfony allows you to create a `.env.local` file as a way to override values in `.env`. And thanks to our `.gitignore`, `.env.local` is ignored from Git. And, though it's not as common, you can *also* create a `.env.dev.local` file. It works the same way: it overrides `.env` and isn't committed. The only difference - which is super minor - is that it's *only* loaded in the dev environment.

The point is: this "local" vault thing is nothing more than a fancy way of setting an environment variable to this "local" file.

[Environment Variables Override Secrets](#)

And... wait: that's kind of beautiful! I mentioned earlier that when you use the environment variable system - when you use that `%env()%` syntax - Symfony *first* looks for MAILER_DSN as an *environment* variable. If it finds it, it uses it. And only if it does *not* find it, does it *then* go and try to see if it is a secret.

So now, in the dev environment on my machine, it *will* find MAILER_DSN as an environment variable! Go refresh the page to prove it. There it is: my local override.

You can use this cool `secrets:set --local` thing if you want... but really all you need to understand is that if you want to override a secret value locally, just set it as an environment variable.

And, personally, I don't love having `.env.local` *and* `.env.dev.local` - it seems like overkill. So I would delete `.env.dev.local` and instead put my overridden MAILER_DSN directly into `.env.local`.

But... don't do that - delete the override entirely: it'll help me show you one more thing.

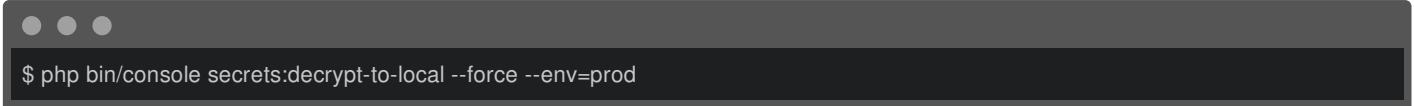
Now that we understand that environment variables override secrets, we can unlock three possibilities. The first is what we just saw: we can override a secret locally by creating an environment variable. The other two deal with a performance optimization on production *and*... our test environment... which is currently busted! That's next.

Chapter 25: Prod Vault Optimization & Vault for Tests

Now that we know that environment variables override secrets, we can use that to our advantage in two ways.

[Dumping Secrets on Deploy: secrets:decrypt-to-local](#)

The first thing is that, during deployment, we can dump our production secrets into a local file. Check it out. Run:



```
$ php bin/console secrets:decrypt-to-local --force --env=prod
```

And... no output. *Lame! SO lame* that, in Symfony 5.1, this command *will* have output - that pull request was already merged.

Anyways, this just created a new `.env.prod.local` file... which contains *all* our prod secrets... which is just one right now. This means that, when we're in the prod environment, it will read from *this* file and will *never* read secrets from the vault.

Why... is that interesting? Um, good question. Two reasons. First, while deploying, you can add the decrypt key file, run this command, then *delete* the key file. The private key file then does not need to live on your production server at all. That's one less thing that could be exposed if someone got access to your servers.

And second, this will give you a *minor* performance boost because the secrets don't need to be decrypted at runtime.

Now, you might be thinking:

Ryan! You crazy man, you've got the brains of a watering can! We went to *all* this trouble to encrypt our secrets, and now you want me to store them in plain text on production? Are you mad?

I never get mad! The truth is, your sensitive values are *never* fully safe on production: there is always *some* way - called an "attack vector" - to get them. If someone gets access to the files on your server, then they would already have your encrypted values *and* the private key to decrypt them. Storing the secrets in plain text but *removing* the decrypt key from production is really the same thing from a security standpoint.


The point is: there's no security difference. Let's delete the `.env.prod.local` file, because we don't need it right now.

[Secrets for the test Environment](#)

The *other* interesting thing that we can do now that we understand that environment variables override secrets is related to the test environment. Because... our test environment is *totally* broken.

Think about it: in the test environment, there is *no* vault! And so there is *no* MAILER_DSN secret. Do we *also* need a test vault? Nah. There's a simpler solution.

First, let's *run* our tests to see what's going on:



```
$ php bin/phpunit
```

Ignore the deprecation warnings. Woh! *Huge* error. If you look closely... yep:

Environment variable not found: "MAILER_DSN"

[New in 4.4: Easier HTML Errors in Tests](#)

By the way, trying to find the error message inside the HTML in a test... *sucks*. But it's easier in Symfony 4.4 because Symfony dumps the error as a comment on the top of the HTML. It also.... yep! Puts that same comment at the bottom. So actually... I didn't need to scroll so far up.

[Secrets in the Test Environment](#)

So we *do* need to specify a MAILER_DSN secret to use in the test environment. But for simplicity, instead of making another vault, let's just add it to .env.test. I'll copy my old null transport value from .env, and put it into .env.test:

```
8 lines | .env.test
... lines 1 - 6
7 MAILER_DSN=null://null
```

Done! So *really*, when you need to add a new secret, you need to add it to your dev vault, prod vault *and* .env.test.

Let's try the tests again:

```
$ php bin/phpunit
```

Much better! So... that's it for the secrets system! Pretty freakin' cool! Let's clean up our debugging code... nobody likes data being dumped on production. I'll remove the bind:

```
51 lines | config/services.yaml
... lines 1 - 11
12 services:
13     # default configuration for services in *this* file
14     _defaults:
... lines 15 - 18
19     bind:
... lines 20 - 24
25     $mailerDsn: '%env(MAILER_DSN)%'
... lines 26 - 51
```

then go to ArticleController and take out the \$mailerDsn stuff there:

```
66 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 28
29     public function homepage(ArticleRepository $repository, $mailerDsn)
30     {
31         dump($mailerDsn);die;
... lines 32 - 36
37     }
... lines 38 - 64
65 }
```

Next, let's talk about a really cool new feature called "validation auto mapping". It's a wicked-smart feature that automatically adds validation constraints based on your Doctrine metadata and also based on the way that your PHP code is written in your class.

Chapter 26: Validation Auto-Mapping

Head over to `/admin/article` and log in as an admin user: `admin1@thespacebar.com` password `engage`. Use this unchecked admin *power* to go to `/admin/article` and click to create a new article.

I *love* the new "secrets" feature... but what I'm about to show you might be my *second* favorite new thing. It actually comes from Symfony 4.3 but was improved in 4.4. It's called: validation auto-mapping... and it's one more step towards robots doing my programming for me.

Start by going into `templates/article_admin/_form.html.twig`:

42 lines | [templates/article_admin/ form.html.twig](#)

```
1  {{ form_start(articleForm) }}
2  {{ form_row(articleForm.title, {
3      label: 'Article title'
4  }) }}
5
6  <div class="row">
7      <div class="col-sm-9">
8          {{ form_row(articleForm.imageFile, {
9              attr: {
10                  'placeholder': 'Select an article image'
11              }
12          }) }}
13      </div>
14      <div class="col-sm-3">
15          {% if articleForm.vars.data.imageFilename|default %}
16              <a href="{{ uploaded_asset(articleForm.vars.data.imagePath) }}" target="_blank">
17                  
18              </a>
19          {% endif %}
20      </div>
21  </div>
22
23  {{ form_row(articleForm.author) }}
24  {{ form_row(articleForm.location, {
25      attr: {
26          'data-specific-location-url': path('admin_article_location_select'),
27          'class': 'js-article-form-location'
28      }
29  }) }}
30  <div class="js-specific-location-target">
31      {% if articleForm.specificLocationName is defined %}
32          {{ form_row(articleForm.specificLocationName) }}
33      {% endif %}
34  </div>
35  {{ form_row(articleForm.content) }}
36  {% if articleForm.publishedAt is defined %}
37      {{ form_row(articleForm.publishedAt) }}
38  {% endif %}
39
40  <button type="submit" class="btn btn-primary">{{ button_text }}</button>
41  {{ form_end(articleForm) }}
```

This is the form that renders the article admin page. To help us play with validation, on the button, add a `formnovalidate` attribute:

42 lines | [templates/article_admin/ form.html.twig](#)

```
1  {{ form_start(articleForm) }}
... lines 2 - 39
40  <button type="submit" class="btn btn-primary" formnovalidate>{{ button_text }}</button>
41  {{ form_end(articleForm) }}
```

Thanks to that, after you refresh, HTML5 validation is *disabled* and we can submit the entire form blank to see... our server-side validation errors. These come from the annotations on the Article class, like `@Assert\NotBlank` above `$title`:

```

325 lines | src/Entity/Article.php
... lines 1 - 12
13 use Symfony\Component\Validator\Constraints as Assert;
... lines 14 - 18
19 class Article
20 {
... lines 21 - 29
30 /**
... line 31
32 * @Assert\NotBlank(message="Get creative and think of a title!")
33 */
34 private $title;
... lines 35 - 323
324 }

```

So it's *no* surprise that if we remove the `@Assert\NotBlank` annotation... I'll move it as a comment below the property:

```

325 lines | src/Entity/Article.php
... lines 1 - 18
19 class Article
20 {
... lines 21 - 29
30 /**
31 * @ORM\Column(type="string", length=255)
32 */
33 private $title;
34 // @Assert\NotBlank(message="Get creative and think of a title!")
... lines 35 - 323
324 }

```

That's as good as deleting it. And then re-submit the blank form... the validation error is *gone* from that field.

[@Assert\EnableAutoMapping](#)

Ready for the magic? Go back to Article and, on top of the class, add `@Assert\EnableAutoMapping()`:

```

326 lines | src/Entity/Article.php
... lines 1 - 12
13 use Symfony\Component\Validator\Constraints as Assert;
... lines 14 - 15
16 /**
... line 17
18 * @Assert\EnableAutoMapping()
19 */
20 class Article
21 {
... lines 22 - 324
325 }

```

As *soon* as we do that, we can refresh to see... Kidding! We refresh to see... the validation error is *back* for the title field!

This value should not be null

Yep! A `@NotNull` constraint was *automatically* added to the property! How the heck did that work? The system - validation auto-mapping - automatically adds sensible validation constraints based off of your Doctrine metadata. The Doctrine Column annotation has a nullable option and its *default* value is `nullable=false`:

326 lines | [src/Entity/Article.php](#)

... lines 1 - 19

20 class Article

21 {

... lines 22 - 30

31 /**

32 * @ORM\Column(type="string", length=255)

33 */

34 private \$title;

... lines 35 - 324

325 }

In other words, the title column is required in the database! And so, a constraint is added to make it required on the form.

Auto-mapping can *also* add constraints based *solely* on how your code is written... I'll show you an example of that in a few minutes. Oh, and by the way, to get the most out of this feature, make sure you have the symfony/property-info component installed.

```
$ composer show symfony/property-info
```

If that package doesn't come up, install it to allow the feature to get as *much* info as possible.

[Auto-Mapping is Smart](#)

Let's play with this a bit more, like change this to nullable=true:

326 lines | [src/Entity/Article.php](#)

... lines 1 - 19

20 class Article

21 {

... lines 22 - 30

31 /**

32 * @ORM\Column(type="string", length=255, nullable=true)

33 */

34 private \$title;

... lines 35 - 324

325 }

This means that the column should now be *optional* in the database. What happens when we submit the form now? The validation error is gone: the NotNull constraint was *not* added.

Oh, but it gets even *cooler* than this. Remove the @ORM\Column entirely - we'll pretend like this property isn't even being saved to the database. I also need to remove this @Gedmo\Slug annotation to avoid an error:

```

326 lines | src/Entity/Article.php
... lines 1 - 19
20 class Article
21 {
... lines 22 - 30
31 /**
32  * @ORM\Column(type="string", length=255, nullable=true)
33  */
34 private $title;
... lines 35 - 36
37 /**
... line 38
39  * @Gedmo\Slug(fields={"title"})
40  */
41 private $slug;
... lines 42 - 324
325 }

```

What do you think will happen now? Well think about it: the auto-mapping system won't be able to ask Doctrine if this property is required or not... so my guess is that it *won't* add any constraints. Try it! Refresh!

Duh, duh, duh! The NotNull validation constraint is back! Whaaaaat? The Doctrine metadata is just *one* source of info for auto-mapping: it can also look directly at your *code*. In this case, Symfony looks for a setter method. Search for setTitle():

```

326 lines | src/Entity/Article.php
... lines 1 - 19
20 class Article
21 {
... lines 22 - 113
114 public function setTitle(string $title): self
115 {
116     $this->title = $title;
117
118     return $this;
119 }
... lines 120 - 324
325 }

```

Ah yes, the \$title argument is type-hinted with string. And because that type-hint does *not* allow null, it assumes that \$title must be required and adds the validation constraint.

Watch this: add a ? before string to make null an allowed value:

```

322 lines | src/Entity/Article.php
... lines 1 - 19
20 class Article
21 {
... lines 22 - 109
110 public function setTitle(?string $title): self
111 {
112     $this->title = $title;
113
114     return $this;
115 }
... lines 116 - 320
321 }

```

Refresh now and... the error is gone.

[Avoiding Duplicate Constraints](#)

Let's put *everything* back to where it was in the beginning. What I *love* about this feature is that... it's just so smart! It accurately *reflects* what your code is already communicating.

And even if I add back my `@Assert\NotBlank` annotation:

```
326 lines | src/Entity/Article.php
... lines 1 - 15
16  /**
... line 17
18  * @Assert\EnableAutoMapping()
19  */
20  class Article
21  {
... lines 22 - 30
31  /**
32  * @ORM\Column(type="string", length=255)
33  * @Assert\NotBlank(message="Get creative and think of a title!")
34  */
35  private $title;
36
37  /**
38  * @ORM\Column(type="string", length=100, unique=true)
39  * @Gedmo\Slug(fields={"title"})
40  */
41  private $slug;
... lines 42 - 113
114  public function setTitle(string $title): self
115  {
116      $this->title = $title;
117
118      return $this;
119  }
... lines 120 - 324
325 }
```

And refresh... check it out. We don't get 2 errors! The auto-mapping system is smart enough to realize that, because I added a `NotBlank` annotation constraint to this property, it should not *also* add the `NotNull` constraint: that would basically be duplication and the user would see two errors. Like I said, it's smart.

[Automatic Length Annotation](#)

And it's not all about the `NotNull` constraint. The length of this column in the database is 255 - that's the default for a string type. Let's type a super-creative title over and over and over and over again... until we know that we're above that limit. Submit and... awesome:

This value is too long. It should have 255 characters or less.

Behind-the-scenes, auto-mapping *also* added an `@Length` annotation to limit this field to the column size. Say goodbye to accidentally allowing large input... that then gets truncated in the database.

[Disabling Auto-Mapping when it Doesn't Make Sense](#)

As *cool* as this feature is, automatic functionality will never work in *all* cases. And that's fine for two reasons. First, it's *your* choice to opt-into this feature by adding the `@EnableAutoMapping` annotation:

```

326 lines | src/Entity/Article.php
... lines 1 - 12
13 use Symfony\Component\Validator\Constraints as Assert;
... lines 14 - 15
16 /**
... line 17
18 * @Assert\EnableAutoMapping()
19 */
20 class Article
21 {
... lines 22 - 324
325 }

```

And second, you can disable it on a field-by-field basis.

A great example of when this feature can be a problem is in the User class. Imagine we added `@EnableAutoMapping` here and created a registration form bound to this class. Well... that's going to be a problem because it will add a `NotNull` constraint to the `$password` field! And we *don't* want that!

```

285 lines | src/Entity/User.php
... lines 1 - 19
20 class User implements UserInterface
21 {
... lines 22 - 47
48 /**
49 * @ORM\Column(type="string", length=255)
50 */
51 private $password;
... lines 52 - 283
284 }

```

In a typical registration form - like the one that the `make:registration-form` command creates - the `$password` property is set to its hashed value only *after* the form is submitted & validated. Basically, this is not a field the user sets directly and having the `NotNull` constraint causes a validation error on submit.

How do you solve this? You could disable auto-mapping for the whole class. Or, you could disable it for the `$password` property *only* by adding `@Assert\DisableAutoMapping`:

```

// src/Entity/User.php

class User implements UserInterface
{
    // ...
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\DisableAutoMapping()
     */
    private $password;
    // ...
}

```

This is the *one* ugly case for this feature, but it's easy to fix.

[Configuring Auto-Mapping Globally](#)

Oh, and one more thing! You can control the feature a bit in `config/packages/validator.yaml`. By default, you need to enable auto-mapping on a class-by-class basis by adding the `@Assert\EnableAutoMapping` annotation:

```
326 lines | src/Entity/Article.php
... lines 1 - 12
13 use Symfony\Component\Validator\Constraints as Assert;
... lines 14 - 15
16 /**
... line 17
18 * @Assert\EnableAutoMapping()
19 */
20 class Article
21 {
... lines 22 - 324
325 }
```

But, you can also *automatically* enable it for specific namespaces:

```
9 lines | config/packages/validator.yaml
1 framework:
2   validation:
... lines 3 - 4
5     # Enables validator auto-mapping support.
6     # For instance, basic validation constraints will be inferred from Doctrine's metadata.
7     #auto_mapping:
8     #   App\Entity\: []
```

If we uncommented this `App\Entity` line, every entity would get auto-mapped validation without needing the extra annotation. I like being a bit more explicit - but it's your call.

Next, ready to talk about something super geeky? No, not Star Trek, but that would awesome. This is probably *even* better: let's chat about password hashing algorithms. Trust me, it's actually *pretty* neat stuff. Specifically, I want to talk about safely *upgrading* hashed passwords in your database to stay up-to-date with security best-practices.

Chapter 27: Migrate Password Hashing

On our User entity, this \$password field - which is stored in the database - does *not* contain a plain-text version of the user's password:

```
285 lines | src/Entity/User.php
... lines 1 - 19
20 class User implements UserInterface
21 {
... lines 22 - 47
48 /**
49  * @ORM\Column(type="string", length=255)
50  */
51 private $password;
... lines 52 - 283
284 }
```

Next to allowing SQL injection attacks, storing plain-text passwords is *just* about the worst thing you can do in a web app.

Hashing Algorithms Over Time

Anyways, what's *actually* stored on this field is a "hash" or kind of "fingerprint" of the plaintext password and there are multiple hashing algorithms available. The one *you're* using is configured in config/packages/security.yaml:

```
61 lines | config/packages/security.yaml
1 security:
2   encoders:
3     App\Entity\User:
4       algorithm: bcrypt
... lines 5 - 61
```

The encoders section says that whenever we encode, or really, "hash" a password - like when someone registers or when they log in - the bcrypt algorithm will be used. That's great. But... over time, as processing power of computers get better and better, it becomes more and more possible that *if* your database of passwords somehow got exposed, someone could use a computer to *crack* them. It probably *won't* happen, but it's a security best-practice to change your algorithm over time to one that requires more processing power or memory.

Changing Algorithms

Comment-out the bcrypt algorithm and replace it with sodium:

```
62 lines | config/packages/security.yaml
1 security:
2   encoders:
3     App\Entity\User:
4       #algorithm: bcrypt
5       algorithm: sodium
... lines 6 - 62
```

This stuff can be confusing. Sodium is a hashing library that uses the Argon2 algorithm, which is *currently* considered the best algorithm.

So... great! We just changed from bcrypt to Argon2 and increased the security of our application. We deserve a donut!

Wait a second... put that donut down. You - usually - can't simply change from one algorithm to another. Why? The problem is

that all your *existing* users already have their passwords hashed with bcrypt. If *those* users tried to log in, suddenly Symfony would use sodium to hash the submitted password and it would not match the hash in the database.

Now, the *full* truth is that, in *this* case - going from bcrypt to sodium - nothing would break: Sodium is smart enough to detect that the existing passwords are hashed with bcrypt and use it instead. But in general, you can't change from one algorithm to another without breaking stuff. And even in this case, shouldn't we *also* re-hash the passwords of all our existing users with the newer algorithm?

The migrate_from Encoder Option

Symfony 4.4 comes with a wonderful new feature to help with this - submitted by the amazing [Nicolas Grekas](#), who is also responsible - along with [J  r  my Deruss  ](#) for the secrets management system.

Here's how it works: add a new encoder, it can be called anything, how about legacy_bcrypt. Make sure it has the *exact* configuration of your original encoder:

```
68 lines | config/packages/security.yaml
1  security:
2    encoders:
3      legacy_bcrypt:
4        algorithm: bcrypt
... lines 5 - 68
```

Next, under the *new* encoder - the one that will be used for my User class - add a new option: migrate_from. Below that, add a list of all encoders that existing users might be using - for us, just legacy_bcrypt:

```
68 lines | config/packages/security.yaml
1  security:
2    encoders:
3      legacy_bcrypt:
4        algorithm: bcrypt
5
6      App\Entity\User:
7        algorithm: sodium
8        migrate_from:
9          # allow existing bcrypt accounts to log in
10         # and migrate to sodium
11         - legacy_bcrypt
... lines 12 - 68
```

That's it! This says:

Hey! When somebody logs in, try to use the sodium algorithm. If that doesn't work, try the legacy_bcrypt algorithm. If *that* doesn't work, panic! I mean, if that doesn't work, the password is invalid.

Thanks to this, we can have a database where *some* passwords are hashed with sodium and others are hashed with bcrypt. Let's try it: log out and try to log back in: admin1@thespacebar.com, password engage. Got it!

Seeing the Hashed Passwords

It's *also* kinda fun to see how this looks in the database. Find your terminal and run:

```
$ php bin/console doctrine:query:sql 'SELECT email, password FROM user'
```

Interesting: every hashed password starts with the same \$2y thing. That's no accident: that's what the bcrypt hashing format looks like.

Let's see what sodium-encoded passwords look like: go back to your browser, log out, and register as a new user: Ryan, spacecadet@example.com, the same password - engage, but that doesn't matter - and register!

Try that query again:

```
$ php bin/console doctrine:query:sql 'SELECT email, password FROM user'
```

Cool! It's *pretty* obvious the new user's password is hashed with Argon.

Upgrading old Password

We now have a database mixed with passwords hashed with the older algorithm and the newer algorithm. That's fine... but in a *perfect* world, we would re-hash *all* the passwords using the newer algorithm.

But... we can't do that. Boo. In order to hash a password, we need the original *plain* password, which we don't have. So it's *not* possible to upgrade all existing users to the new algorithm.

Except, hmm, there is *one* time when we *do* have the plaintext password: at the moment any old user logs into the site. At that *instant*, in theory, we *could* re-hash the password using sodium and save it to the database. That would actually be pretty awesome.

And... that's *precisely* what `migrate_from` does automatically:

```
68 lines | config/packages/security.yaml
```

```
1 security:
2   encoders:
3     ... lines 3 - 5
6     App\Entity\User:
7     ... line 7
8     migrate_from:
9     ... lines 9 - 68
```

Well, *almost* automatically: we need to do two things in our code to enable it.

Guard PasswordAuthenticatedInterface

First, *if* you're using Guard authentication for your login form, your authenticator needs a new interface. I'll open up `src/Security/LoginFormAuthenticator.php` and add implements `PasswordAuthenticatedInterface`:

```
95 lines | src/Security/LoginFormAuthenticator.php
```

```
... lines 1 - 17
18 use Symfony\Component\Security\Guard\PasswordAuthenticatedInterface;
... lines 19 - 20
21 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements PasswordAuthenticatedInterface
22 {
23     ... lines 23 - 93
94 }
```

Basically, we need to tell the system what the plain-text password is. I'll scroll down and then go to the "Code"->"Generator" menu - or Command+N on a Mac - to generate the required `getPassword()` method:

95 lines | [src/Security/LoginFormAuthenticator.php](#)

... lines 1 - 20

```
21 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements PasswordAuthenticatedInterface
22 {
    ... lines 23 - 75
76     public function getPassword($credentials): ?string
77     {
    ... line 78
79     }
    ... lines 80 - 93
94 }
```

Look up at getCredentials():

95 lines | [src/Security/LoginFormAuthenticator.php](#)

... lines 1 - 20

```
21 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements PasswordAuthenticatedInterface
22 {
    ... lines 23 - 44
45     public function getCredentials(Request $request)
46     {
47         $credentials = [
48             'email' => $request->request->get('email'),
49             'password' => $request->request->get('password'),
50             'csrf_token' => $request->request->get('_csrf_token'),
51         ];
52
53         $request->getSession()->set(
54             Security::LAST_USERNAME,
55             $credentials['email']
56         );
57
58         return $credentials;
59     }
    ... lines 60 - 93
94 }
```

We return an array with the email, password, and csrf_token keys. In getPassword(), we're passed that array as the \$credentials argument. To get the password, return \$credentials['password']:

95 lines | [src/Security/LoginFormAuthenticator.php](#)

... lines 1 - 20

```
21 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements PasswordAuthenticatedInterface
22 {
    ... lines 23 - 75
76     public function getPassword($credentials): ?string
77     {
78         return $credentials['password'];
79     }
    ... lines 80 - 93
94 }
```

[UserRepository PasswordUpgraderInterface](#)

The *second* change we need to make is inside src/Repository/UserRepository.php. Implement a new interface here too called PasswordUpgraderInterface:

94 lines | [src/Repository/UserRepository.php](#)

... lines 1 - 7

```
8 use Symfony\Component\Security\Core\User\PasswordUpgraderInterface;
```

... lines 9 - 16

```
17 class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
```

```
18 {
```

... lines 19 - 92

```
93 }
```

This requires one new method. Go to the "Code"->"Generate" menu - or Command+N on a Mac - select "Implement Methods" and choose upgradePassword():

94 lines | [src/Repository/UserRepository.php](#)

... lines 1 - 8

```
9 use Symfony\Component\Security\Core\User\UserInterface;
```

... lines 10 - 16

```
17 class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
```

```
18 {
```

... lines 19 - 59

```
60 public function upgradePassword(UserInterface $user, string $newEncodedPassword): void
```

```
61 {
```

```
62
```

```
63 }
```

... lines 64 - 92

```
93 }
```

Here's the idea: when we log in, *if* the user's password is hashed with an old algorithm, the security system will call getPassword() on our authenticator to get the plain-text password and then hash it using the latest algorithm. To save that newly-hashed string to the user table, it will call this upgradePassword() method and pass it to us.

So, our job here is to update the database. I'll add a little PHPDoc above this method: we know the \$user variable will be *our* User object:

98 lines | [src/Repository/UserRepository.php](#)

... lines 1 - 16

```
17 class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
```

```
18 {
```

... lines 19 - 59

```
60 /**
```

```
61  * @param User $user
```

```
62  */
```

```
63 public function upgradePassword(UserInterface $user, string $newEncodedPassword): void
```

```
64 {
```

... lines 65 - 66

```
67 }
```

... lines 68 - 96

```
97 }
```

Now add \$user->setPassword(\$newEncodedPassword) and then \$this->getEntityManager()->flush(\$user):

98 lines | [src/Repository/UserRepository.php](#)

... lines 1 - 16

```
17 class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
18 {
    ... lines 19 - 59
60     /**
61      * @param User $user
62      */
63     public function upgradePassword(UserInterface $user, string $newEncodedPassword): void
64     {
65         $user->setPassword($newEncodedPassword);
66         $this->getEntityManager()->flush($user);
67     }
    ... lines 68 - 96
97 }
```

That's it! Test drive time! Find your browser and log out. Log back in with `admin1@thespacebar.com`, password engage. It works. But the *real* test is what the database looks like! Run that query again:

```
$ php bin/console doctrine:query:sql 'SELECT email, password FROM user'
```

Scroll up and... there it is! `admin0` still has the `bcrypt` format but `admin1` - the user we just logged in as - has an `argon`-hashed password!

So that's it! By adding a few lines of config and two simple methods, our existing users will be upgraded to the latest algorithm safely over time. And we can brag about this cool feature to our friends.

Next, we're *just* about done with our tour through my favorite new Symfony 5 features. But before we're done, I want to talk about PHP 7.4 preloading *and* a way to double-check that service wiring across your *entire* app is working correctly. Because, surprise! We have a hidden bug.

Chapter 28: PHP 7.4 preload

There are two last small - but cool - features I want to talk about.

[Huh? Preload?](#)

For the first, search for "Symfony preload" to find a blog post talking about it: "New in Symfony 4.4: Preloading Symfony Applications in PHP 7.4".


Here's the deal: in PHP 7.4 a new feature was added called "preloading". Basically, in your php.ini file, you can point an opcache.preload setting at a file that contains a list of all the PHP files that your application uses. By doing this, when PHP starts, it will "preload" those files into OPcache. You're effectively giving your web-server a "head" start: telling it to load the source code it will need into memory *now* so that it's ready when you start serving traffic.

What's the catch? Well, first, you need to create this "list of files", which we'll talk about in a minute. Second, each time these files change - so on each deploy - you need to restart your web server. And third, until PHP 7.4.2, this feature was a little buggy. It *should* be fine now, but there still could be some bugs left. Proceed with caution.

[The Generated Preload File](#)

So how does Symfony fit into this? Symfony knows a lot about your app, like which classes your app uses. And so, it can build that "preload" file automatically.

Check it out, at your terminal, clear the prod cache:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. The command prompt shows the command to clear the prod cache.

```
$ php bin/console cache:clear --env=prod
```

Now, in PhpStorm, check out the var/cache/prod/ directory... here it is: App_KernelProdContainer.preload.php. *This* file - which basically includes a bunch of classes - is a PHP 7.4 preload file. All *you* need to do is update the opcache.preload setting in php.ini to point to this file, restart your web server any time you deploy and, voilà! Instant performance boost!

How much of a boost? I'm not sure. It's such a new feature that benchmarks are only *starting* to be released. The blog post says 30 to 50%, I've seen other places saying more like 10 or 15%. Either way, if you can get your system set up to use it, free performance!

Next, let's talk about one last feature: a command you can run to make sure *all* your service wiring and type-hints are playing together nicely. Because in our app, there *is* a problem.

Chapter 29: Is your Container Running? Catch It!

lint:container

Symfony's service container is special... like *super-powers* special. Why? Because it's "compiled". That's a fancy way of saying that, instead of Symfony figuring out how to instantiate each service at runtime, when you build your cache, it figures out *every* argument to *every* service and dumps that info into a cache file - called the "compiled container". That's a major reason why Symfony is so fast.

But it has another benefit: if you misconfigured a service - like used a wrong class name - you don't have to go to a page that *uses* that service to notice the problem. Nope, *every* page of your app will be broken. That means less surprise bugs on production.

Another type of error that Symfony's container will catch immediately is a missing argument. For example, imagine you registered a service and forgot to configure an argument. Or, better example, Symfony couldn't figure out what to pass to this Mailer argument for some reason:

```
67 lines | src/Service/Mailer.php
... lines 1 - 12
13 class Mailer
14 {
... lines 15 - 19
20 public function __construct(MailerInterface $mailer, Environment $twig, Pdf $pdf, EntrypointLookupInterface $entrypointLookup)
21 {
... lines 22 - 25
26 }
... lines 27 - 65
66 }
```

If that happened, you'll get an error when the container builds... meaning that *every* page will be broken - even if a page doesn't use this service.

[Detecting Type Problems](#)

Starting in Symfony 4.4, Symfony can now *also* detect if the wrong *type* will be passed for an argument. For example, if we type-hint an argument with MailerInterface, but due to some misconfiguration, some *other* object - or maybe a string or an integer - will be passed here, we can find out immediately. But this type of problem won't break the container build. Instead, you need to *ask* Symfony to check for type problems by running:

```
$ php bin/console lint:container
```

And... oh! Woh! This is a perfect example!

```
Invalid definition for service nexy_slack.client. Argument 1 of Nexy\Slack\Client accepts a Psr ClientInterface,
HttpMethodsClient passed.
```

Apparently the container is configured to pass the wrong type of object to this service! This service comes from NexylanSlackBundle - I broke something when I upgraded that bundle... and didn't even realize it because I haven't navigated to a page that uses that service!

[Fixing our lint Problem](#)

After some digging, it turns out that the bundle has a *tiny* bug that allowed us to accidentally use a version of a dependency that is too old. Run:


```
$ composer why php-http/httpplug
```

I won't bore you with the details, but basically the problem is that this library needs to be at version 2 to make the bundle happy. We have version 1 and a few other libraries depend on it.

The fix is to go to composer.json and change the guzzle6-adapter to version 2:

```
105 lines | composer.json
1  {
    ... lines 2 - 3
4  "require": {
    ... lines 5 - 22
23  "php-http/guzzle6-adapter": "^2.0",
    ... lines 24 - 44
45  },
    ... lines 46 - 103
104 }
```

Why? Again, if you dug into this, you'd find that we need version 2 of guzzle6-adapter in order to be compatible with version 2 of httpplug... which is needed to be compatible with the bundle. Sheesh.

Now run composer update with all three of these libraries: php-http/httpplug, php-http/client-common - so that it can upgrade to a new version that allows version 2 of HTTPPlug - and guzzle6-adapter:

```
$ composer update php-http/httpplug php-http/client-common php-http/guzzle6-adapter
```

And... cool! Now run:

```
$ php bin/console lint:container
```

We get no output because *now* our container is happy. And because a few libraries had major version upgrades, if you looked in the CHANGELOGs, you'd find that we *also* need one more package to *truly* get things to work:

```
$ composer require http-interop/http-factory-guzzle
```

The point is: lint:container is a *free* tool you can add to your continuous integration system to help catch errors earlier. The more type-hints you use in your code, the more it will catch. It's a win win!

And..... that's it! We upgraded to Symfony 4.4, fixed deprecations, upgraded to Symfony 5, jumped into some of the *best* new features and, ultimately, I think we became friends. Can you feel it?

If you have any upgrade problems, we're here for you in the comments. Let us know what's going on, tell us a funny story, or leave us a Symfony 5 Haiku:

Reading your comments After a long weekend break Brings joy to keyboards

Alright friends, seeya next time!

