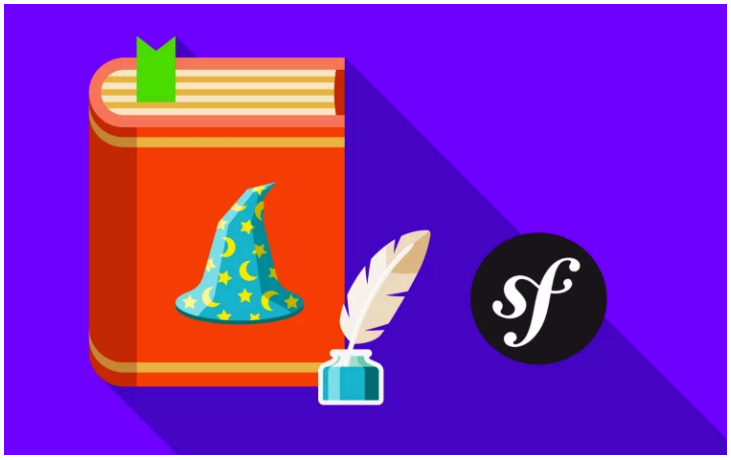
Upgrading & What's New in Symfony 6!



Chapter 1: Project Setup & The Plan

Hey friends! If you're like me, you probably have a Symfony 5 project - or 10 - lying around just *waiting* to get upgraded to Symfony 6. Well... you've come to the right place! That's *exactly* what we're going to do in this tutorial! But *more* than that! This is a *particularly* interesting upgrade, because it also involves updating our code to use PHP 8. And *that* includes a transformation from using *annotations* to PHP 8 *attributes*. I need to find my monocle, because we're getting *fancy*. It also includes several other PHP 8 features, which you're *really* going to like. Plus, for the first time, we're going to use a tool called "Rector" to automate as much of this as possible. And... because I just *can't* help myself, we'll discover nice new Symfony 6 features along the way.

Getting the Project Running

All right! To get this upgrade party started, you should *definitely* code along with me. Download the course code from this page and unzip it to find a `start/` directory with the same code you see here. Follow this `README.md` file for all the setup goodies. I've already followed most of these steps... but I still need to build my Webpack Encore assets and start a web server. So let's do that!

Over in my terminal (this is already inside the project), run

```
yarn install
```

or

```
npm install
```

to download the Node packages. I want to get this running properly because we're going to *upgrade* some of our JavaScript tools a bit later.

Then run:

```
yarn watch
```

or

```
npm run watch
```

to build the frontend assets... and then watch for changes.

For the last step: open a new terminal tab and get a local web server started. I'm going to use the Symfony server like normal by running:

```
symfony serve -d
```

And... awesome! That starts a new web server at https://127.0.0.1:8000. I'll click that and say... "Hello" to Cauldron Overflow! My old friend! This is the site we've been building throughout our Symfony 5 series. And if you check its `composer.json` file... and look down here for Symfony stuff... whoa.. it is *old*. All of the main Symfony libraries are version "5.0". That was *ages* ago. I was so young then!

The Plan

Here's our upgrade strategy. Step one: we're going to upgrade our project to Symfony 5.4. That's safe to do because Symfony doesn't include any backwards compatibility breaks on *minor* version upgrade. So anytime you upgrade *just* this middle number - called the "minor" number, like 5.0 to 5.4 - that's always going to be safe.

Step two: once we're on Symfony 5.4, to prepare our code for Symfony 6, all we need to do is hunt down and fix all of the deprecations in our code. Once we've fixed those, it will be safe to go to Symfony 6. To *find* those deprecations, we're going to use a few tools, like "Rector" to upgrade parts of our code, the new recipes update system and the tried-and-true Symfony "deprecations reporting".

After *all* of that, once we have a Symfony 5.4 project with no deprecations... we can just "flip the switch" and upgrade to Symfony 6. Easy peasy!

And at the *very* end, we'll cover a few more new features that you might like. Are you ready? Great! Let's upgrade our site to Symfony 5.4 next.

Chapter 2: Upgrading to Symfony 5.4

Step one to upgrading our app to Symfony 6 is to upgrade all of the Symfony libraries to 5.4. And... that's pretty easy: it's just a composer thing.

Tweaking the Composer Version Constraints

In `composer.json`, we have *quite a few* libraries that start with `symfony/`. *Most* of these are part of the "main" Symfony project and they follow Symfony's familiar versioning, with versions like 5.0, 5.1, up to 5.4 and then 6.0. *Those* are the packages that we're going to focus on upgrading.

But a few of these, like `symfony/maker-bundle`, follow their *own* versioning scheme. What a diva! We're not going to worry about upgrading those right now, but we *will* make sure that, by the end, we've upgraded everything.

Okay, what we need to do is change all of these `5.0.*` to `5.4.*`. I'm going to do a "Find & Replace" to replace `5.0.*` with `5.4.*`. Hit "Replace All".

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 21
        "symfony/asset": "5.4.*",
        "symfony/console": "5.4.*",
        "symfony/dotenv": "5.4.*",
// ... line 25
        "symfony/form": "5.4.*",
        "symfony/framework-bundle": "5.4.*",
// ... line 28
        "symfony/property-access": "5.4.*",
        "symfony/property-info": "5.4.*",
        "symfony/proxy-manager-bridge": "5.4.*",
        "symfony/routing": "5.4.*",
        "symfony/security-bundle": "5.4.*",
        "symfony/serializer": "5.4.*",
        "symfony/stopwatch": "5.4.*",
        "symfony/twig-bundle": "5.4.*",
// ... line 37
        "symfony/validator": "5.4.*",
// ... lines 39 - 44
    },
    "require-dev": {
// ... line 47
        "symfony/debug-bundle": "5.4.*",
// ... lines 49 - 51
        "symfony/web-profiler-bundle": "5.4.*",
// ... line 53
    },
// ... lines 55 - 100
    "extra": {
        "symfony": {
            "allow-contrib": false,
            "require": "5.4.*"
        }
    }
}
```

Nice! And notice that, in addition to the packages themselves, we also needed to change the `extra.symfony.require` key. This is a performance optimization from Flex: it basically makes sure that Flex only considers Symfony packages that match this version. Just make sure that you don't forget to update it.

Ok... let's see. This updated *a lot* of libraries. To make sure we didn't miss anything, search for `symfony/`... and scroll down a bit. The `monolog-bundle` has its own versioning, so that's ok. But, ooh... I *did* miss one: `symfony/routing`. For *some* reason, this was already at Symfony 5.1. So let's change that to `5.4.*` as well.

And... everything else looks okay: each is changed to `5.4.*` or it has its own versioning strategy... and we're not going to worry about it right now.

Updating the Dependencies

To *actually* update these, over at your terminal, we *could* try to upgrade *just* the Symfony packages with:

```
composer up 'symfony/*'
```

There's a good chance that's going to fail... because in order to upgrade all of the Symfony packages, some *other* package will need to be upgraded, like `symfony/proxy-manager-bridge`. If you wanted to, you could add *that* to the `composer up` command... or add the `-W` flag, which tells Composer to upgrade all of the `symfony/` libraries *and* their dependencies.

But... I'm going to upgrade *everything* with:

```
composer up
```

Look: in our `composer.json` file, the version constraints on all of the packages (Symfony *and* other libraries) are really good! They allow *minor* version updates, like 4.0 to 4.1, but they don't allow *major* version updates. So if there were a new version *5* of this library, running `composer up` would *not* upgrade to that new major version.

In other words, updating should only upgrade *minor* versions... and those, in theory, won't contain any breaks. So let's do this:

```
composer up
```

And... hello upgrades! Wow! Look at that huge list! Lots of Symfony stuff... but plenty of other libraries too.

Ok, so that was a *big* upgrade. Does the site still work? I don't know! Head over, refresh and... it does! Symfony is amazing!

Checking out the Deprecations

Now that we're on Symfony 5.4, we can see the full list of deprecated code paths that we hit when rendering this page. *Your* number will vary... and the number might even change when you refresh the page... that's due to some pages using cache. It looks like I have about 71 deprecations.

If you click into this, *so* cool. We can see what all of those are.

So at this point, our job is simple... but not necessarily *easy*. We need to hunt down every single one of these deprecations, figure out what code needs to change, and then make that change. Some of these will be pretty obvious... and some of them *won't*.

So before we even *attempt* to hunt them down manually, let's... do something more automatic. We're programmers right! Let's use a tool called Rector to automate as many changes to our code as possible. That's next.

Chapter 3: Automating Upgrades with Rector

Now that we're on Symfony 5.4, our job is simple: hunt down and update all of our deprecated code. As soon as we do that, it will be safe to upgrade to Symfony 6. That's because the *only* difference between Symfony 5.4 and 6.0 is that all the deprecated code paths are removed.

Fortunately, Symfony is amazing and tells us - via the web debug toolbar - *exactly* what code is deprecated. But understanding what all of these mean... isn't always easy. So before we even try, we're going to automate as much of this as possible. And we're going to do that with a tool called Rector.

Installing Rector

Head to https://github.com/rectorphp/rector. This is an *awesome* command-line tool with *one* job: to automate all *sorts* of upgrades to your code, like upgrading your code from Symfony 5.0 compatible code to Symfony 5.4 compatible code. *Or* upgrading your code to be PHP 8 compatible. It's a powerful tool... and if you want to learn more about it, they even released a book where you can go deeper... and also help support the project.

All right, let's get this thing installed! Head over to your terminal and run:

```
composer require rector/rector --dev
```

Beautiful! In order for Rector to work, it needs a config file. And we can bootstrap one by running rector with:

Tip

In newer versions of Rector, instead of `./vendor/bin/rector init`, just run `./vendor/bin/rector` to do the same thing.

```
./vendor/bin/rector init
```

Awesome! That creates the `rector.php` file... which we can see over at the root of our project.

Tip

The latest version of Rector will generate config that looks a bit different than this. But don't worry, it still works exactly the same.

Inside of this callback function, our job is to configure which *types* of upgrades we want to apply. These are called "rules" or sometimes "set lists" or rules. We're going to start with a set of Symfony upgrades.

rector.php

```
// ... lines 1 - 9
return static function (ContainerConfigurator $containerConfigurator): void {
    // get parameters
    $parameters = $containerConfigurator->parameters();
    $parameters->set(Option::PATHS, [
        __DIR__ . '/src'
    ]);

    // Define what rule sets will be applied
    $containerConfigurator->import(LevelSetList::UP_TO_PHP_74);

    // get services (needed for register a single rule)
    // $services = $containerConfigurator->services();

    // register a single rule
    // $services->set(TypedPropertyRector::class);
};
```

Configuring Rector for the Symfony Upgrade

If you look back at the documentation, you'll see a link to a Symfony repository where it tells you about a bunch of Symfony "rules" - fancy word for "upgrades" - that they've already prepared! That was nice of them!

Tip

The config on this page will now look different than in the video. But, it still works the same. Copy the latest version into your app.

Below, copy the inside of their callback function... and paste it over what we have.

rector.php

```
// ... lines 1 - 7
use Rector\Symfony\Set\SymfonySetList;
use Symfony\Component\DependencyInjection\Loader\Configurator\ContainerConfigurator;
// ... line 10
return static function (ContainerConfigurator $containerConfigurator): void {
    // region Symfony Container
    $parameters = $containerConfigurator->parameters();
    $parameters->set(
        Option::SYMFONY_CONTAINER_XML_PATH_PARAMETER,
        __DIR__ . '/var/cache/dev/App_KernelDevDebugContainer.xml'
    );
    // endregion

    $containerConfigurator->import(SymfonySetList::SYMFONY_52);
    $containerConfigurator->import(SymfonySetList::SYMFONY_CODE_QUALITY);
    $containerConfigurator->import(SymfonySetList::SYMFONY_CONSTRUCTOR_INJECTION);
};
```

This points Rector to a cache file that helps it do its job... and most importantly, it tells Rector that we want to upgrade our code to be Symfony 5.2 compatible, as well as upgrade our code to some Symfony code quality standards and "constructor" injection. If you want to know more about what these do, you could follow the constants to check out the code.

But, wait, we don't want to upgrade our code to Symfony 5.2! We want to upgrade it *all* the way to Symfony 5.4. You might expect me to just put "54" here. And we *could* do that. But instead, I'm going to use `SymfonyLevelSetList::UP_TO_SYMFONY_54`. Oh... it looks like I also need to add a `use` statement for `SymfonySetList::`. Let me retype that, hit "tab" and... great!

rector.php

```
↕ // ... lines 1 - 7
8  use Rector\Symfony\Set\SymfonyLevelSetList;
9  use Rector\Symfony\Set\SymfonySetList;
↕ // ... lines 10 - 11
12 return static function (ContainerConfigurator $containerConfigurator): void {
↕ // ... lines 13 - 20
21     $containerConfigurator->import(SymfonyLevelSetList::UP_TO_SYMFONY_54);
↕ // ... lines 22 - 23
24 };
```

Anyways. We need to upgrade our code from 5.0 to 5.1... then 5.1 to 5.2.. and so on *up* to Symfony 5.4. That's what `UP_TO_SYMFONY_54` means: it will include *all* of the "rules" for upgrading our code to 5.1, 5.2, 5.3 and finally 5.4.

And... that's it! We're ready to run this. But before we do, I'm curious what changes this will make. So let's add all of the changes to git... and commit. Perfect!

Running Rector

To run Rector, say `./vendor/bin/rector process src/`. We could also point this at the `config/` or `templates/` directories... but the vast majority of the changes it will make apply to our classes in `src/`:

```
vendor/bin/rector process src/
```

And... it's working! Awesome! Eight files were changed by Rector. Let's scroll to the top. This is cool: it shows you the file that was changed, the actual *change* and, below, which *rules* caused that change.

One modificiation it made is `UserPasswordEncoderInterface` to `UserPasswordHasherInterface`. That's a good change: the old interface is deprecated in favor of the new one. It also changed `UsernameNotFoundException` to `UserNotFoundException`. Another good, low-level update to some deprecated code.

There was also a change to a class in `Kernel`... and a few other similar things. Near the bottom, the Symfony code quality set list added a `Response` return type to every controller. That's optional... but nice!

So it didn't make a *ton* of changes, but it *did* fix a few deprecations without us needing to do *anything*.

Though... it's not perfect. One problem is that, sometimes, Rector will mess with your coding style. That's because Rector doesn't really understand what your coding style is... and so it doesn't even try. But that's by design and will be easy to fix.

Second, while it *did* change the interface from `UserPasswordEncoderInterface` to `UserPasswordHasherInterface`, it inlined the *whole* class name... instead of adding a `use` statement.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 11
12 use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
↕ // ... lines 13 - 24
25 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements
   PasswordAuthenticatedInterface
26 {
↕ // ... lines 27 - 36
37     public function __construct(SessionInterface $session, EntityManagerInterface
   $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
   $csrfTokenManager, \Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface
   $passwordEncoder)
38     {
↕ // ... lines 39 - 43
44     }
↕ // ... lines 45 - 119
120 }
```

And third, it didn't change any variable names. So even though it changed this argument to `UserPasswordHasherInterface`, the argument is still called `$passwordEncoder`... along with the property. Worse, the `UserPasswordHasherInterface` has a different *method* on it... and it didn't update the code down here to *use* that new method name.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 24
25 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements
   PasswordAuthenticatedInterface
26 {
↕ // ... lines 27 - 34
35     private $passwordEncoder;
↕ // ... line 36
37     public function __construct(SessionInterface $session, EntityManagerInterface
   $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
   $csrfTokenManager, \Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface
   $passwordEncoder)
38     {
↕ // ... lines 39 - 42
43         $this->passwordEncoder = $passwordEncoder;
44     }
↕ // ... lines 45 - 119
120 }
```

So Rector is a *great* starting point to catch a bunch of changes. But we're going to need to take what we've found and finish the job. Let's do that next. We'll do part of that by hand... but a lot of it automatically with PHP CS Fixer.

Chapter 4: Post-Rector Cleanups & Tweaks

Rector just automated several changes to our app that are needed to remove deprecations on Symfony 5.4. *Plus* it did some bonus refactoring, like adding the optional `Response` return type on our controllers.

But as nice as that is, it's not *perfect*. All the class names are inlined, instead of having a `use` statement. And even though it renamed some interfaces, it didn't rename the methods that we *call* on those objects to reflect the change. No need to worry, though. Rector gave us a great start and helped highlight several changes that we need to make. Now, let's finish the job.

Installing php-cs-fixer

First, for these long class names with no `use` statement and, in general for coding styles, Rector doesn't know what coding style we prefer, so it doesn't even try to format things correctly. The official recommendation is to use a tool in your project like PHP CS Fixer to reformat the code *after* running Rector. PHP CS Fixer is a great tool anyway... so let's get it installed so it can help us along our journey.

You can install PHP CS Fixer a few different ways, but oddly enough, the *recommended* way - and the way that I like - is to install it via Composer into its own directory. Run:

```
mkdir -p tools/php-cs-fixer
```

There's nothing special here: just a new `tools/` directory with `php-cs-fixer/` inside. Now install it *into* that directory by running `composer require --working-dir=tools/php-cs-fixer` - that tells Composer to behave like I'm running it from inside of `tools/php-cs-fixer` - and then `friendsofphp/php-cs-fixer`.

If you're wondering why we're not just installing this directly into our *main* `composer.json` dependencies, well... that's a bit tricky. PHP CS Fixer is a standalone executable tool. If I install it into our *app's* dependencies, then it could cause problems if some of *its* dependencies don't match versions that we have already in our app. Well, really, this is a potential problem whenever you install *any* library. But since all we need is a standalone binary... there's no reason to mix it into our app. We could have done the same thing with Rector.

This gives us, in that directory, `composer.json` and `composer.lock` files. And in *its* `vendor/bin` directory... yes: `php-cs-fixer`. *That's* the executable.

And because we have a new `vendor/` directory, open up the root `.gitignore` file and, at the bottom, *ignore* that: `/tools/php-cs-fixer/vendor`. And while we're here, let's also ignore `/.php-cs-fixer.cache`. That's a cache file that PHP CS Fixer will create when it does its work.

.gitignore

```
// ... lines 1 - 23
/tools/php-cs-fixer/vendor/
/.php-cs-fixer.cache
```

Adding php-cs-fixer Config

The *last* thing we need to do is add a config file. Up here, create a new file called `.php-cs-fixer.php`. Inside, I'm going to paste about 10 lines of code. This is pretty simple. It tells PHP CS Fixer where to find our `src/` files... then, below, which *rules* to apply. I'm using a pretty standard Symfony set of rules.

.php-cs-fixer.php

```
// ... lines 1 - 2
$finder = PhpCsFixer\Finder::create()
    ->in(__DIR__.'/src')
;

$config = new PhpCsFixer\Config();
return $config->setRules([
        '@Symfony' => true,
        'yoda_style' => false,
    ])
    ->setFinder($finder)
;
```

And... we're ready to run this! To see what it does, over at the command line, add all the changes to git with:

```
git add .
```

Then check on them:

```
git status
```

But don't *commit* them yet. I still want to be able to review the changes that Rector made before we *finally* commit. But at least, *now*, we'll be able to see what PHP CS Fixer does.

Let's run it:

```
./tools/php-cs-fixer/vendor/bin/php-cs-fixer fix
```

And... nice! It modified 6 files. Let's check them out!

```
git diff
```

Awesome! It removed the *long* class names for `Response` across our entire codebase! It also deleted a few old `use` statements that we don't need. So the code from Rector still isn't perfect, but that was a nice step towards making it better!

Fixing the Password Hasher Code

For the final fixes, we'll do them manually by digging into the changes that Rector made, one by one. I'll help out by zooming us into the places that need updates.

The first is `RegistrationController`: `src/Controller/RegistrationController.php`. This is one of the places where it changed `UserPasswordEncoderInterface` to `UserPasswordHasherInterface`. Notice that PHP CS Fixer *did* fix a lot of the *long*, inlined class names... but not *all* of them. It depends on if there was already a `use` statement for that class or not.

So let's fix this by hand. Hover over the class, hit "alt" + "enter" and then go to "Simplify FQN". That shortens it and adds the `use` statement on top.

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 11
12 use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
↕ // ... lines 13 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 21
22     public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
   VerifyEmailHelperInterface $verifyEmailHelper): Response
↕ // ... lines 23 - 99
100 }
```

But there's another problem. If we trace down to where this is used, previously we were calling `->encodePassword()`. But... that method doesn't exist on the new interface! We need to call `->hashPassword()`.

I'm also going to rename the argument. Go to "Refactor" then "Rename" and call it `$userPasswordHasher`... just because that's a more fitting name.

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 21
22     public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
   VerifyEmailHelperInterface $verifyEmailHelper): Response
23     {
↕ // ... lines 24 - 27
28         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... line 29
30             $user->setPassword(
31                 $userPasswordHasher->hashPassword(
↕ // ... lines 32 - 34
35             );
↕ // ... lines 36 - 55
56         }
↕ // ... lines 57 - 60
61     }
↕ // ... lines 62 - 101
```

Next up is `src/Factory/UserFactory.php` for the *same* change. Scroll down and... once again, we have a long class name. Hit "alt" + "enter" and go to "Simplify FQN" to add that `use` statement. Then... let's "Refactor" and "Rename" the argument to `$passwordHasher`... good... and "Refactor", "Rename" the property *also* to `$passwordHasher`.

src/Factory/UserFactory.php

```
// ... lines 1 - 6
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
// ... lines 8 - 29
final class UserFactory extends ModelFactory
{
    private UserPasswordHasherInterface $userPasswordHasher;

    public function __construct(UserPasswordHasherInterface $userPasswordHasher)
    {
// ... lines 36 - 37
        $this->userPasswordHasher = $userPasswordHasher;
    }
// ... lines 40 - 68
}
```

Finally, below, we need to call `->hashPassword()` instead of `->encodePassword()`.

src/Factory/UserFactory.php

```
// ... lines 1 - 50
    protected function initialize(): self
    {
// ... line 53
        return $this
            ->afterInstantiate(function (User $user) {
                if ($user->getPlainPassword()) {
                    $user->setPassword(
                        $this->userPasswordHasher->hashPassword($user, $user->getPlainPassword())
                    );
                }
            })
        ;
    }
// ... lines 64 - 70
```

Done!

Just *one* more spot where we need this same change: `src/Security/LoginFormAuthenticator.php`. We're going to refactor this class later to use the new security system... but let's at least get it working. Find the `UserPasswordHasherInterface` argument, shorten that with "Simplify FQN"... then rename the argument to `$passwordHasher`... and rename the property to `$passwordHasher`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
↕ // ... lines 10 - 24
25 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements
   PasswordAuthenticatedInterface
26 {
↕ // ... lines 27 - 34
35     private $userPasswordHasher;
36
37     public function __construct(SessionInterface $session, EntityManagerInterface
   $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
   $csrfTokenManager, UserPasswordHasherInterface $userPasswordHasher)
38     {
↕ // ... lines 39 - 42
43         $this->userPasswordHasher = $userPasswordHasher;
44     }
↕ // ... lines 45 - 119
120 }
```

Then we check to see where this is used... I'll search for "hasher"... there we go! Down on line 84, the `->isPasswordValid()` actually *does* exist on the new interface, so this is one case where we *don't* need to change anything else.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 14
15 use Symfony\Component\Security\Core\Exception\UserNotFoundException;
↕ // ... lines 16 - 24
25 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements
   PasswordAuthenticatedInterface
26 {
↕ // ... lines 27 - 66
67     public function getUser($credentials, UserProviderInterface $userProvider)
68     {
↕ // ... lines 69 - 75
76         if (!$user) {
77             throw new UserNotFoundException('Email could not be found.');
78         }
↕ // ... lines 79 - 80
81     }
↕ // ... lines 82 - 119
120 }
```

Oh, but while we're in here, the `UserNotFoundException` is another *long* class name. Hit "Simplify FQN" again.

Beautiful! That should be everything.

The big question *now* is: does our app work? If we go back to the Homepage... it *doesn't*. We're back on the Welcome to Symfony page? That's weird...

Spin back over to your terminal and run:

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

Wow. In fact, *all* of our routes are *gone*. This is due to one other change that Rector made that we need to pay close attention to. It's inside of our `Kernel` class. We're going to talk more about this class later when we upgrade our recipes. Rector changed the argument to `RoutingConfigurator`, but it didn't update the code below.

src/Kernel.php

```
// ... lines 1 - 10
class Kernel extends BaseKernel
{
// ... lines 13 - 44
    protected function configureRoutes(\Symfony\Component\Routing\Loader\Configurator\RoutingConfigurator $routes): void
    {
        $confDir = $this->getProjectDir().'/config';

        $routes->import($confDir.'/{routes}/'.$this->environment.'/*'.self::CONFIG_EXTS, '/', 'glob');
        $routes->import($confDir.'/{routes}/*'.self::CONFIG_EXTS, '/', 'glob');
        $routes->import($confDir.'/{routes}'.self::CONFIG_EXTS, '/', 'glob');
    }
}
```

So again, Rector is really good for finding some of these changes, but you should always double-check the final result.

Fortunately, the entire `configureRoutes()` method has been moved into this `MicroKernelTrait` - a fact I'll talk about more soon. So we don't need this method in our class at *all* anymore. As soon as we delete it, the correct version from the trait is used... our routes are back.... and the page works! Woohoo!

And *hopefully* we have a *few* less deprecations than before. I now see 58. Progress!

So what's next? We've upgraded our dependencies and automated *some* of the changes we need with Rector. Well, there's still *one more* thing we can do before we start going through each deprecation manually: updating our *recipes*. And this has gotten a *whole* heck of a lot easier than the last time you upgraded. Let's find out how next.

Chapter 5: Updating the All-Important FrameworkBundle Recipe

At your terminal, run:

```
composer recipes
```

As you probably know, whenever we install a new package, that package *may* come with a recipe that does things like add configuration files, modify certain files like `.env`, or add other files. Over time, Symfony makes *updates* to these recipes. Sometimes these are minor... like the addition of a comment in a config file. But other times, they're bigger, like renaming config keys to match changes in Symfony itself. And while you don't *have* to update your recipes, it's a great way to keep your app feeling like a standard Symfony app. It's also a free way to update deprecated code!

Hello recipes:update

Until recently, updating recipes was a *pain*. If you're not familiar, just check our "Upgrade to Symfony 5" tutorial! Yikes. *But* no more! Starting with Symfony Flex 1.18 or 2.1, Composer has a proper `recipes:update` command. It literally *patches* your files to the latest version... and it's *awesome*. Let's try it!

Run:

```
composer recipes:update
```

Oh! Before we run this, it tells us to commit everything that we've been working on. Great idea! I'll say that we are:

> *"upgrading some code to Symfony 5.4 with Rector"*

```
git add .
git commit -m "upgrading some code to Symfony 5.4 with Rector"
```

Perfect! Try the `recipes:update` command again. The reason it wants our working copy to be clean is because it's about to patch some files... which *might* involve conflicts.

Let's start with `symfony/framework-bundle`, because this is the *big* one. The *most* important files in our project come from this recipe. I'll hit `4`, clear the screen, and go!

Behind the scenes, this checks to see what the recipe looked like when we *originally* installed it, compares it to what the recipe looks like now, and generates a *diff* that it then applies to our project. In some cases, like this one, that can cause some conflicts, which is pretty cool. The best part might be that it generates a changelog containing *all* the pull requests that contributed to these updates. If you need to figure out *why* something changed, this will be your friend.

Oh, but creating the changelog requires making a *bunch* of API calls to GitHub. So it's *possible* that composer will ask you for a personal access token, like it just did for me. In some *rare* cases with a giant recipe like `framework-bundle`, if your recipe is really, *really* old, you might get this message even if you *have* given an access token to Composer. If that happens, just wait for 1 minute... then re-enter your access token. Congratulations, you just hit GitHub's per-minute API limit.

Anyways, *there's* the CHANGELOG. It's not usually that long, but this recipe is the most important and... well... it was *horribly* out-of-date. Oh, and if you have a trendy terminal like me - this is iTerm - you can click these links to jump directly into the pull request, which will live at https://github.com/symfony/recipes.

Changes to .env

Alright, let's walk through the changes this made. This is the *biggest* and most important recipe, so I want to cover everything.

Since I've already done my homework, I'll clear the changelog and run:

```
git status
```

Woh. It made a *bunch* of changes, including three conflicts. Fun! Let's go through those first. Move over and start inside `.env`. Let's see: apparently the recipe *removed* these `#TRUSTED_PROXIES` and `#TRUSTED_HOSTS` lines.

.env

```
↕ // ... lines 1 - 15
16 ###> symfony/framework-bundle ###
↕ // ... lines 17 - 18
19 #TRUSTED_PROXIES=127.0.0.0/8,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16
20 #TRUSTED_HOSTS='^(localhost|example\.com)$'
21 ###

↕ // ... lines 22 - 32
```

Both of these are now set in a config file. And while you *could* still set an environment variable and reference it from that config file, the recipe no longer ships with these comments. I'm not sure why this caused a conflict, but let's delete them.

Changes to services.yaml

The next conflict is up in `config/services.yaml`. This one is pretty simple. This is *our* config and below, the *new* config. The recipe removed the `App\Controller\` entry. This... was never needed unless you make super-fancy controllers that do *not* extend `AbstractController`. It was removed from the recipe for simplicity. It also looks like the updated recipe reformats the `exclude` onto multiple lines, which is nice. So let's take their version entirely.

config/services.yaml

```
// ... lines 1 - 8
services:
// ... lines 10 - 18
    App\:
        resource: '../src/'
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'
// ... lines 25 - 30
```

Changes to src/Kernel.php

The final conflict is in `src/Kernel.php`... where you can see that *our* side has a bunch of code in it... and their side has nothing.

Remember how I mentioned that `configureRoutes()` was moved into `MicroKernelTrait`? Well it turns out that *all* of these methods were moved into `MicroKernelTrait`. So unless you have some custom logic - which is pretty rare - you can delete everything.

src/Kernel.php

```
// ... lines 1 - 7
class Kernel extends BaseKernel
{
    use MicroKernelTrait;
}
```

Ok, back at the terminal, let's add those three files:

```
git add .env config/services.yaml src/Kernel.php
```

And then run

```
git status
```

to see what else the recipe update did.

Updated public/index.php, deleted bootstrap.php!

Interesting. It deleted `config/bootstrap.php` and modified `public/index.php`. Those are related. Look at the diff of `index.php`:

```
git diff --cached public/index.php
```

This file *used* to require `config/bootstrap.php`. And *that* file's job was to read and set up all the environment variables:

```
git diff --cached config/
```

Let's go check out the *new* `public/index.php`. Here it is. *Now* this requires some `vendor/autoload_runtime.php`. And the file is much shorter than before. What we're seeing is Symfony's new Runtime component in action.

public/index.php

```
// ... lines 1 - 4
require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

You can check out its introduction blog post to learn more about it.

Basically, the job of booting up Symfony and loading all of the environment variables was extracted *into* the runtime component. But... we don't actually have that component installed yet... which is why, if we try to refresh the page, we're gonna have a bad time:

> *"Failed to open `autoload_runtime.php`."*

To fix this, head over to your terminal and run:

```
composer require symfony/runtime
```

This package includes a Composer plugin... so it's going to ask us if we trust it. Say "yes". Then it installs... and promptly explodes when it tries to clear the cache! Ignore that for now: we'll fix it in a few minutes. It involves updating *another* recipe.

But if we try our site... it works!

New Environment-Specific Configuration

Ok, we're almost done! Back at the terminal, let's see what else changed:

```
git status
```

Notice that it deleted `config/packages/test/framework.yaml`, but *modified* `config/packages/framework.yaml`. This is probably *the* most common change that you'll see when you update your recipes today.

Open `config/packages/framework.yaml`. At the bottom... there's a new `when@test` section.

config/packages/framework.yaml

```
// ... lines 1 - 19
when@test:
    framework:
        test: true
        session:
            storage_factory_id: session.storage.factory.mock_file
```

Starting in Symfony 5.3, you can now add environment-specific config using this syntax. This configuration *used* to live inside of `config/packages/test/framework.yaml`. But for simplicity, the recipe *deleted* that file and just moved that config to the bottom of *this* file.

Back at the terminal, diff that file... it's hiding two other changes:

```
git diff --cached config/packages/framework.yaml
```

The recipe also changed `http_method_override` to `false`. That disables, by default, a feature that you probably weren't using anyways. It *also* set `storage_factory_id` to `session.storage.factory.native`. This has to do with how your session is stored. Internally, the key changed from `storage_id` to `storage_factory_id`, and it *should* now be configured.

Environment-Specific Routing Config

Back at the terminal, let's look at the final changes:

```
git status
```

Speaking of environment-specific config, you can do that same trick with routing files. See how it deleted `config/routes/dev/framework.yaml`, but *added* `config/routes/framework.yaml`? If we open up `config/routes/framework.yaml`, yup! It has `when@dev` and it imports the routes that allow us to test our error pages.

config/routes/framework.yaml

```
when@dev:
    _errors:
        resource: '@FrameworkBundle/Resources/config/routing/errors.xml'
        prefix: /_error
```

This is yet another example of the recipe moving configuration out of the environment directory and into the main configuration file... just for simplicity.

The new preload.php File

Finally, the recipe added a `config/preload.php` file. This one is pretty simple, and it leverages PHP's preloading functionality.

config/preload.php

```
// ... lines 1 - 2
if (file_exists(dirname(__DIR__).'/var/cache/prod/App_KernelProdContainer.preload.php')) {
    require dirname(__DIR__).'/var/cache/prod/App_KernelProdContainer.preload.php';
}
```

Essentially, on production, if you point your `php.ini`, `opcache.preload` at this file, you'll get a free performance boost! It's *that* simple. Well... *mostly* that simple. The only other thing you need to do is restart your web server on every deploy... or PHP-FPM if you're using that. We leverage this at SymfonyCasts for a little extra performance boost.

And... phew! The biggest recipe update is *done*. So let's add everything and commit. Because next, *more* recipe updates! But with FrameworkBundle behind us, the rest will be easier and faster.

Chapter 6: Recipe Upgrades with recipes:update

Let's keep upgrading recipes! There are a *bunch* of them to do, but most of these will be easy. We'll move quickly, but I'll highlight any important changes as we go.

symfony/console Recipe Update

For the next update, let's skip down to `symfony/console` since that's another important one.

```
composer recipes:update symfony/console
```

This updated just one file: `bin/console`. Check out the changes with:

```
git diff --cached bin/console
```

Hmm. It changed from being *kind of* long to... pretty darn short! This is, once again, the Symfony Runtime component in action. The code to boot up Symfony for the console has moved into `symfony/runtime`. And... this fixed our `bin/console` command, which had been broken since we upgraded the FrameworkBundle recipe.

Let's commit this change... and keep going:

```
composer recipes:update
```

symfony/twig-bundle Recipe

Skip down to `symfony/twig-bundle`. That's number `7`. I'll clear the screen and... okay! We have conflicts. *Exciting*! I'll clear the changelog since I've already looked at it. Ok, this deleted an environment-specific config file... and then we have two conflicts. Let's go check out `config/packages/twig.yaml`.

Once again, we're seeing the new environment-specific config feature. This `when@test` stuff *used to* live in `config/packages/test/twig.yaml`, but it's now been moved here. And because I have a custom `form_themes` config, it conflicted. We want to keep both things.

config/packages/twig.yaml

```
1  twig:
↕  // ... line 2
3      form_themes: ['bootstrap_4_layout.html.twig']
4
5  when@test:
6      twig:
7          strict_variables: true
```

The second conflict is in `templates/base.html.twig`. Our `base.html.twig` is pretty customized, so we likely don't need to worry about any new changes. The recipe added a new `favicon` by default. You probably *won't* use this since you'll have your own. To fix this conflict, since my project *doesn't* have a `favicon` yet, I'll copy the new stuff, use *our* code, but paste the `favicon`.

templates/base.html.twig

```
↕  // ... lines 1 - 2
3      <head>
↕  // ... line 4
5          <title>{% block title %}Welcome!{% endblock %}</title>
6          <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22
   viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size=%2296%22>â«ï¸</text></svg>">
↕  // ... lines 7 - 14
15     </head>
↕  // ... lines 16 - 95
```

Perfect! Now we can commit everything.

doctrine/doctrine-bundle Recipe Update

Let's keep going!

```
composer recipes:update
```

We'll work on the rest from top to bottom. Next is `doctrine/doctrine-bundle`. This is a cool update. Once again, I'll clear the screen and run:

```
git status
```

It conflicted inside the `.env` file... which is probably the *least* interesting change. Recently, DoctrineBundle's recipe started shipping with PostgreSQL as the default database. You can *totally* change that to be whatever you want, but PostgreSQL is *such* a good database engine that we started shipping with *it* as the default suggestion.

But I'm using MySQL in this project, so I'm going to keep that. But to be *super* cool, I'll at least take their new example config... which looks a *little* different... and update my comments on top with it. Then I'll use *my* version of the conflict. The end-result is a few tweaks to the comments, but nothing else.

.env

```
// ... lines 1 - 24
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"
# DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7&charset=utf8mb4"
# DATABASE_URL="postgresql://symfony:ChangeMe@127.0.0.1:5432/app?serverVersion=13&charset=utf8"
DATABASE_URL="mysql://root@127.0.0.1:3306/symfony6_upgrade?serverVersion=5.7"
// ... lines 29 - 30
```

The other changes from the recipe relate to the config files, and I bet you can see what's happening. It deleted two environment-specific config files and updated the main one. Hmm.

Open `config/packages/doctrine.yaml`. Sure enough, at the bottom, we see `when@test` and `when@prod`. That's nice! Everything is now in one file. Just make sure that if you had any custom config in the *old* deleted, files, that you move it over to *this* file.

config/packages/doctrine.yaml

```
// ... lines 1 - 18
when@test:
    doctrine:
        dbal:
            # "TEST_TOKEN" is typically set by ParaTest
            dbname_suffix: '_test%env(default::TEST_TOKEN)%'

when@prod:
    doctrine:
        orm:
            auto_generate_proxy_classes: false
// ... lines 29 - 43
```

One other change that's new is this `dbname_suffix` under `when@test`. This is cool. When you're running tests, this will automatically reuse the same database connection configuration, but with a different database name: your normal name followed by `_test`. And this fancy part on the end makes it really easy to run parallel tests with Paratest. This will ensure that each parallel process will use a different database name. You get that all, for free, thanks to the updated recipe.

There's one *other* change in this file, and it's important. In PHPStorm, I can see that the recipe update deleted the `type: annotation` line. Right now, we *are* still using annotations in our project for entity configuration. We're going to *change* that in a few minutes to use PHP 8 attributes, which is going to be *amazing*. But anyways, in the DoctrineBundle configuration, you *no* longer need this `type: annotation` line. If you *don't* have it, the correct format will be detected automatically. If Doctrine sees annotations, it'll *load* annotations! If it sees PHP 8 attributes, it will load *those*. So the best config is *no* config... which tells Doctrine to figure out things *for* us.

Once again, add all these changes, commit, and... let's keep going! Well, let's keep going in the *next* chapter, where we upgrade DoctrineExtensionsBundle, some debug recipes, routing, security and more!

Chapter 7: Recipe Upgrades: Part 2!

Run:

```
composer recipes:update
```

Next up is `doctrine-extensions-bundle`. This one... when we look... just modified a comment! Easy!. So commit that... and then move onto `debug-bundle`.

symfony/debug-bundle Recipe

```
composer recipes:update
```

I'll clear the screen and run that. This made two changes. Run:

```
git status
```

The first change was that it deleted an environment-specific file... and moved it into the main file. The second change, which isn't very common in recipe updates, is that in `config/bundles.php`, it previously loaded `DebugBundle` in the `dev` environment *and* `test` environment. We now recommend *only* loading it in the `dev` environment. You *can* load it in `test` environment, but it tends to slow things down, so it's been removed by default.

config/bundles.php

```
↕ // ... lines 1 - 2
3 return [
↕ // ... lines 4 - 9
10     Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true],
↕ // ... lines 11 - 24
25 ];
```

Easy! Commit those changes... and keep going!

symfony/monolog-bundle Recipe

```
composer recipes:update
```

Next up is `symfony/monolog-bundle`. This one *does* have a conflict, but it's fairly simple. Previously, we had environment-specific files in the `dev/`, `prod/`, and `test/` directories. These have *all* been moved into the central `config/packages/monolog.yaml` file. The only reason it conflicted on *my* project is because I had previously created this file in a tutorial to add a new `markdown` channel. I'll move my `markdown` channel down here... and keep the new stuff.

config/packages/monolog.yaml

```
monolog:
    channels:
        - markdown
// ... lines 4 - 63
```

Below this, you can see the `dev` configuration for logging, the `test` config, and `prod` config. Again, if you had custom config in your old files, make sure you bring that over to the *new* file so it doesn't get lost.

config/packages/monolog.yaml

```
// ... lines 1 - 5
when@dev:
    monolog:
        handlers:
            main:
                type: stream
// ... lines 11 - 26
when@test:
    monolog:
        handlers:
            main:
                type: fingers_crossed
// ... lines 32 - 40
when@prod:
    monolog:
        handlers:
            main:
                type: fingers_crossed
// ... lines 46 - 63
```

Add *these* changes... and... commit.

symfony/routing Recipe

Then right back to:

```
composer recipes:update
```

We're getting closer! Update `symfony/routing`. Let's see. This deleted another environment-specific config file. Yay! Less files! It also highlights a new `default_uri` config that you set if you ever need to generate absolute URLs from inside a command.

Previously, you accomplished this by setting `router.request_context` parameters. It's easier now, and this advertises that.

config/packages/routing.yaml

```yaml
framework:
    router:
        utf8: true

        # Configure how to generate URLs in non-HTTP contexts, such as CLI commands.
        # See https://symfony.com/doc/current/routing.html#generating-urls-in-commands
        #default_uri: http://localhost

when@prod:
    framework:
        router:
            strict_requirements: null
```

Commit this stuff... and let's keep going!

symfony/security-bundle Recipe

```
composer recipes:update
```

We've made it to `symfony/security-bundle`. This one has a conflict... and it's inside `config/packages/security.yaml`. There are a few important things happening. The recipe update added `enable_authenticator_manager: true`. This enables the new security system. We're going to talk about that later. For now, set this to `false` so that we're still using the *old* security system.

config/packages/security.yaml

```yaml
security:
// ... lines 2 - 9
    enable_authenticator_manager: false
// ... lines 11 - 64
```

It also added something called `password_hashers`, which replaces `encoders`. We're also going to talk about *that* later. For right now, I want you to keep both things.

config/packages/security.yaml

```yaml
security:
    encoders:
        App\Entity\User:
            algorithm: auto
// ... lines 5 - 11
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
// ... lines 14 - 64
```

There's also a conflict down on the firewall. The important change is that the new recipe has `lazy: true`. That replaces `anonymous: lazy`, so we can go ahead and keep that change... but use the rest of *our* firewall.

config/packages/security.yaml

```
security:
// ... lines 2 - 20
    firewalls:
// ... lines 22 - 24
        main:
            lazy: true
            provider: app_user_provider
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator
            logout:
                path: app_logout
// ... lines 33 - 64
```

Oh, and at the bottom, we get one shiny new `when@test` section, which sets a custom password hasher. You can read the comment. This accelerates your tests by making it *much* faster to hash passwords in the test environment, where we don't care how secure our hashing algorithm is.

config/packages/security.yaml

```
// ... lines 1 - 51
when@test:
    security:
        password_hashers:
            # By default, password hashers are resource intensive and take time. This is
            # important to generate secure password hashes. In tests however, secure hashes
            # are not important, waste resources and increase test times. The following
            # reduces the work factor to the lowest possible values.
            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon
```

Let's add the files... then keep going.

symfony/translation Recipe

Next up is `symfony/translation`. This isn't important... it just shows off some new config options. Those are all commented out, so... they're cool to see, but not important.

config/packages/translation.yaml

```
framework:
// ... line 2
    translator:
// ... lines 4 - 6
#        providers:
#            crowdin:
#                dsn: '%env(CROWDIN_DSN)%'
#            loco:
#                dsn: '%env(LOCO_DSN)%'
#            lokalise:
#                dsn: '%env(LOKALISE_DSN)%'
```

Commit and... keep going!

symfony/validator Recipe

Next is `symfony/validator`. Simple! This moved the config from `config/test/validator.yaml` into the main `validator.yaml`.

Commit *that*!

symfony/web-profiler-bundle Recipe

Let's update *one more* recipe right now: `web-profiler-bundle`. Can you guess what it did? It added *more* environment-specific config. So the config from `dev/web_profiler.yaml` and `test/web_profiler.yaml` was moved into the main `web_profiler.yaml`. The same thing happened for routes. The config from `dev` was moved into a new `config/routes/web_profiler.yaml`. Let's commit that and... phew! We've *almost* done it! Just *two* recipes left!

Let's update those next. The WebpackEncoreBundle recipe will also give us a chance to upgrade our JavaScript to the new Stimulus 3 version.

Chapter 8: Upgrading Encore and your assets/ Setup

Just two recipes left to update! Let's do `webpack-encore-bundle` next. This recipe changed quite a bit over the past year and a half, so depending on how old your version is, this might be easy.... *or* maybe not so easy. Hmm, let's say that it might be "interesting".

To see what we're working with, run:

```
git status
```

Ok: we have a *number* of modified and deleted files *and* some conflicts. Let's go through those first, starting with `assets/app.js`. As you can see, I enabled some custom `Collapse` functionality from bootstrap. I'm not sure why this conflicted, but it's an easy fix.

assets/app.js

```
↕ // ... lines 1 - 13
14 // activates collapse functionality
15 import { Collapse } from 'bootstrap';
```

Next is `bootstrap.js`. This might actually be a *new* file for you, depending on how old your recipe was. The job of this file is to initialize the Stimulus JavaScript library and load all of the files in the `controllers/` directory as Stimulus controllers. In this case, I already *had* this file, but apparently the expression for how it finds the files changed slightly. The new version is probably better, so let's use that.

assets/bootstrap.js

```
↕ // ... lines 1 - 3
4 export const app = startStimulusApp(require.context(
↕ // ... lines 5 - 6
7     /\.(j|t)sx?$/
8 ));
↕ // ... lines 9 - 12
```

Next up is `controllers.json`. I'm not sure why this is conflicting either... I have a feeling that I may have added these files manually in the past... and now the recipe upgrade is *re-adding* them. I want to keep my custom version.

assets/controllers.json

```
1 {
2     "controllers": {
3         "@symfony/ux-chartjs": {
↕ // ... lines 4 - 7
8         }
9     },
10     "entrypoints": []
11 }
```

Next up is `styles/app.css`. The same thing happened here. The recipe added this file... all the way at the bottom... with just a body background-color. I must have added this file manually... so conflict! Keep all of our custom stuff and...

good!

assets/styles/app.css

```
@import "~bootstrap";

body {
    font-family: spartan;
    color: #444;
}
// ... lines 7 - 138
```

Hello @hotwired/Stimulus v3

The *last* conflict is down here in `package.json`. This one is a bit more interesting. My project was *already* using Stimulus: I have `stimulus` down here and also Symfony's `stimulus-bridge`. The updated recipe now has `@hotwired/stimulus`, and instead of `"@symfony/stimulus-bridge": "^2.0.0"`, it has `"@symfony/stimulus-bridge": "^3.0.0"`.

So what's going on? First, Stimulus version *3* was released. Yay! But... the only real difference between version 2 and 3 is that they renamed the library from `stimulus` to `@hotwired/stimulus`. And in order to get version 3 to work, we *also* need version 3 of `stimulus-bridge`... instead of 2.

So let's take this as a golden opportunity to upgrade from Stimulus 2 to Stimulus 3. As a bonus, after upgrading, you'll get cool new debugging messages in your browser's console when working with Stimulus locally.

Anyways, keep `@hotwired/stimulus`... but move it up so it's in alphabetical order. Use version 3 of `stimulus-bridge`... and even though it doesn't *really* matter since this version constraint allows *any* version `1`, I'll also use the new `webpack-encore` version... and then fix the conflict. Oh, and be sure to delete `stimulus`. We don't want version 2 of `stimulus` hanging around and confusing things.

package.json

```
{
    "devDependencies": {
        "@hotwired/stimulus": "^3.0.0",
// ... line 4
        "@symfony/stimulus-bridge": "^3.0.0",
// ... line 6
        "@symfony/webpack-encore": "^1.7.0",
// ... lines 8 - 13
    },
// ... lines 15 - 22
}
```

Fantastic! Because we just changed some files in `package.json`, find your terminal tab that's rocking Encore, hit "ctrl" + "C", and then run:

```
yarn install
```

or

```
npm install
```

Perfect! Now restart Encore:

```
yarn watch
```

And... it fails!? That's a long error message... but it eventually says:

> "`assets/controllers/answer-vote_controller.js contains a reference to the file "stimulus"`."

The most important, but *boring* part of upgrading from Stimulus 2 to 3 is that you need to go into all of your controllers and change `import { Controller } from 'stimulus'` to `import { Controller } from '@hotwired/stimulus'`.

assets/controllers/answer-vote_controller.js

```
1 import { Controller } from '@hotwired/stimulus';
↕ // ... lines 2 - 22
```

But it's *that* simple. I'm also going to delete `hello_controller.js`... this is just an example controller that the recipe gave us. In the last controller, change to `@hotwired/stimulus`.

assets/controllers/user-api_controller.js

```
1 import { Controller } from '@hotwired/stimulus';
↕ // ... lines 2 - 15
```

Awesome! Stop `yarn watch` again.. and re-run it:

```
yarn watch
```

Dang! We still get an error! This is coming from `@symfony/ux-chartjs/dist/controller.js`.

Upgrading UX Libraries

In my project, I've installed one of the Symfony UX packages, which are PHP packages that *also* give you some JavaScript. Apparently, the JavaScript for that package is still referencing `stimulus` instead of the new `@hotwired/stimulus`. What this tells me is that I probably need to upgrade that PHP package. So, in `composer.json`, down here on `symfony/ux-chartjs`, if you do some research, you'll find out that there's a new version 2 out that supports Stimulus 3.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 37
        "symfony/ux-chartjs": "^2.0",
// ... lines 39 - 45
    },
// ... lines 47 - 109
}
```

After changing that, find your main terminal tab and run:

```
composer up symfony/ux-chartjs
```

to upgrade that *one* package. And... nice! We've upgraded to version 2.1.0. Now it wants us to run:

```
npm install --force
```

or

```
yarn install --force
```

That re-initializes the JavaScript from the package. One thing I want to highlight for this *particular* package is that when we upgraded to version 2 in our `composer.json` file, Flex then updated our `chart.js` dependency from version 2.9 to 3.4. That's because the JavaScript in this new version is meant to work with `chart.js` 3 instead of `chart.js` 2. Flex made that change *for* us. We don't need to do anything here, but it's good to be aware of that.

At last! We should be ready to go. Run

```
yarn watch
```

and... got it! Successful build! Over in the main terminal tab, let's add everything... since we fixed all of the conflicts... and commit!

Upgrading Foundry's Recipe

Now, dear friends, we are on the *last* update. It's `zenstruck/foundry`. This is an easy one. Run:

```
git status
```

It is, once again, environment configuration going into a main file. So let's commit that.

config/packages/zenstruck_foundry.yaml

```yaml
when@dev: &dev
    # See full configuration: https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#full-default-bundle-configuration
    zenstruck_foundry:
        # Whether to auto-refresh proxies by default (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#auto-refresh)
        auto_refresh_proxies: true

when@test: *dev
```

And... we're *done*! All of our recipes are updated! And remember, part of the reason we did all of this is because some of those recipes replaced old deprecated code with new *shiny* code. Hopefully, when we refresh the page, our site will not only still *work*, but will have less deprecations. On my project, if I refresh a few times, it looks like I'm settling in at about 22. Progress!

We *do* need to squash those deprecations. But next, one *other* thing we need to do is... upgrade our code to PHP 8! This is another spot where Rector can help!

Chapter 9: Upgrading to PHP 8

Let's keep track of our goal. Now that we've upgraded to Symfony 5.4, as soon as we remove all of these deprecations, we can *safely* upgrade to Symfony 6. *But* Symfony 6 requires PHP 8, and I've been building this project in PHP 7. So the *next* step is to update our code to be PHP 8 compatible. In practice, that means updating parts of our code to use some cool new PHP 8 features. Woo! And this is another spot where Rector can help us.

Rector Upgrading To PHP 8!

Start by opening up `rector.php` and removing the three Symfony upgrade lines. Replace these with `LevelSetList::UP_TO_PHP_80`. Just like with Symfony, you can upgrade *specifically* to PHP 7.3 or 7.4, but they have these nice `UP_TO_[...]` statements that will upgrade our code across all versions of PHP *up to* PHP 8.0.

rector.php

```
↕ // ... lines 1 - 12
13 return static function (ContainerConfigurator $containerConfigurator): void {
↕ // ... lines 14 - 22
23     $containerConfigurator->import(LevelSetList::UP_TO_PHP_80);
↕ // ... lines 24 - 29
30 };
```

And... that's all we need!

Over at your terminal, I've committed all of my changes, except for the one we just made. So now we can run:

```
vendor/bin/rector process src
```

Cool! Let's walk through some of these changes. If you want to go deeper, search for a getrector.org blog post, which shows you how to do what we just did... but *also* gives you more information about what Rector did and *why*.

For example, one of the changes that it makes is replacing `switch()` statements with a new PHP 8 `match()` function. This explains *that*... and many other changes. Oh, and the vast majority of these changes aren't *required*: you don't *have* to do them to upgrade to PHP 8. They're just nice.

PHP 8 Property Promotion

The most important change, which is *coincidentally* the most common, is something called "Promoted Properties". This is one of my favorite features in PHP 8, and you can see it right here. In PHP 8, you can add a `private`, `public`, or `protected` keyword right before an argument in the constructor... and that will both *create* that property and *set* it to this value. So you no longer need to add a property manually or set it below. Just add `private` and... done!

src/Twig/MarkdownExtension.php

```
↕ // ... lines 1 - 8
9  class MarkdownExtension extends AbstractExtension
10 {
11     public function __construct(private MarkdownHelper $markdownHelper)
12     {
13     }
↕ // ... lines 14 - 28
29 }
```

The vast majority of the changes in this file are exactly that... here's another example in `MarkdownHelper`. Most of the *other* changes are minor. It altered some callback functions to use the new short `=>` syntax, which is actually from PHP 7.4.

src/Service/MarkdownHelper.php

```
↕ // ... lines 1 - 8
9  class MarkdownHelper
10 {
↕ // ... lines 11 - 14
15     public function parse(string $source): string
16     {
↕ // ... lines 17 - 24
25         return $this->cache->get('markdown_'.md5($source), fn() => $this->markdownParser->transformMarkdown($source));
26     }
27 }
```

You can also see, down here, an example of refactoring `switch()` statements to use the new `match()` function.

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 11
12 class QuestionVoter extends Voter
13 {
↕ // ... lines 14 - 24
25     protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
26     {
↕ // ... lines 27 - 40
41         return match ($attribute) {
42             'EDIT' => $user === $subject->getOwner(),
43             default => false,
44         };
45     }
46 }
```

All of this is optional, but it's nice that our code has been updated to use some of the new features. If I scroll down just a little more, you'll see more of these.

Entity Property Types?

Oh, and inside of our entities, notice that, in some cases, it added property types! For `$roles`, this property is initialized to an array. Rector *realized* that... so it added the `array` type.

src/Entity/User.php

```
↕ // ... lines 1 - 15
16 class User implements UserInterface
17 {
↕ // ... lines 18 - 29
30     /**
31      * @ORM\Column(type="json")
32      */
33     private array $roles = [];
↕ // ... lines 34 - 226
227 }
```

In other cases, like `$password`, it saw that we have PHPDoc above it, so it added the type there as well.

src/Entity/User.php

```
↕ // ... lines 1 - 15
16 class User implements UserInterface
17 {
↕ // ... lines 18 - 34
35     /**
36      * @var string The hashed password
37      * @ORM\Column(type="string")
38      */
39     private string $password;
↕ // ... lines 40 - 226
227 }
```

Though, this is a little questionable. The `$password` *could* also be null.

Open up `src/Entity/User.php` and scroll down to `$password`. Rector gave this a `string` type... but that's wrong! If you look at the constructor down here, we don't initialize `$password` to any value... which means it will *start* `null`. So the *correct* type for this is a *nullable* `?string`. The reason Rector did this wrong is... well.. because *I* had a bug in my documentation!. This should be `string|null`

src/Entity/User.php

```
↕ // ... lines 1 - 15
16 class User implements UserInterface
17 {
↕ // ... lines 18 - 34
35     /**
36      * @var string|null The hashed password
37      * @ORM\Column(type="string")
38      */
39     private ?string $password = null;
↕ // ... lines 40 - 226
227 }
```

One of the biggest changes that I've been doing in my code over the past year or so since PHP 7.3 was released, has been adding property types like this, both in my entity classes and also my service classes. If this was a little confusing, don't worry. We're going to talk more about property types inside of *entities* in a few minutes. You can see that Rector added *some*, but a lot of our properties are still missing them.

Setting PHP 8 in composer.json

Okay, our code *should* now be ready for PHP 8. Yay! So let's go upgrade our *dependencies* for PHP 8. In `composer.json`, under the `require` key, it currently says that my project works with PHP 7.4 or 8. I'm going to change that to just say `"php": "^8.0.2"`, which is the minimum version for Symfony 6.0.

composer.json

```
{
// ... lines 2 - 5
    "require": {
        "php": "^8.0.2",
// ... lines 8 - 45
    },
// ... lines 47 - 109
}
```

By the way, Symfony 6.1 requires PHP 8.1. So if you're going to upgrade to that pretty soon, you could jump straight to 8.1.

There's one other thing I have down here near the bottom. Let's see... here we go. On `config`, `platform`, I have PHP set to 7.4. That ensures that if someone is using PHP 8, Composer will *still* make sure it downloads dependencies compatible with PHP 7.4. Change this to `8.0.2`.

composer.json

```
{
// ... lines 2 - 56
    "config": {
// ... lines 58 - 61
        "platform": {
            "php": "8.0.2"
        },
// ... lines 65 - 68
    },
// ... lines 70 - 109
}
```

Sweet! And now, because we're using PHP 8 in our project, there's a good chance some dependencies will be eligible for updates. Run:

```
composer up
```

And... yeah! There are several. It looks like `psr/cache`, `psr/log`, and `symfony/event-dispatcher-contracts` all upgraded. Most likely all of these new versions require PHP 8. We couldn't upgrade before, but now we *can*. If we go over to our page and reload... everything still works!

Updating Symfony Flex

One other thing in `composer.json` is Symfony Flex itself. Flex uses its own version scheme, and the latest version is 2.1. At this moment, Flex version 2 and Flex version 1 are identical... except that Flex 2 requires PHP 8. Since we're using that, let's upgrade! Change the version to `^2.1`... then head back to your terminal and run:

```
composer up
```

one more time. Beautiful!

All right, team! Our project is now using PHP 8. To celebrate, let's refactor from using annotations to PHP 8 native attributes. OOOoo. I love this change... in part because Rector makes it *super* easy.

Chapter 10: Annotations to Attributes

Now that we're on PHP 8, let's convert our PHP annotations to the more hip and happening PHP 8 attributes. Refactoring annotations to attributes is basically just... busy work. You *can* do it by hand: attributes and annotations work exactly the same and use the same classes. Even the syntax is only a little different: you use colons to separate arguments... because you're actually leveraging PHP *named* arguments. Neato.

Configuring Rector to Upgrade Annotations

So, converting is simple... but oof, I am *not* excited to do *all* of that manually. Fortunately, Rector comes back to the rescue!! Search for "rector annotations to attributes" to find a blog post that tells you the exact import configuration we need in `rector.php`. Copy these three things. Oh, and starting in Rector 0.12, there's a new, simpler `RectorConfig` object that you'll see on this page. If you have that version, feel free to use *that* code.

Oh, and before we paste this in, find your terminal, add everything... and then commit. Perfect!

Back over in `rector.php`, replace the *one* line with these *four* lines... except we don't need the `NetteSetList`... and we need to add a few `use` statements. I'll retype the "t" in `DoctrineSetList`, hit "tab", and do the same for `SensiolabsSetList`.

rector.php

```
// ... lines 1 - 6
use Rector\Doctrine\Set\DoctrineSetList;
// ... lines 8 - 9
use Rector\Symfony\Set\SensiolabsSetList;
// ... lines 11 - 14
return static function (ContainerConfigurator $containerConfigurator): void {
// ... lines 16 - 24
    $containerConfigurator->import(DoctrineSetList::ANNOTATIONS_TO_ATTRIBUTES);
    $containerConfigurator->import(SymfonySetList::ANNOTATIONS_TO_ATTRIBUTES);
    $containerConfigurator->import(SensiolabsSetList::FRAMEWORK_EXTRA_61);
// ... lines 28 - 33
};
```

Now, you know the drill. Run

```
vendor/bin/rector process src
```

and see what happens. Whoa... this is awesome! Look! It beautifully refactored this annotation to an attribute and... it did this *all over* the place! We have routes up here. And all of our entity annotations, like the `Answer` entity have *also* been converted. That was a *ton* of work... all automatic!

src/Controller/UserController.php

```
// ... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
use Symfony\Component\Routing\Annotation\Route;

class UserController extends BaseController
{
    #[Route(path: '/api/me', name: 'app_user_api_me')]
    #[IsGranted('IS_AUTHENTICATED_REMEMBERED')]
    public function apiMe(): \Symfony\Component\HttpFoundation\Response
    {
// ... lines 14 - 16
    }
}
```

src/Entity/User.php

```
// ... lines 1 - 11
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\Table(name: '`user`')]
class User implements UserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;
// ... lines 20 - 203
}
```

Fixing PHP CS

Though it did, as Rector sometimes does, mess up some of our coding standards. For example, all the way at the bottom, it *did* refactor this `Route` annotation to an attribute... but then it added a little extra space before the `Response` return type. That's no problem. After you run Rector, it's always a good idea to run PHP CS Fixer. Do it:

```
tools/php-cs-fixer/vendor/bin/php-cs-fixer fix
```

Love it. A bunch of fixes to bring our code back in line. Run

```
git diff
```

to see how things look now. The `Route` annotation changed into an attribute... and PHP CS Fixer put the `Response` return type back the way it was before. Rector even refactored `IsGranted` from SensioFrameworkExtraBundle into an attribute.

But if you keep scrolling down until you find an entity... here we go... uh oh! It killed the line breaks between our properties! It's not super obvious on the diff, but if you open any entity... yikes! This looks... *cramped*. I *like* the line breaks between my entity properties.

src/Entity/Answer.php

```
↕ // ... lines 1 - 9
10 class Answer
11 {
12     use TimestampableEntity;
13     public const STATUS_NEEDS_APPROVAL = 'needs_approval';
14     public const STATUS_SPAM = 'spam';
15     public const STATUS_APPROVED = 'approved';
16     #[ORM\Id]
17     #[ORM\GeneratedValue]
18     #[ORM\Column(type: 'integer')]
19     private $id;
20     #[ORM\Column(type: 'text')]
21     private $content;
↕ // ... lines 22 - 48
49     public function getUsername(): ?string
↕ // ... lines 50 - 113
114 }
```

We *could* fix this by hand... but I'm wondering if we can teach PHP CS Fixer to do this for us.

Open `php-cs-fixer.php`. The rule that controls these line breaks is called `class_attributes_separation` with an "s" - I'll fix that in a minute. Set this to an array that describes all of the different parts of our class and how each should behave. For example, we can say `['method' => 'one']` to say that we want *one* empty line between each method. We can also say `['property' => 'one']` to have one line break between our properties. There's also another called `trait_import`. Set that to `one` too. That gives us an empty line between our trait imports, which is something that we have on top of `Answer`.

.php-cs-fixer.php

```
↕ // ... lines 1 - 7
8 return $config->setRules([
↕ // ... lines 9 - 10
11         'class_attributes_separation' => [
12             'elements' => ['method' => 'one', 'property' => 'one', 'trait_import' => 'one']
13         ]
14     ])
↕ // ... line 15
16 ;
```

Now try php-cs-fixer again:

```
tools/php-cs-fixer/vendor/bin/php-cs-fixer fix
```

Whoops!

> *"The rules contain unknown fixers: "class_attribute_separation""*

I meant to say `class_attributes_separation` with an "s". What a great error though. Let's try that again and... cool! It changed five files, and if you check those... they're back!

src/Entity/Answer.php

```
// ... lines 1 - 9
class Answer
{
    use TimestampableEntity;

    public const STATUS_NEEDS_APPROVAL = 'needs_approval';
    public const STATUS_SPAM = 'spam';
    public const STATUS_APPROVED = 'approved';

    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;

    #[ORM\Column(type: 'text')]
    private $content;
// ... lines 25 - 120
}
```

With just a few commands we've converted our entire site from annotations to attributes. Woo!

Next, let's add property types to our entities. That's going to allow us to have *less* entity config thanks to a new feature in Doctrine.

Chapter 11: Adding Property Types to Entities

A new feature snuck into Doctrine a while back, and it's *super* cool. Doctrine can now *guess* some configuration about a property via its *type*. We'll start with the relationship properties. But first, I want to make sure that my database is in sync with my entities. Run:

```
symfony console doctrine:schema:update --dump-sql
```

And... yep! My database *does* look like my entities. We'll run this command again later after we make a bunch of changes... because our goal isn't actually to *change* any of our database config: just to simplify it. Oh, and yes, this dumped out a bunch of deprecations... we *will* fix those... eventually... I promise!

Removing targetEntity

So here's change number one. This `question` property holds a `Question` object. So let's add a `Question` *type*. But we have to be careful. It needs to be a *nullable* `Question`. Even though this is required in the database, after we instantiate the object, the property won't *instantly* be populated: it will, at least temporarily, *not* be set. You'll see me do this with all of my entity property types. If it's *possible* for a property to be `null` - even for a moment - we need to reflect that.

src/Entity/Answer.php

```
// ... lines 1 - 9
class Answer
{
// ... lines 12 - 33
    private ?Question $question = null;
// ... lines 35 - 120
}
```

I'm also going to initialize this with `= null`. If you're new to property types, here's the deal. If you add a type to a property... then try to access it *before* that property has been set to some value, you'll get an error, like

> *"Typed property Answer::$question must not be accessed before initialization."*

Without a property type, the `= null` isn't needed, but now it is. Thanks to this, if we instantiate an `Answer` and then call `getQuestion()` before that property is set, things won't explode.

Ok, so adding property types is nice: it makes our code cleaner and tighter. *But*, there's another big advantage: we don't need the `targetEntity` anymore! Doctrine is now able to figure that out for us. So delete this... and celebrate!

src/Entity/Answer.php

```
// ... lines 1 - 9
class Answer
{
// ... lines 12 - 31
    #[ORM\ManyToOne(inversedBy: 'answers')]
// ... line 33
    private ?Question $question = null;
// ... lines 35 - 120
}
```

Then... keep going to `Question`. I'm looking specifically for relationship fields. This one is a `OneToMany`, which holds a collection of `$answers`. We *are* going to add a type here... but in a minute. Let's focus on the `ManyToOne` relationships first.

Down here, for `owner`, add `?User`, `$owner = null`, then get rid of `targetEntity`.

src/Entity/Question.php

```
// ... lines 1 - 13
class Question
{
// ... lines 16 - 47
    #[ORM\ManyToOne(inversedBy: 'questions')]
// ... line 49
    private ?User $owner = null;
// ... lines 51 - 219
}
```

And then in `QuestionTag`, do the same thing: `?Question $question = null`... and do your victory lap by removing `targetEntity`.

src/Entity/QuestionTag.php

```
// ... lines 1 - 8
class QuestionTag
{
// ... lines 11 - 15
    #[ORM\ManyToOne(inversedBy: 'questionTags')]
// ... line 17
    private ?Question $question = null;
// ... lines 19 - 71
}
```

And... down here... one more time! `?Tag $tag = null`... and say bye bye to `targetEntity`.

src/Entity/QuestionTag.php

```
// ... lines 1 - 8
class QuestionTag
{
// ... lines 11 - 19
    #[ORM\ManyToOne()]
// ... line 21
    private ?Tag $tag = null;
// ... lines 23 - 71
}
```

Sweet! To make sure we didn't mess anything up, re-run the `schema:update` command from earlier:

```
symfony console doctrine:schema:update --dump-sql
```

And... we're still good!

Adding Types to All Properties

Ok, let's go further and add types to *every* property. This will be more work, but the result is worth it. For `$id`, this will be a nullable `int`... and initialize it to `null`. Thanks to that, we don't need `type: 'integer'`: Doctrine can now figure that out.

src/Entity/Answer.php

```
↕ // ... lines 1 - 9
10 class Answer
11 {
↕ // ... lines 12 - 19
20     #[ORM\Column()]
21     private ?int $id = null;
↕ // ... lines 22 - 120
121 }
```

For `$content`, a nullable string... with `= null`. But in this case, we *do* need to keep `type: 'text'`. When Doctrine sees the `string` type, it *guesses* `type: 'string'`... which holds a maximum of 255 characters. Since this field holds a *lot* of text, override the guess with `type: 'text'`.

src/Entity/Answer.php

```
↕ // ... lines 1 - 9
10 class Answer
11 {
↕ // ... lines 12 - 22
23     #[ORM\Column(type: 'text')]
24     private ?string $content = null;
↕ // ... lines 25 - 120
121 }
```

Initialize string Field to null or ''?

By the way, some of you might be wondering why I don't use `$content = ''` instead. Heck, then we could remove the nullable `?` on the type! That's a good question! The reason is that this field is required in the database. If we initialize the property to empty quotes... and I had a bug in my code where I *forgot* to set the `$content` property, it would *successfully* save to the database with content set to an empty string. By initializing it to `null`, if we forget to set this field, it will *explode* before it enters the database. Then, we can fix that bug... instead of it just silently saving the empty string. It may be sneaky, but we're *sneakier*.

Okay, let's keep going! A lot of this will be busy work... so let's move as quickly as we can. Add the type to `username`... and remove the Doctrine `type` option. We can also delete `length`... since the default has always been `255`. The `$votes` property looks good, but we can get rid of `type: 'integer'`. And down here for `$status`, this already has the type, so delete `type: 'string'`. But we *do* need to keep the `length` if we want it to be shorter than 255.

src/Entity/Answer.php

```
// ... lines 1 - 9
class Answer
{
// ... lines 12 - 25
    #[ORM\Column()]
    private ?string $username = null;

    #[ORM\Column()]
    private int $votes = 0;
// ... lines 31 - 35
    #[ORM\Column(length: 15)]
    private string $status = self::STATUS_NEEDS_APPROVAL;
// ... lines 38 - 120
}
```

Moving on to the `Question` entity. Give `$id` the type... remove its `type` Doctrine option, update `$name`... delete *all* of its options.... and repeat this for `$slug`. Notice that `$slug` still uses an annotation from `@Gedmo\Slug`. We'll fix that in a minute.

Update `$question`... then `$askedAt`. This is a `type: 'datetime'`, so that's going to hold a `?\DateTime` instance. I'll also initialize it to null. Oh, and I forgot to do it, but we *could* now remove `type: 'datetime'`.

src/Entity/Question.php

```
// ... lines 1 - 13
class Question
{
// ... lines 16 - 19
    #[ORM\Column()]
    private ?int $id = null;

    #[ORM\Column()]
    private ?string $name = null;

    /**
     * @Gedmo\Slug(fields={"name"})
     */
    #[ORM\Column(length: 100, unique: true)]
    private ?string $slug = null;

    #[ORM\Column(type: 'text')]
    private ?string $question = null;

    #[ORM\Column(nullable: true)]
    private ?\DateTime $askedAt = null;
// ... lines 37 - 219
}
```

Typing Collection Properties

And now we're back to the `OneToMany` relationship. If you look down, this is initialized in the constructor to an `ArrayCollection`. So you might think we should use `ArrayCollection` for the type. But instead, say `Collection`.

That's an interface from Doctrine that `ArrayCollection` implements. We need to use `Collection` here because, when we *query* for a `Question` from the database and then fetch the `$answers` property, Doctrine will set that to a

different object: a `PersistentCollection`. So this property might be an `ArrayCollection`, *or* a `PersistentCollection`... but in all cases, it will implement this `Collection` interface. And this does *not* need to be nullable because it's initialized *inside* the constructor. Do the same thing for `$questionTags`.

src/Entity/Question.php

```
↕ // ... lines 1 - 13
14 class Question
15 {
↕ // ... lines 16 - 42
43     private Collection $answers;
↕ // ... lines 44 - 45
46     private Collection $questionTags;
↕ // ... lines 47 - 219
220 }
```

Believe it our not, we're in the home stretch! In `QuestionTag`... make our usual `$id` changes... then head down to `$taggedAt`. This is a `datetime_immutable` type, so use `\DateTimeImmutable`. Notice that I did *not* make this nullable and I'm not *initializing* it to null. That's simply because we're setting this in the constructor. So we're guaranteed that it will *always* hold a `\DateTimeImmutable` instance: it will never be null.

src/Entity/QuestionTag.php

```
↕ // ... lines 1 - 8
9 class QuestionTag
10 {
↕ // ... lines 11 - 12
13     #[ORM\Column()]
14     private ?int $id = null;
↕ // ... lines 15 - 23
24     #[ORM\Column()]
25     private \DateTimeImmutable $taggedAt;
↕ // ... lines 26 - 71
72 }
```

Ok, now to `Tag`. Do our usual `$id` dance. But wait... back in `QuestionTag`, I forgot to remove the `type: 'integer'`. It doesn't hurt anything... it's just not needed. And... same for `type: 'datetime_immutable'`.

Back over in `Tag`, let's keep going with the `$name` property... this is all normal...

src/Entity/Tag.php

```
↕ // ... lines 1 - 9
10 class Tag
11 {
↕ // ... lines 12 - 15
16     #[ORM\Column()]
17     private ?int $id = null;
18
19     #[ORM\Column()]
20     private ?string $name = null;
↕ // ... lines 21 - 37
38 }
```

Then jump to our *last* class: `User`. I'll speed through the boring changes to `$id` and `$email`... and `$password`. Let's also remove the `@var` PHP Doc above this: that's now totally redundant. Do that same thing for `$plainPassword`. Heck, this `@var` wasn't even right - it should have been `string|null`!

Let's zoom through the last changes: `$firstName`, add `Collection` to `$questions`... and no `type` needed for `$isVerified`.

src/Entity/User.php

```
// ... lines 1 - 13
class User implements UserInterface
{
// ... lines 16 - 17
    #[ORM\Column()]
    private ?int $id = null;

    #[ORM\Column(length: 180, unique: true)]
    private ?string $email = null;
// ... lines 23 - 29
    #[ORM\Column(type: 'string')]
    private ?string $password = null;

    /**
     * Non-mapped field
     */
    private ?string $plainPassword = null;

    #[ORM\Column()]
    private ?string $firstName = null;

    #[ORM\OneToMany(targetEntity: Question::class, mappedBy: 'owner')]
    private Collection $questions;

    #[ORM\Column(type: 'boolean')]
    private bool $isVerified = false;
// ... lines 46 - 210
}
```

And... we're done! This *was* a chore. But going forward, using property types will mean tighter code... and less Doctrine config.

But... let's see if we messed anything up. Run `doctrine:schema:update` one last time:

```
symfony console doctrine:schema:update --dump-sql
```

It's clean! We changed a ton of config, but that didn't actually change how any of our entities are mapped. Mission accomplished.

Updating Gedmo\Slug Annotation

Oh, and as promised, there's one last annotation that we need to change: it's in the `Question` entity above the `$slug` field. This comes from the Doctrine extensions library. Rector didn't update it... but it's super easy. As long as you have Doctrine Extensions 3.6 or higher, you can use this as an attribute. So `#[Gedmo\Slug()]` with a `fields` option that we need to set to an array. The cool thing about PHP attributes are... they're just PHP code! So writing an array in attributes... is the same as writing an array in PHP. Inside, pass `'name'`... using single quotes, just like we usually do in PHP.

src/Entity/Question.php

```
↕ // ... lines 1 - 9
10 use Gedmo\Mapping\Annotation as Gedmo;
↕ // ... lines 11 - 13
14 class Question
15 {
↕ // ... lines 16 - 25
26     #[Gedmo\Slug(fields: ['name'])]
27     #[ORM\Column(length: 100, unique: true)]
28     private ?string $slug = null;
↕ // ... lines 29 - 217
218 }
```

Ok team: we just took our codebase a *huge* step forward. Next, let's dial in on these remaining deprecations and work on squashing them. We're going to start with the elephant in the room: converting to the new security system. But don't worry! It's easier than you might think!

Chapter 12: Security Upgrades

It's time to fix these deprecations so that we can *finally* upgrade to Symfony 6. Go to any page on the site and click the deprecations down on the web debug toolbar to see the list. This is a *big* list... but a lot of these relate to the same thing: security.

The biggest - and perhaps most wonderful - change in Symfony 5.4 and Symfony 6, is the new security system. But don't worry. It's not *that* much different from the old one... and the upgrade path is surprisingly easy.

UserInterface, getPassword & PasswordAuthenticatedUserInterface

For the first change, open up the `User` entity. In addition to `UserInterface`, add a second `PasswordAuthenticatedUserInterface`. Until recently, `UserInterface` had a *lot* of methods on it, including `getPassword()`.

src/Entity/User.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
↕ // ... lines 10 - 14
15 class User implements UserInterface, PasswordAuthenticatedUserInterface
16 {
↕ // ... lines 17 - 211
212 }
```

But... this didn't always make sense. For example, some security systems have users that *don't* have passwords. For example, if your users log in via a single sign-on system, then there *are* no passwords to handle. Well, the user might enter their password into *that* system... but as far as our app is concerned, there are no passwords.

To make this cleaner, in Symfony 6, `getPassword()` was *removed* from `UserInterface`. So you *still* always need to implement `UserInterface`... but then the `getPassword()` method and its `PasswordAuthenticatedUserInterface` are optional.

UserInterface: getUsername() -> getUserIdentifier()

Another change relates to `getUsername()`. This method lives on `UserInterface`... but its name was *always* confusing. It made it *seem* like you needed to have a *username*... when really, this method is *supposed* to return any unique user identifier - not *necessarily* a username. Because of that, in Symfony 6, this has been renamed from `getUsername()` to `getUserIdentifier()`. Copy this, paste, change `getUsername` to `getUserIdentifier()`... and that's it.

src/Entity/User.php

```
// ... lines 1 - 14
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 17 - 69
    /**
     * A visual identifier that represents this user.
     *
     * @see UserInterface
     */
    public function getUserIdentifier(): string
    {
        return (string) $this->email;
    }
// ... lines 79 - 221
}
```

We *do* need to keep `getUsername()` for now because we're still on Symfony 5.4... but once we upgrade to Symfony 6, we can safely remove it.

New Security System: enable_authenticator_manager

But the *biggest* change in Symfony's security system can be found in `config/packages/security.yaml`. It's this `enable_authenticator_manager`. When we upgraded the recipe, it gave us this config... but it was set to `true`.

config/packages/security.yaml

```
security:
// ... lines 2 - 9
    enable_authenticator_manager: false
// ... lines 11 - 64
```

This teenie, tiny, innocent-looking line allows us to switch from the old security system to the new one. And what *that* means, in practice, is that all of the ways you authenticate - like a custom authenticator or `form_login` or `http_basic` - will suddenly start using an entirely new system under the hood.

For the most part, if you're using one of the *built-in* authentication systems, like `form_login` or `http_basic`... you probably won't notice any changes. You can activate the new system by setting this to true... and everything will work exactly like before.... even though the code behind `form_login` will suddenly be very different. In a lot of ways, the new security system is an internal refactoring to make the core code more readable and to give us more flexibility, when we need it.

Guard -> Custom Authenticator Conversion

However, if you have any custom `guard` authenticators... like we do, you'll need to convert these to the new authenticator system... which is super fun anyways... so let's do it!

Open up our custom authenticator: `src/Security/LoginFormAuthenticator.php`. We can already see that `AbstractFormLoginAuthenticator` from the old system is deprecated. Change this to `AbstractLoginFormAuthenticator`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 20
21 use Symfony\Component\Security\Http\Authenticator\AbstractLoginFormAuthenticator;
↕ // ... lines 22 - 23
24 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
25 {
↕ // ... lines 26 - 107
108 }
```

I know, it's almost the exact same name: we just swapped "Form" and "Login" around. If your custom authenticator is *not* for a login form, then change your class to `AbstractAuthenticator`.

Oh, and we don't need to implement `PasswordAuthenticatedInterface` anymore: that was something for the *old* system.

Adding the New Authenticator Methods

The old Guard system and new authenticator system *do* the same thing: they figure out who's trying to log in, check the password, and decide what to do on success and failure. But the new authenticator style will feel quite a bit different. For example, you can *immediately* see that PhpStorm is furious because we now need to implement a new method called `authenticate()`.

Ok! I'll go down below `supports()`, go to "Code Generate" - or "cmd" + "N" on a Mac - and implement that new `authenticate()` method. This is the *core* of the new authenticator system... and we're going to talk about it in a few minutes.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 24
25 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
26 {
↕ // ... lines 27 - 40
41     public function authenticate(Request $request)
42     {
43         // TODO: Implement authenticate() method.
44     }
↕ // ... lines 45 - 113
114 }
```

Oh, but the old and new systems do *share* several methods. Like, they both have a method called `supports()`... but the new system has a `bool` return type. As soon as we add that, PhpStorm is happy.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 35
36     public function supports(Request $request): bool
37     {
↕ // ... lines 38 - 39
40     }
↕ // ... lines 41 - 114
115 }
```

Below, on `onAuthenticationSuccess()`, it looks like we need to add a return type here as well. At the end, add the `Response` type from HttpFoundation. Nice! And while we're working on this method, rename the `$providerKey` argument to `$firewallName`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 90
91     public function onAuthenticationSuccess(Request $request, TokenInterface $token, string
   $firewallName): Response
92     {
↕ // ... lines 93 - 97
98     }
↕ // ... lines 99 - 114
115 }
```

You don't *have* to do this, that's just the new name of the argument... and it's more clear.

Next, down on `onAuthenticationFailure()`, add the `Response` return type there as well. Oh, and for `onAuthenticationSuccess()`, I just remembered that this can return a *nullable* `Response`. In some systems - like API token authentication - you will *not* return a response.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 99
100     public function onAuthenticationFailure(Request $request, AuthenticationException
   $exception): Response
101     {
↕ // ... lines 102 - 108
109     }
↕ // ... lines 110 - 114
115 }
```

Finally, we *still* need a `getLoginUrl()` method, but in the new system, this accepts a `Request $request` argument and returns a `string`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 110
111     protected function getLoginUrl(Request $request): string
112     {
113         return $this->urlGenerator->generate(self::LOGIN_ROUTE);
114     }
115 }
```

Alright! we still need to fill in the "guts", but we *at least* have all the methods we need.

Removing supports() for "form login" authenticators

And actually, we can remove one of these! Delete the `supports()` method.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 35
36     public function supports(Request $request): bool
37     {
38         return self::LOGIN_ROUTE === $request->attributes->get('_route')
39             && $request->isMethod('POST');
40     }
↕ // ... lines 41 - 114
115 }
```

Ok, this method *is* still needed by custom authenticators and its job is the same as before. But, if you jump into the base class, in the new system, the `supports()` method is implemented *for* you. It checks to make sure that the current request is a `POST` and that the *current* URL is the same as the *login* URL. Basically, it says

> *"I support authenticating this request if this is a POST request to the login form."*

We wrote our logic a bit differently before, but that's *exactly* what we were checking.

Ok, it's time to get to the *meat* of our custom authenticator: the `authenticate()` method. Let's do that next.

Chapter 13: Custom Authenticator authenticate() Method

We're currently converting our old Guard authenticator to the new authenticator system. And, nicely, these two systems *do* share some methods, like `supports()`, `onAuthenticationSuccess()` and `onAuthenticationFailure()`.

The *big* difference is down inside the new `authenticate()` method. In the old Guard system, we split up authentication into a few methods. We had `getCredentials()`, where we grab some information, `getUser()`, where we found the `User` object, and `checkCredentials()`, where we checked the password. All three of these things are now combined into the `authenticate()` method... with a few nice bonuses. For example, as you'll see in a second, it's no longer *our* responsibility to check the password. That now happens *automatically*.

The Passport Object

Our job in `authenticate()` is simple: to return a `Passport`. Go ahead and add a `Passport` return type. That's actually needed in Symfony 6. It wasn't added *automatically* due to a deprecation layer and the fact that the return type changed from `PassportInterface` to `Passport` in Symfony 5.4.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 26
27 use Symfony\Component\Security\Http\Authenticator\Passport\Passport;
↕ // ... lines 28 - 29
30 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
31 {
↕ // ... lines 32 - 39
40     public function authenticate(Request $request): Passport
41     {
↕ // ... lines 42 - 66
67     }
↕ // ... lines 68 - 136
137 }
```

Anyways, this method returns a `Passport`... so do it: `return new Passport()`. By the way, if you're new to the custom authenticator system and want to learn more, check out our Symfony 5 Security tutorial where we talk *all* about this. I'll go through the basics now, but the details live there.

Before we fill in the `Passport`, grab all the info from the `Request` that we need... paste... then set each of these as variables: `$email =`, `$password =`... and let's worry about the CSRF token later.

src/Security/LoginFormAuthenticator.php

```
// ... lines 1 - 39
    public function authenticate(Request $request): Passport
    {
        $email = $request->request->get('email');
        $password = $request->request->get('password');

        return new Passport(
// ... lines 46 - 65
        );
    }
// ... lines 68 - 138
```

The first argument to the `Passport` is a `new UserBadge()`. What you pass here is the *user identifier*. In our system, we're logging in via the email, so pass `$email`!

And... if you want, you can stop right here. If you only pass one argument to `UserBadge`, Symfony will use the "user provider" from `security.yaml` to find that user. We're using an `entity` provider, which tells Symfony to try to query for the `User` object in the database via the `email` property.

Optional Custom User Query

In the old system, we did this all manually by querying the `UserRepository`. That is *not* needed anymore. But sometimes... if you have custom logic, you might *still* need to find the user manually.

If you have this use-case, pass a `function()` to the second argument that accepts a `$userIdentifier` argument. Now, when the authentication system needs the User object, it will call our function and pass us the "user identifier"... which will be whatever *we* passed to the first argument. So, the email.

Our job is to *use* that to return the user. Start with
`$user = $this->entityManager->getRepository(User::class)`

And yea, I could have injected the `UserRepository` instead of the entity manager... that would be better... but this is fine. Then `->findOneBy(['email' => $userIdentifier])`.

If we did *not* find a user, we need to `throw` a `new UserNotFoundException()`. *Then*, `return $user`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 39
40      public function authenticate(Request $request): Passport
41      {
↕ // ... lines 42 - 44
45          return new Passport(
46              new UserBadge($email, function($userIdentifier) {
47                  // optionally pass a callback to load the User manually
48                  $user = $this->entityManager
49                      ->getRepository(User::class)
50                      ->findOneBy(['email' => $userIdentifier]);
51
52                  if (!$user) {
53                      throw new UserNotFoundException();
54                  }
55
56                  return $user;
57              }),
↕ // ... lines 58 - 65
66          );
67      }
↕ // ... lines 68 - 138
```

First `Passport` argument is done!

PasswordCredentials

For the second argument, down here, change my bad semicolon to a comma - then say `new PasswordCredentials()` and pass this the submitted `$password`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 39
40      public function authenticate(Request $request): Passport
41      {
↕ // ... lines 42 - 44
45          return new Passport(
46              new UserBadge($email, function($userIdentifier) {
↕ // ... lines 47 - 56
57              }),
58              new PasswordCredentials($password),
↕ // ... lines 59 - 65
66          );
67      }
↕ // ... lines 68 - 138
```

That's all we need! That's right: we do *not* need to actually *check* the password! We pass a `PasswordCredentials()`... and then another system is responsible for checking the submitted password against the hashed password in the database! How cool is that?

Extra Badges

Finally, the `Passport` accepts an optional array of "badges", which are extra "stuff" that you want to add... usually to activate other features.

We only need to pass *one*: a `new CsrfTokenBadge()`. This is because our login form is protected by a CSRF token. Previously, we checked that manually. Lame!

But no more! Pass the string `authenticate` to the first argument... which just needs to match the string used when we generate the token in the template: `login.html.twig`. If I search for `csrf_token`... there it is!

For the second argument, pass the submitted CSRF token: `$request->request->get('_csrf_token')`, which you can also see in the login form.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 39
40     public function authenticate(Request $request): Passport
41     {
↕ // ... lines 42 - 44
45         return new Passport(
↕ // ... lines 46 - 57
58             new PasswordCredentials($password),
59             [
60                 new CsrfTokenBadge(
61                     'authenticate',
62                     $request->request->get('_csrf_token')
63                 ),
↕ // ... line 64
65             ]
66         );
67     }
↕ // ... lines 68 - 138
```

And... done! *Just* by passing the badge, the CSRF token will be validated.

Oh, and while we don't *need* to do this, I'm also going to pass a `new RememberMeBadge()`. If you use the "Remember Me" system, then you need to pass this badge. It tells the system that you opt "into" having a remember me cookie set if the user logs in using this authenticator. But you *still* need to have a "Remember Me" checkbox here... for it to work. Or, to *always* enable it, add `->enable()` on the badge.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 39
40     public function authenticate(Request $request): Passport
41     {
↕ // ... lines 42 - 44
45         return new Passport(
↕ // ... lines 46 - 57
58             new PasswordCredentials($password),
59             [
↕ // ... lines 60 - 63
64                 (new RememberMeBadge())->enable(),
65             ]
66         );
67     }
↕ // ... lines 68 - 138
```

And, of course, none of this will work unless you activate the `remember_me` system under your firewall, which I don't actually have yet. It's still safe to add that badge... but there won't be any system to process it and add the cookie. So, the badge will be ignored.

Deleting Old Methods!

Anyways, we're done! If that felt overwhelming, back up and watch our Symfony Security tutorial to get more context.

The cool thing is that we don't need `getCredentials()`, `getUser()`, `checkCredentials()`, *or* `getPassword()` anymore. All we need is `authenticate()`, `onAuthenticationSuccess()`, `onAuthenticationFailure()`, and `getLoginUrl()`. We can even celebrate up here by removing a *bunch* of old use statements. Yay!

Oh, and look at the constructor. We no longer need `CsrfTokenManagerInterface` or `UserPasswordHasherInterface`: both of those checks are now done somewhere *else*. And... that gives us two *more* `use` statements to delete.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 28
29     public function __construct(private SessionInterface $session, private
       EntityManagerInterface $entityManager, private UrlGeneratorInterface $urlGenerator)
30     {
31     }
↕ // ... lines 32 - 87
```

Activating the New Security System

At this point, our one custom authenticator *has* been moved to the new authenticator system. This mean that, in `security.yaml`, we are ready to switch to the new system! Say `enable_authenticator_manager: true`.

config/packages/security.yaml

```
1  security:
↕  // ... lines 2 - 9
10     enable_authenticator_manager: true
↕  // ... lines 11 - 64
```

And these custom authenticators aren't under a `guard` key anymore. Instead, add `custom_authenticator` and add this directly below that.

config/packages/security.yaml

```
1  security:
↕  // ... lines 2 - 20
21     firewalls:
↕  // ... lines 22 - 24
25         main:
↕  // ... lines 26 - 27
28             custom_authenticator:
29                 - App\Security\LoginFormAuthenticator
↕  // ... lines 30 - 63
```

Okay, moment of truth! We just *completely* switched to the new system. Will it work? Head back to the homepage, reload and... it does! And check out those deprecations! It went from around 45 to 4. Woh!

Some of those relate to *one* more security change. Next: let's update to the new `password_hasher` & check out a new command for debugging security firewalls.

Chapter 14: Password encoders -> password_hashers & debug:firewall

By converting to the new security system, our deprecations just went *way* down. If you look at what's left, one of them says:

> *"The child node "encoders" at path "security" is deprecated, use "password_hashers" instead."*

This is *another* change that we saw when upgrading the `security-bundle` recipe. Originally, we had `encoders`. This tells Symfony which algorithm to use to hash passwords. This has been renamed to `password_hashers`. And instead of needing our custom class, we can always just use this config. This says:

> *"Any class that that implements `PasswordAuthenticatedUserInterface` should use the `auto` algorithm."*

config/packages/security.yaml

```
1  security:
↕  // ... lines 2 - 11
12     password_hashers:
13         Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
↕  // ... lines 14 - 63
```

And since... *every* user class with a password needs to implement this - including our class - that covers us.

Oh, but if you had a *different* algorithm before, move that down to this line. We don't want to *change* the algorithm: we just want to delete `encoders` in favor of `password_hashers`.

Now, on the homepage... we have even less deprecations! Two left! Let's try to log in. Ah! I think I missed some conflicts in my base layout earlier.

Let's swing over and fix these. In `templates/base.html.twig`... yep. When we upgraded the `twig-bundle` recipe, this conflicted and I didn't even notice! Shame on me!

Now... much better. Let's log in: we have a user called `abraca_admin@example.com` with password `tada`. Sign in and... it's alive!

The debug:firewall Command

Speaking of "security" and "firewalls" and other nerdery, Symfony ships with a new command to help debug and *visualize* your firewall. It's called, appropriately, `debug:firewall`. If you run it with no arguments:

```
php bin/console debug:firewall
```

It'll tell you your firewall names: `dev` and `main`. Re-run this with `main`:

```
php bin/console debug:firewall main
```

Here we go! This tells us what authenticators this firewall has, which user provider it's using - though our app usually only has one - and also the entry point, which is something we talk about in our Security tutorial.

Ok, put a big ol' check mark next to "Upgrade Security". Next, let's *crush* the last few deprecations and learn how we can be sure that we didn't miss any.

Chapter 15: Hunting Down the Final Deprecations

All right team! Let's fix these last few deprecations. One of the trickiest things about these is that, sometimes, they come from third-party bundles. I don't have any examples here, but sometimes you'll get a deprecation and... if you look into it, you'll realize it's not your fault. It's coming from a library or a bundle you're using. When this happens, you need to upgrade that bundle, and *hope* there's a new version without any deprecations. We actually *did* have some examples of this *way* back at the beginning of the tutorial. But... we've already run `composer update` a few times, and have, apparently, upgraded all of our dependencies to versions *without* deprecations. Yay, efficiency!

ROLE_PREVIOUS_ADMIN -> IS_IMPERSONATOR

Ok, let's take a look at this list. It says that, in Symfony 5.1, `ROLE_PREVIOUS_ADMIN` is deprecated and we should use `IS_IMPERSONATOR` instead. You can show the context or trace to try to get more info, like *where* this is coming from. It isn't always obvious... and that's one of the trickiest things about deprecations. But *this* one is coming from `base.html.twig`.

Great! Open `templates/base.html.twig` and search for "previous_admin". In an earlier tutorial, we used this to check if we are currently impersonating a user with Symfony's `switch_user` feature. If we *are*, we changed the background to red to make it really obvious.

To fix the deprecation, very simply, change this to `IS_IMPERSONATOR`. Copy that... because there's one other spot on this page where we need to do the same thing: `IS_IMPERSONATOR`. Done! One less deprecation!

templates/base.html.twig

```
↕ // ... lines 1 - 21
22          <nav
↕ // ... line 23
24              {{ is_granted('IS_IMPERSONATOR') ? 'style="background-color: red !important"' }}
25          >
↕ // ... lines 26 - 60
61                              <ul class="dropdown-menu dropdown-menu-end" aria-labelledby="user-
    dropdown">
62                                  {% if is_granted('IS_IMPERSONATOR') %}
↕ // ... lines 63 - 67
68                                  {% endif %}
↕ // ... lines 69 - 71
72                              </ul>
↕ // ... lines 73 - 79
80          </nav>
↕ // ... lines 81 - 95
```

IS_AUTHENTICATED_ANONYMOUSLY -> PUBLIC_ACCESS

While we're talking security, open up `config/packages/security.yaml` and head down to `access_control`. I have a few entries - `/logout`, `/admin/login` - that I want to make *absolutely* sure are accessible by *everyone*, even users that are *not* logged in. To do, we added these rules on top and, *previously* used

`IS_AUTHENTICATED_ANONYMOUSLY`. So if I go to `/logout`, *only* this `access_control` is matched... and since the `role` is `IS_AUTHENTICATED_ANONYMOUSLY` access is *always* granted.

In Symfony 6, `IS_AUTHENTICATED_ANONYMOUSLY` has changed to `PUBLIC_ACCESS`. So use that in both places.

config/packages/security.yaml

```
security:
// ... lines 2 - 38
    access_control:
// ... lines 40 - 41
        - { path: ^/logout, role: PUBLIC_ACCESS }
        - { path: ^/admin/login, roles: PUBLIC_ACCESS }
// ... lines 44 - 59
```

If you're wondering why we didn't have a deprecation for this... well... it's a rare case where Symfony is unable to *catch* that deprecated path and show it to us. This doesn't happen very often, but it's a situation where a tool like SymfonyInsight can help catch this.... even when Symfony itself can't.

The Deprecated Session Service

Okay, the last deprecation on the list says:

> *"`SessionInterface` aliases are deprecated, use `$requestStack->getSession()` instead. It's being referenced by the `LoginFormAuthenticator` service."*

Let's go check that out! Open `src/Security/LoginFormAuthenticator.php`. Ahh. I'm autowiring the `SessionInterface` service. In Symfony 6, that service *no* longer exists. There are some technical reasons for this... but long story short, the session wasn't ever, really a *true* service. What you're supposed to do now is get it from the `Request`.

So, no big deal. Remove the `SessionInterface` constructor argument... and we don't need this `use` statement anymore either.

src/Security/LoginFormAuthenticator.php

```
// ... lines 1 - 21
class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
{
// ... lines 24 - 27
    public function __construct(private EntityManagerInterface $entityManager, private
UrlGeneratorInterface $urlGenerator)
    {
    }
// ... lines 31 - 84
}
```

Now search for "session". We're using it down in `onAuthenticationSuccess()`. Fortunately, this already passes us the `$request` object! So we can just say `$request->getSession()`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 60
61     public function onAuthenticationSuccess(Request $request, TokenInterface $token, string
       $firewallName): Response
62     {
63         if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
↕ // ... line 64
65         }
↕ // ... lines 66 - 67
68     }
↕ // ... lines 69 - 86
```

Hunting Down the Final Deprecations

Done! So... did we do it? Have we achieved zero deprecations and spiritual enlightenment? Go back to the homepage, refresh and... we did! Well, at least that first part... no deprecations! And if we surf around our site a bit... I'm not seeing any deprecations on *any* of these pages!

Does this mean we're done? Well, we've manually tested all of the pages that we can click on. But what about `POST` requests... like submitting the login or registration forms? And what about API endpoints? We have one called `/api/me`... which doesn't work because I'm not logged in. Log *back* in as "abraca_admin@example.com" with password "tada" and then... yea, `/api/me` works.

We can't see the web debug toolbar for this request, but I bet you already know the trick. Go to `/_profiler` to see the last ten requests. Here's the `POST` request to `/login`. Go down to Logs. Great! That had no deprecations. Go back and also check the API endpoint. If we look at Logs again, it *also* had no deprecations. We're on a roll!

Another option, instead of checking the profiler all the time, is to go over to your terminal and tail the log file:

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

This will *constantly* stream any new logs. Actually, hit "ctrl" + "C" and run that again, but grep for `deprecation`:

```
tail -f var/log/dev.log | grep deprecation
```

Perfect. Now, if any logs come through that contain the word "deprecation", we'll see them. And since deprecated code paths trigger a log in the `dev` environment, this is a powerful tool.

Deprecated $this->getDoctrine() Method

For example, let's go register as a new user. I'll log out, then "Sign up". It asks me for my name, email, and a password. Click to "Agree" to some made-up terms and submit. Oh, my password is too short: my own validation rules coming back to haunt me! Fix that, hit "Register" again and... it works!

But if we go *back* to our terminal... rut roo!

> *"Since symfony/framework-bundle 5.4, method `AbstractController::getDoctrine()` is deprecated. Inject an instance of `ManagerRegistry` in your controller instead."*

It's not easy to see where this is coming from in our code, but we *did* just register... so let's open up `RegistrationController`. Ah, it's complaining about this right here: the `getDoctrine()` method is deprecated.

Instead of using this, we can inject the `$entityManager`. At the end of the argument list, autowire `EntityManagerInterface $entityManager`. And... then down here, delete this line because `$entityManager` is now being injected. Another deprecation gone!

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 7
8  use Doctrine\ORM\EntityManagerInterface;
↕ // ... lines 9 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... line 19
20     public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
   VerifyEmailHelperInterface $verifyEmailHelper, EntityManagerInterface $entityManager): Response
21     {
↕ // ... lines 22 - 56
57     }
↕ // ... lines 58 - 88
89 }
```

Logging Deprecations on Production

Are we done *now*? *Probably.* Our project is pretty small, so checking all the pages manually isn't that big of a deal. But for bigger projects, it might be... a *huge* deal to check everything manually! And you *really* want to be sure that you didn't miss anything before you upgrade.

One great option to make sure you didn't miss anything is to *log* your deprecations on production. Open `config/packages/monolog.yaml` and go down to `when@prod`. This has a number of handlers that will log all errors to `php://stderr`. There's also a `deprecation` section. With this config, Symfony will log any deprecation messages (that's what this `channels: [deprecation]` means) to `php://stderr`.

config/packages/monolog.yaml

```
↕ // ... lines 1 - 40
41 when@prod:
42     monolog:
43         handlers:
↕ // ... lines 44 - 58
59             deprecation:
60                 type: stream
61                 channels: [deprecation]
62                 path: php://stderr
```

This means that you can deploy, wait for an hour, day or week, then... just check the log! If you want to log to a file instead, change the path to something like `%kernel.logs_dir%/deprecations.log`.

So that's *my* favorite thing to do: deploy it, and then see - in the *real* world - whether or not anyone is still hitting deprecated code paths.

At this point, I'm not seeing any more deprecations on our web debug toolbar, so I think we're done! And *that* means we're ready for Symfony 6! Let's do the upgrade next!

Chapter 16: Upgrading to Symfony 6.0

Finally, it's time to upgrade to Symfony 6! Woo!

Rector Upgrades to 6.0

But first, just in case, let's run Rector *one* more time. Go back to Rector's repository, click the Symfony link, and... steal the same code that we had earlier. Paste that into our `rector.php` file. Then, just like we did for Symfony 5.4, change `SymfonySetList` to `SymfonyLevelSetList`, and this time, say `UP_TO_SYMFONY_60`.

```
rector.php
↕ // ... lines 1 - 14
15 return static function (ContainerConfigurator $containerConfigurator): void {
↕ // ... lines 16 - 23
24     $containerConfigurator->import(SymfonyLevelSetList::UP_TO_SYMFONY_60);
25     $containerConfigurator->import(SymfonySetList::SYMFONY_CODE_QUALITY);
26     $containerConfigurator->import(SymfonySetList::SYMFONY_CONSTRUCTOR_INJECTION);
27 };
```

In theory, there shouldn't be *any* code differences needed between Symfony 5.4 and 6.0... though sometimes there are minor cleanups you can do once you *have* upgraded.

Let's run this and see what happens. Say:

```
vendor/bin/rector process src/
```

And... okay. It made one change. This is to our event subscriber: it added an `array` return type. This was done because, in the future, this interface may add an `array` return type. So now our code is *future* compatible.

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 9
10 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
11 {
↕ // ... lines 12 - 24
25     public static function getSubscribedEvents(): array
26     {
↕ // ... lines 27 - 29
30     }
31 }
```

Upgrading via Composer

With that done, let's upgrade! In `composer.json`, we need to find the main Symfony libraries and change their version from `5.4.*` to `6.0.*`. Let's take the lazy way out and do that with a "Find & Replace".

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 21
        "symfony/asset": "6.0.*",
        "symfony/console": "6.0.*",
        "symfony/dotenv": "6.0.*",
// ... line 25
        "symfony/form": "6.0.*",
        "symfony/framework-bundle": "6.0.*",
// ... line 28
        "symfony/property-access": "6.0.*",
        "symfony/property-info": "6.0.*",
        "symfony/proxy-manager-bridge": "6.0.*",
        "symfony/routing": "6.0.*",
        "symfony/runtime": "6.0.*",
        "symfony/security-bundle": "6.0.*",
        "symfony/serializer": "6.0.*",
        "symfony/stopwatch": "6.0.*",
        "symfony/twig-bundle": "6.0.*",
// ... line 38
        "symfony/validator": "6.0.*",
// ... line 40
        "symfony/yaml": "6.0.*",
// ... lines 42 - 45
    },
    "require-dev": {
// ... lines 48 - 50
        "symfony/debug-bundle": "6.0.*",
// ... line 52
        "symfony/var-dumper": "6.0.*",
        "symfony/web-profiler-bundle": "6.0.*",
// ... line 55
    },
// ... lines 57 - 103
    "extra": {
        "symfony": {
// ... line 106
            "require": "6.0.*"
        }
    }
}
```

Awesome! Like before, we're not touching any Symfony libraries that are *not* part of the main package and which follow their *own* versioning scheme. Oh, and at the bottom, this *did* also change `extra.symfony.require` to `6.0.*`.

So, we're ready! Just like before, we *could* say:

```
composer up 'symfony/*'
```

But... I'm not going to bother with that. Let's update *everything* with just:

```
composer up
```

And... it fails! Hmm. One of the libraries I'm using is `babdev/pagerfanta-bundle`... and apparently it requires PHP 7.2... but we're using PHP 8. If you look further, there are some errors about `pagerfanta-bundle[v2.8.0]` requiring `symfony/config ^3.4 || ^4.4 || ^5.1`, but not Symfony 6. So what's happening here? It turns out that `pagerfanta-bundle[v2.8.0]` does *not* support Symfony 6. Gasp!

Run

```
composer outdated
```

to see a list of outdated packages. Oooh! `babdev/pagerfanta-bundle` has a new version `3.6.1`. Go into `composer.json` and find that... here it is! Change its version to `^3.6`.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 9
        "babdev/pagerfanta-bundle": "^3.6",
// ... lines 11 - 45
    },
// ... lines 47 - 109
}
```

This *is* a *major* version upgrade. So it *may* contain some backwards compatibility breaks. We'll check into that in a minute. Try:

```
composer up
```

again and... it's doing it! Everything just upgraded to Symfony 6!

Fixing PasswordUpgraderInterface::upgradePassword()

And then... to celebrate... it immediately *exploded* while clearing the cache. Uh oh... I think we may have missed a deprecation:

> *"In `UserRepository`, `upgradePassword([...]): void` must be compatible with `PasswordUpgraderInterface`."*

If you want to see this in color, you can refresh the homepage to see the same thing.

By the way, in Symfony 5.4, we can now click this icon to copy the file path to our clipboard. Now, if I go back over to my editor, hit "shift" + "shift" and paste, I jump *directly* to the file - and even the *line* - where the problem is.

And... phew! PhpStorm is *not* happy. That's because the `upgradePassword()` method changed from requiring a `UserInterface` to requiring a `PasswordAuthenticatedUserInterface`. So we just need to change that and... done!

src/Repository/UserRepository.php

```
// ... lines 1 - 8
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
// ... lines 10 - 18
class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
{
// ... lines 21 - 28
    public function upgradePassword(PasswordAuthenticatedUserInterface $user, string $newHashedPassword): void
    {
// ... lines 31 - 37
    }
}
```

Back at our terminal, if we run:

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

Now it's happy. We're still getting some deprecations down here from a different library... but I'm going to ignore those. These come from a deprecated package that... I really just need to remove from this project entirely.

PagerFanta Updates

Let's go make sure the homepage works. It... doesn't!? We get

> *"Attempted the load class `QueryAdapter` from namespace "`Pagerfanta\Doctrine\ORM`."*

This shouldn't be a surprise... since we *did* upgrade pagerfanta-bundle from 2.8 to 3.6.

This is a situation where you need to find the GitHub page for the library and *hope* that they have an upgrade document. This one actually *does*. If you read this closely, you'd discover that a bunch of classes that were *previously* part of Pagerfanta have now been broken into *independent* libraries. So if we want to use this `QueryAdapter`, we need to install a separate package. Do that with:

```
composer require pagerfanta/doctrine-orm-adapter
```

Cool... and if we refresh now... another error? This one's even better:

> *"Unknown function `pagerfanta`. Did you forget to run `composer require pagerfanta/twig` in `question/homepage.html.twig`?."*

The Twig integration was *also* moved to its own package... so we need to run that command too:

```
composer require pagerfanta/twig
```

And... after that's done... it's *alive*! We have a Symfony 6 project! Woohoo! If we click around, things seem to be working just fine. We did it!

Checking for Outdated Packages

Over at our command line, run

```
composer outdated
```

to see all of the outdated packages we have left. The list is now *very* short. One package is `knplabs/knp-markdown-bundle`, which *is* fully upgraded... but it's been abandoned. If you have this in a real project, refactor it to use `twig/markdown-extra`. I'm not going to bother, but that's why it's on this list.

The biggest thing here is that `doctrine/dbal` has a new *major* version! So hey! While we're here upgrading things, let's upgrade it too! That's next, along with some final cleanups.

Chapter 17: Final Upgrades & Cleanups

While we're doing all of these major upgrades, we might as well make sure *everything* is upgraded. When we run

```
composer outdated
```

it gives us a list of all of the things we still need to update. As I mentioned, we're going to ignore `knplabs/knp-markdown-bundle`. But if you have that in a real project, refactor it to use `twig/markdown-extra`.

Upgrading doctrine/dbal to v3

What *I'm* interested in is `doctrine/dbal`, which has a new *major* version we can upgrade to. But... this begs the question: Why didn't this upgrade *automatically* when we did `composer up`? Run

```
composer why-not doctrine/dbal 3
```

to find out what is preventing us from upgrading to version *3* of this package. Of course! *We're* holding it back. It says that *our* project requires `doctrine/dbal (^2.13)`. Whoops...

Head over to `composer.json` and... sure enough: `^2.13`. Change that to the latest `^3.3`. Moment of truth. Run

composer.json

```
1   {
↕   // ... lines 2 - 5
6       "require": {
↕   // ... lines 7 - 12
13          "doctrine/dbal": "^3.3",
↕   // ... lines 14 - 47
48      },
↕   // ... lines 49 - 111
112 }
```

```
composer up
```

And... woo! It updated! Do

```
composer outdated
```

again. Alright! Other than `knp-markdown-bundle`, this is empty.

We *did* just perform a *major* version upgrade. So the new version *does* contain backwards-compatibility breaks. You'll want to look into the library's CHANGELOG a bit deeper to make sure you're not affected. *But*, I can tell you that most of the changes relate to if you're using `doctrine/dbal` directly, for example to make queries direclty in DBAL. Typically, when you're working with the Doctrine ORM & entities - even if you're making custom queries - you're not doing that. On our site... we seem to be just fine.

Final Recipe Upgrades

Now that we've upgraded from Symfony 5.4 to 6.0, it's *possible* that some recipes have new versions we can update to. Run:

```
composer recipes:update
```

Oh, whoops! I need to commit my changes:

```
git commit -m 'upgrading doctrine/dbal from 2 to 3'
```

Perfect! *Now* run

```
composer recipes:update
```

and... cool! There are *two*. Start with `symfony/routing`. And... we have conflicts! Run:

```
git status
```

Moving Route Attribute Loading

The problem is in `config/routes.yaml`. Let's check that out. Ok, so previously, *I* commented out this route.

config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\QuestionController::homepage
```

The recipe update added the `controllers` and `kernel` imports. Let's keep *their* changes. These are actually importing our route annotations or attributes from the `../src/Controller` directory... and also allowing you to add routes and controllers directly to your `Kernel.php` file.

config/routes.yaml

```yaml
controllers:
    resource: ../src/Controller/
    type: annotation

kernel:
    resource: ../src/Kernel.php
    type: annotation
```

It says `type: annotation`... but that importer is able to load annotations *or* PHP 8 attributes. One of the nice things about Symfony 6 is that you can load route *attributes* without *any* external library. It's just... part of the routing system. For that reason, these route imports were added to our main `config/routes.yaml` file when we install `symfony/routing`.

Go ahead and commit that. This change will make even *more* sense after we upgrade the *final* recipe.

Run

```
composer recipes:update
```

again and, this time, let's update the `doctrine/annotations` recipe. Interesting. It *deleted* `config/routes/annotations.yaml`. If you look closely, that actually contained the two lines that were added by the *previous* recipe update!

Here's the deal. Back before PHP 8 - when we only had annotation routes - the `doctrine/annotations` library was *required* to import route annotations. So we only *gave* you these imports lines once you *installed* that library.

But now that we use *attribute* routes, that's no longer true! We do *not* need the `doctrine/annotations` package anymore. For that reason, we now *immediately* give you these attribute route import lines as *soon* as you install the routing component.

If we look over here, nothing changes on our front end. All of our routes *still* work.

Removing Un-needed Code

Finally, now that we're on Symfony 6, we can remove some code that was only needed to keep things working on Symfony 5. There's not much of this that I know of... the only I can think of is in `User.php`.

As I mentioned earlier, in Symfony 6, `UserInterface`... I'll click into that... renamed `getUsername()` to `getUserIdentifier()`. In Symfony 5.4, to remove the deprecations but keep your code working, we need to have *both* of these methods. But as soon as you upgrade to Symfony 6, you don't need the old one anymore! Just make sure that you're not calling this directly from *your* code.

src/Entity/User.php

```
↕ // ... lines 1 - 14
15 class User implements UserInterface, PasswordAuthenticatedUserInterface
16 {
↕ // ... lines 17 - 79
80     /**
81      * A visual identifier that represents this user.
82      *
83      * @see UserInterface
84      */
85     public function getUsername(): string
86     {
87         return (string) $this->email;
88     }
↕ // ... lines 89 - 221
222 }
```

Another spot down here... is `getSalt()`. This is an old method related to how you hash passwords, and it's no longer needed in Symfony 6. Modern password hashing algorithms take care of the salting themselves, so this is completely useless.

And... that's it team! We're done! Our Symfony 6 app is fully upgraded! We upgraded recipes, updated to PHP 8 code, are using PHP 8 attributes instead of annotations and more. That was a *ton* of stuff... and we just modernized our codebase *big time*.

I think this deserves a *whole* pizza to celebrate. Then come right back, because I want to take a quick test drive of a few more new features that we haven't talked about. Those are next.

Chapter 18: Form Improvements for Symfony 6

Let's explore some new features! There are *tons* of them, and we've already seen a bunch. I don't have time to show *everything* but *fortunately*, I don't need to! If you go to https://symfony.com/blog, the new stuff is *really* well-documented. Click on "Living on the Edge". Here, you can see blog posts that are categorized by each version. This is a collection of blog posts about what's new in Symfony 5.1, like the new security system. And... here are posts about what's new in Symfony 5.3, *or* 5.4 through 6.0. So if you want to go deeper and see *all* the new stuff, it's been *beautifully* documented in these posts.

The new features *I* want to show right now have to do with the form component.

Form Field Sorting

Since Symfony 5.3, we have a nice new feature called Form Field Sorting. If you go to the registration page, this renders four fields. Let's open the template for that: `templates/registration/register.html.twig`. I'm rendering all the fields by hand. Let's *replace* this with the very lazy `{{ form_widget(registrationForm) }}`... which just dumps out all of the fields in whatever order they're added.

templates/registration/register.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
↕ // ... lines 6 - 10
11                 {{ form_start(registrationForm) }}
12                     {{ form_widget(registrationForm) }}
↕ // ... line 13
14                 {{ form_end(registrationForm) }}
↕ // ... lines 15 - 17
18 {% endblock %}
```

Unfortunately... now the form... looks *weird*. To fix this, open the form type class for this, which is `src/Form/RegistrationFormType.php`. Every single field now has an option called `priority`. Let's add that.

Starting with `firstName`, pass `null` for the type so Symfony keeps guessing. Then, set `priority` to `4`, because I want this to be the first field. `email` should be the second field, so pass `null` again and set its `priority` to `3`. Then give `plainPassword` a `priority` of `2`... and finally set `agreeTerms` to `priority` `1`.

src/Form/RegistrationFormType.php

```
// ... lines 1 - 16
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('email', null, [
                'priority' => 3,
            ])
            ->add('agreeTerms', CheckboxType::class, [
                'priority' => 1,
// ... lines 25 - 30
            ])
            ->add('firstName', null, [
                'priority' => 4,
            ])
            ->add('plainPassword', PasswordType::class, [
                'priority' => 2,
// ... lines 37 - 52
            ])
        ;
    }
// ... lines 56 - 62
}
```

And now... it looks great! So if you want to lazily render your fields, you can do that... and not have to worry about them being in a strange order.

Hello renderForm()

While we're on the topic of forms, open up the controller for this page: `src/Controller/RegistrationController.php`. In Symfony 5.3, when you render a template and pass in a form, there's a new shortcut! Instead of `render()` say `renderForm()`. The only other difference is that you get to *remove* the `->createView()` call.

src/Controller/RegistrationController.php

```
// ... lines 1 - 16
class RegistrationController extends AbstractController
{
// ... line 19
    public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper, EntityManagerInterface $entityManager): Response
    {
// ... lines 22 - 53
        return $this->renderForm('registration/register.html.twig', [
            'registrationForm' => $form,
        ]);
    }
// ... lines 58 - 88
}
```

That's it! this `renderForm()` method is *just like* `render()`. It *still* renders this template, and it *still* passes any of these variables *into* the template. But if any of the variables we're passing are a "form" object, it calls the `createView()` method *for* us... which is nice!

This also makes one other change, which isn't very noticeable. If you have a validation error, your controller will now return a response with its status code set to 422. But that won't look any different in your browser. If I submit a password that's too short, it all looks the same... though the status code *is* now 422.

Symfony made this change for two reasons. First... it's just technically more correct to have an *error* status code if there is a validation error. And second, if you're using Turbo, this is needed so that Turbo knows that your form validation failed. You get that for free just by using the new shortcut method.

Next, Symfony comes with some nice and *optional* Docker integration for local development. Some parts of this integration have recently changed. Let's see how we can use Docker to get a cool email catching system added to our app that will help us *test* emails.

Chapter 19: Enhanced Docker Integration & Testing Emails

Symfony has had Docker support for a while, in particular, to help with local web development. For example, I have PHP installed locally. So I'm not using Docker to get PHP itself. *But* my project has a `docker-compose.yml` file that defines a database service. Remember that the local web server we're using comes from the Symfony binary... and it's *smart*. It *automatically* detects that I have `docker-compose` running with a `database` service... and so it reads the connection parameters from this container and exposes them as a `DATABASE_URL` environment variable.

Check this out! On any page, click into the web debug toolbar. Make sure you're on "Request/ Response", then go to "Server Parameters". Scroll down to find `DATABASE_URL` set to (in my case) `127.0.0.1` on port `56239`. The way my `docker-compose.yml` is set up, it will create a new random port each time it starts.

docker-compose.yml

```
// ... line 1
services:
    database:
        image: 'mysql:8.0'
        environment:
            MYSQL_ROOT_PASSWORD: password
        ports:
            # To allow the host machine to access the ports below, modify the lines below.
            # For example, to allow the host to connect to port 3306 on the container, you would change
            # "3306" to "3306:3306". Where the first port is exposed to the host and the second is the container port.
            # See https://docs.docker.com/compose/compose-file/#ports for more information.
            - '3306'
```

The Symfony binary will then figure out *which* random port it is and create the environment variable accordingly. Finally, just like normal, thanks to our `config/packages/doctrine.yaml` configuration, the `DATABASE_URL` environment variable is used to talk to the database. So the Symfony binary *plus* Docker is a nice way to quickly and easily boot up external services like a database, elastic search, or more.

New Docker Integration with Flex Recipes

Recently, Symfony took this to the next level. On Symfony.com, you'll find a blog post called Introducing Docker support. The idea is pretty simple. When you install a new package - Doctrine, for example - that package's recipe *may* ship with some Docker configuration. And so, just by installing the package, you get Docker configuration *automatically*.

Let's see this in action! Since we already have Doctrine installed, let's install Mailer, which will come with `docker-compose` config for a service called MailCatcher. At your terminal, run:

```
composer require mailer
```

Awesome! It stops us and asks:

> *"The recipe for this package contains some Docker configuration. Do you want to include Docker configuration from recipes?"*

I'm going to say `p` for "Yes permanently". If you *don't* want the Docker stuff, no worries! Answer no or "No permanently" and it will never ask you again.

And... done! Now we can run

```
git status
```

to see that it updated the normal stuff, but *also* gave us a new `docker-compose.override.yml`. If you're not familiar, Docker will first read `docker-compose.yml` and *then* will read `docker-compose.override.yml`. The purpose of the override file is to *change* configuration that is specific to your machine. In this case, our local machine.

```
docker-compose.override.yml
↕ // ... lines 1 - 2
3 services:
4 ###> symfony/mailer ###
5   mailer:
6     image: schickling/mailcatcher
7     ports: [1025, 1080]
8 ###< symfony/mailer ###
```

The new file adds a service called `mailer`... which boots up something called MailCatcher. MailCatcher is a local debugging tool that starts an SMTP server that you can send emails *to*. And then it gives you a web GUI where you can *review* those emails... inside a pretend inbox.

This service lives inside of `docker-compose.override.yml` because we only want this service to be running locally when we're doing *local* development. If you're using Docker to *deploy* your site, you'll have a different local configuration for production. If you're *not* deploying with Docker, all of this config could live in your main `docker-compose.yml` file if you want.

Testing MailCatcher

Anyways, before we even start using this service, let's get set up to send an email. Open up `src/Controller/RegistrationController.php`. We're already using `symfonycasts/verify-email-bundle`... but instead of actually *sending* the verification email, we're just putting the verification URL directly into a flash message. It was a shortcut I made during the Security tutorial.

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... line 19
20     public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
   VerifyEmailHelperInterface $verifyEmailHelper, EntityManagerInterface $entityManager): Response
21     {
↕ // ... lines 22 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 43
44             // TODO: in a real app, send this as an email!
45             $signedUrl = $signatureComponents->getSignedUrl();
46             $this->addFlash('success', sprintf(
47                 'Confirm your email at: %s',
48                 $signedUrl
49             ));
↕ // ... lines 50 - 51
52         }
↕ // ... lines 53 - 56
57     }
↕ // ... lines 58 - 88
89 }
```

But *now*, let's send a *real* email. I'll go to the bottom of the class and paste a new private function, which you can get from the code blocks on this page. Retype the "e" on `MailerInterface` and hit "tab" to add that `use` statement... and do the same with the "l" on `Email`. Select the one from `Symfony\Component\Mime`.

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\Mailer\MailerInterface;
10 use Symfony\Component\Mime\Email;
↕ // ... lines 11 - 19
20 class RegistrationController extends AbstractController
21 {
↕ // ... lines 22 - 92
93     private function sendVerificationEmail(MailerInterface $mailer, User $user, string
   $signedUrl)
94     {
95         $email = (new Email())
96             ->from('hello@example.com')
97             ->to($user->getEmail())
98             //->cc('cc@example.com')
99             //->bcc('bcc@example.com')
100            //->replyTo('fabien@example.com')
101            //->priority(Email::PRIORITY_HIGH)
102            ->subject('Verify your email on Cauldron Overflow!')
103            ->text('Please, follow the link to verify your email!')
104            ->html(sprintf('<a href="%s">%s</a>', $signedUrl, $signedUrl));
105
106        $mailer->send($email);
107    }
108 }
```

Perfect! This will send a very simple verification email that just contains the verification link.

Now, *all* the way up on the `register()` method, add a new argument at the end: `MailerInterface $mailer`. Then, down here, remove the `TODO`... and replace it with `$this->sendVerificationEmail()` passing `$mailer`, `$user`,

and `$signedUrl`. Finally, in the `success` flash, change the message to tell the user that they should check their email.

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 22
23     public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
       VerifyEmailHelperInterface $verifyEmailHelper, EntityManagerInterface $entityManager,
       MailerInterface $mailer): Response
24     {
↕ // ... lines 25 - 27
28         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 29 - 47
48             $this->sendVerificationEmail($mailer, $user, $signedUrl);
49             $this->addFlash('success', sprintf(
50                 'Confirm your email - the verify link was sent to %s',
51                 $user->getEmail()
52             ));
↕ // ... lines 53 - 54
55         }
↕ // ... lines 56 - 59
60     }
↕ // ... lines 61 - 109
```

Okay, so we have this new `docker-compose.override.yml` file with MailCatcher. However, that container isn't actually *running* yet. But, ignore that for a minute... and let's see if we can get the email working.

Click back to the Register page... whoops! We get an error:

> *"Environment variable not found: "MAILER_DSN"."*

Of course! The mailer service needs this environment variable to tell it *where* to send emails. You can find this inside `.env`: the mailer recipe gave us the `MAILER_DSN` env var, but it's commented-out. *Un-comment* that.

.env

```
↕ // ... lines 1 - 30
31 ###> symfony/mailer ###
32 MAILER_DSN=null://null
33 ###
```

By default, it sends emails to what's called the "null transport"... which means that when we send emails... they go absolutely nowhere. They're not *actually* delivered... which is a nice setting for development.

Refresh, add a fake email address, register, and... it worked! Of course, it didn't *send* the email anywhere... but we can still see, more or less, what the email would look like.

How? Click any link to go into the Profiler, click "Last 10", find the POST request for `/register` and click into that. Down here, go to the "E-mails" section and... voilà! It shows our email including an HTML preview. And *wow* is it ugly... but that's my fault. Btw, the HTML preview is a new feature in Symfony 5.4.

Starting up the MailCatcher Service

Ok that's cool. But let's see how MailCatcher can *also* help us debug emails. First, if you do *not* already have a `docker-compose.yml` file, create one. All you need is the `version` line on top. That way we have a

`docker-compose.yml` file *and* a `docker-compose.override.yml` file.

Now, find your terminal and run:

```
docker-compose up -d
```

I already have `docker-compose` running for my database container, but this will now start the `mailer` container, which will initialize a new mailcatcher SMTP server.

Ok... so how do we configure `mailer` to *deliver* to this smpt server from MailCatcher? What port is that SMTP server running on anyways? The answer is... we *don't know*! And we *don't care*.

Watch this. Go back to any page, refresh... and then click into the Profiler. Once again, make sure you're on the "Request/Response" section then go to "Server Parameters". Scroll down to `MAILER_URL`.

Woh! `MAILER_URL` is suddenly set to `smtp://127.0.0.1:65320`!

Here's what happened. When we started the `mailer` service, Docker exposed port `1025` of that container - which is the SMTP server - to a *random* port on my host machine. The Symfony binary *saw* that, *read* the random port, and then, just like with the database, exposed a `MAILER_URL` environment variable that *points* to it. In other words, our emails will already send to MailCatcher!

Let's try it! I'll sign up again with some other email address, agree to the terms and... cool! No error! To see the email, we *could* go back into the Profiler like we did a minute ago. But in theory, if that sent to MailCatcher, we *should* be able to go to the MailCatcher UI and review the message *there*. The question is, *where is* the `MailCatcher` UI? What port is *that* running on? Because that's *also* running on a random port.

To help with this, hover over the "Server" section of the web debug toolbar. You can see that it detects that `docker-compose` is running, it *is* exposing some environment variables from Docker, and it even detected Webmail! Click "Open" to head into MailCatcher... and *there's* our email!

If you send *more* emails, they'll show up here like a little inbox.

And... that's it! Congrats! You've just upgraded your app to Symfony 6! *And* PHP 8! *And* PHP attributes! Such cool stuff!

If you have any questions or run into any problems during your upgrade that *we* didn't talk about, we're here for you down in the comments. All right, friends, seeya next time!