

# Upgrade to Symfony4 and Flex!



With <3 from SymfonyCasts

# Chapter 1: Upgrade to Symfony 3.4

Symfony 4: it's a *game* changer. It's my *favorite* thing since chocolate milk. Honestly, I've *never* been so excited to start writing tutorials: you are going to *love* it!

But first, we need to talk about how you can *upgrade* to Symfony 4 so that all your projects can enjoy the goodness!

There are two big steps. First, well, of course, we need to *actually* upgrade our app to Symfony 4! And second, we need to update our project to support Symfony *Flex*. *That* is where things get interesting.

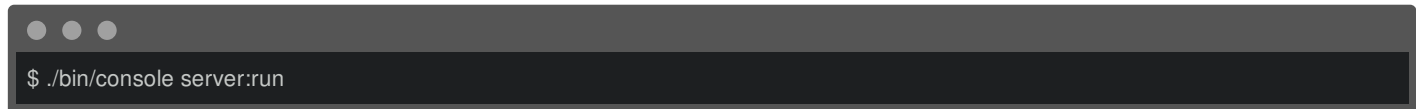
Upgrading *does* take some work. But don't worry: we'll walk through it together and learn a ton on the way.

## [Download the Course Code](#)

As always, if you *really* want to get a handle on upgrading... or you just like chocolate milk, you should download the course code from this page and code along. Full disclosure: the download does *not* contain chocolate milk.

But, after you unzip it, it *does* contain a `start/` directory with the same code you see here. Follow the README.md file for the steps needed to get your project running.

The last step will be to find your terminal and run:

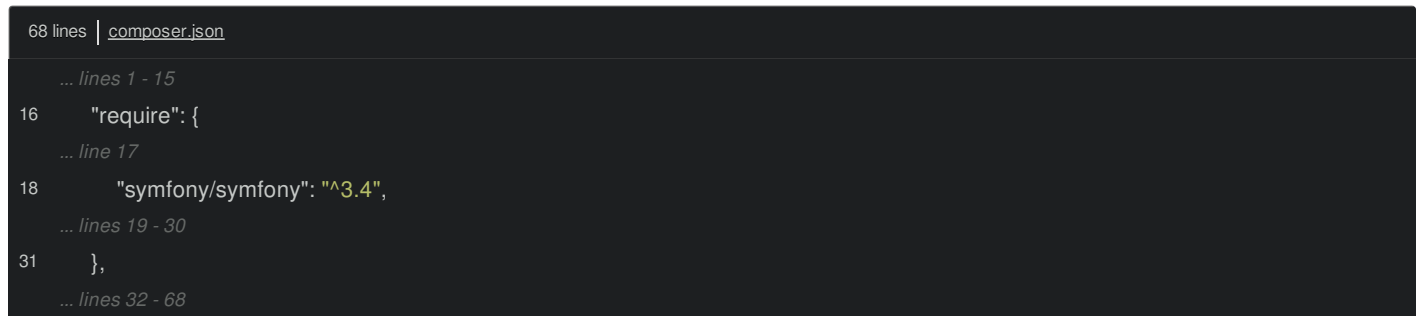


```
$ ./bin/console server:run
```

to start the built-in web server. Check it out at `http://localhost:8000`. Hello AquaNote! This is the Symfony 3.3 project that we've been building in our Symfony series.

## [Upgrading to Symfony 4](#)

So: what's the first step to upgrading to Symfony 4? It's upgrading to 3.4! And that's *delightfully* simple: Open `composer.json`, change the version of `symfony/symfony` to `^3.4`, find your terminal and run:



```
68 lines | composer.json
```

```
... lines 1 - 15
```

```
16     "require": {
```


```
... line 17
```

```
18         "symfony/symfony": "^3.4",
```

```
... lines 19 - 30
```

```
31     },
```

```
... lines 32 - 68
```



```
$ composer update
```

And celebrate! So easy! By the way, you *could* just update `symfony/symfony`. But, honestly, it's just *easier* to upgrade *everything*. And since I keep responsible version constraints in `composer.json`, ahem, no `dev-master` or `*` versions, this is pretty safe and also means I get bug fixes, security fixes and new features.

And... hello Symfony 3.4! The best part? Ah, you guys already know it: thanks to Symfony's backwards-compatibility promise, our project will just work... immediately. Refresh! Yep! Welcome to Symfony 3.4.

## [Symfony 3.4 Versus Symfony 4.0](#)

So why did we do this? Why not just skip *directly* to Symfony 4? Well, Symfony 3.4 and Symfony 4.0 are *identical*: they have the exact same features. The *only* difference is that all deprecated code paths are *removed* in 4.0.

On the web debug toolbar, you can see that *this* page contains 9 deprecated code calls. By upgrading to Symfony 3.4 first, we can hunt around and fix these. As soon as they're all gone... well, we'll be ready for Symfony 4!

### [Deprecation: Kernel::loadClassCache\(\)](#)

Click the icon to go see the full list of deprecations. Check out the first one: Kernel::loadClassCache() is deprecated since version 3.3. You can click "Show Trace" to see where this is coming from, but I already know!

Open web/app.php and web/app\_dev.php. There it is! On line 28, remove the \$kernel->loadClassCache() line. Do the same thing in app.php. Why is this deprecated? This *originally* gave your app a performance boost. But thanks to optimizations in PHP 7, it's not needed anymore. Less code, more speed, woo!

### [Deprecation: GuardAuthenticator::supports\(\)](#)

Close those files. What's next? Hmm, something about AbstractGuardAuthenticator::supports() is deprecated. Oh, and a recommendation! We should implement supports() inside our LoginFormAuthenticator.

Because I obsess over Symfony's development, I know what this is talking about. If you have more of a life than I do and are not already aware of every single little change, you should go to [github.com/symfony/symfony](https://github.com/symfony/symfony) and find the UPGRADE-4.0.md file. It's not perfect, but it contains explanations behind a *lot* of the changes and deprecations you'll see.

Go find the LoginFormAuthenticator in src/AppBundle/Security. We need to add a new method: public function supports() with a Request argument.

Copy the logic from getCredentials() that checks the URL, and just *return* it. Here's the deal: in Symfony 3, getCredentials() was called on *every* request. If it returned null, the authenticator was done: no other methods were called on it.

```
79 lines | src/AppBundle/Security/LoginFormAuthenticator.php
... lines 1 - 15
16 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
17 {
... lines 18 - 30
31 public function supports(Request $request)
32 {
33     return $request->getPathInfo() == '/login' && $request->isMethod('POST');
34 }
... line 35
36 public function getCredentials(Request $request)
37 {
... lines 38 - 47
48 }
... lines 49 - 77
78 }
```

In Symfony 4, supports() is now called on every request instead. If it returns false, the authenticator is done like before. But if it returns true, then getCredentials() is called. We split the work of getCredentials() into two methods.

So, remove the logic at the top of it: we know this will *only* be called when the URL is /login and it's a POST request.

### [Deprecation: Quoting % in YAML](#)

Most of the other deprecations are pretty easy, like the next one:

Not quoting the scalar %cache\_type% starting with the "%" indicator character is deprecated since Symfony 3.1.

This, *and* the next deprecation are in config.yml - and it even tells us the exact lines!

Open up app/config/config.yml and find line 71. Yep! Put quotes around %cache\_type%. To more closely follow the official YAML spec, if a value starts with %, it needs to have quotes around it. Do the same around the directory value.

```

80 lines | app/config/config.yml
... lines 1 - 67
68 doctrine_cache:
69   providers:
70     my_markdown_cache:
71       type: '%cache_type%'
72       file_system:
73         directory: '%kernel.cache_dir%/markdown_cache'
... lines 74 - 80

```

## Deprecation: logout\_on\_user\_change

Back on the list, there is one more easy deprecation!

Not setting `logout_on_user_change` to `true` on firewall "main" is deprecated as of 3.4.

Copy that key. Then, open `app/config/security.yml`. Under the main firewall, paste this and set it to `true`.

```

41 lines | app/config/security.yml
... lines 1 - 14
15 firewalls:
... lines 16 - 20
21   main:
... lines 22 - 29
30     logout_on_user_change: true
... lines 31 - 41

```

So, what the heck is this? Check it out: suppose you change your password while on your work computer. Previously, doing that did *not* cause you to be logged out on any *other* computers, like on your home computer. This was a security flaw, and the behavior was changed in Symfony 4. By turning this on, you can test to make sure your app doesn't have any surprises with that behavior.

Phew! Before we talk about the last deprecations, go back to the homepage and refresh. Yes! In 5 minutes our 9 deprecations are down to 2! Open up the list again. Interesting: it says:

Relying on service auto-registration for Genus is deprecated. Create a service named `AppBundle\Entity\Genus` instead.

That's weird... and involves changes to autowiring. Let's talk about those next!

# Chapter 2: Autowiring & Service Deprecations

Woo! There are only *two* deprecations left on the homepage... but they're weird! And actually, they're *not* real! These are *false* deprecation warnings!

## [Upgrade to the Symfony 3.3 services.yml Config!](#)

In our [Symfony 3.3 Tutorial](#), we talked a lot about all the new service autowiring & auto-registration stuff. We also upgraded our *old* services.yml file to use the new fancy config. It turns out that doing this is one of the *biggest* steps in upgrading to Symfony 4 and Flex. If you have *not* already upgraded your services.yml file to use autowiring & service auto-registration, stop and go through the Symfony 3.3 tutorial *now*.

## [Strict Autowiring Mode](#)

The way that autowiring *works* changed in Symfony 4. In Symfony 3, when Symfony saw a type-hint - like EntityManager - it would *first* look to see if there was a service or alias in the container with that *exact* id: so Doctrine\ORM\EntityManager. If there was *not*, it would then scan *every* service looking to see if any were an instance of this class. That magic is gone.

In Symfony 4, it's simpler: autowiring *only* does the first part: if there is not a service whose id is Doctrine\ORM\EntityManager, it throws an exception. This is a *great* change: the system is simpler and more predictable.

So, of course, if any of your autowiring depends on the old, deprecated logic, you'll see a deprecation message. And yea, that's where *these* messages are coming from!

But, there's a *better* way to find this deprecated logic. Open app/config/config.yml. Under parameters, add container.autowiring.strict\_mode: true.

```
81 lines | app/config/config.yml
... lines 1 - 7
8  parameters:
... lines 9 - 10
11 container.autowiring.strict_mode: true
... lines 12 - 81
```

This tells Symfony to use the simpler, Symfony 4-style autowiring logic right *now*. Instead of deprecations, you'll see great big, beautiful errors when you try to refresh.

So... try it! Refresh the homepage. Woh! No errors! That's because we already fixed all our old autowiring logic in the Symfony 3.3 tutorial. *And...* the 2 deprecation messages are gone! Those were *not* real issues: it's a rare situation where the deprecation system is misreporting.

## [Services are now Private](#)

So yes! This means that... drumroll... our homepage is ready for Symfony 4.0! But the rest of the site might not be. Surf around to see what other deprecations we can find. The login page looks ok: login with weaverryan+1@gmail.com, password iliketurtles. Go to /genus... still no issues and then... ah! Finally, 1 deprecation on the genus show page.

Check it out. Interesting:

The "logger" service is private, getting it from the container is deprecated since Symfony 3.2. You should either make this service public or stop using the container directly.

Wow! This is coming from GenusController line 84. Go find it! Close a few files, then open this one. Scroll down to 84. Ah!

This is *really* important. Open services.yml. These days, all of our services are *private* by default: public: false. This allows Symfony to optimize the container and all it *really* means is that we *cannot* fetch these services directly from the container. So \$container->get() will *not* work.

In Symfony 4, this is even *more* important because a lot of *previously* public services, like logger, are now *private*. Here's the

point: you need to stop fetching services directly from the container... everywhere. It's just *not* needed anymore.

## Fixing \$container->get()

What's the solution? Since we're in a controller, add a LoggerInterface \$logger argument. Then, just \$logger->info().

```
142 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 78
79 public function showAction(Genus $genus, MarkdownTransformer $markdownTransformer, LoggerInterface $logger)
80 {
... lines 81 - 84
85     $logger->info('Showing genus: '.$genus->getName());
... lines 86 - 94
95 }
... lines 96 - 140
141 }
```

Isn't that better anyways? As *soon* as we refresh the page... deprecation gone!

Where *else* are we using \$container->get()? Let's find out! In your terminal, run:

```
$ git grep "\->get("
```

Ah, just two more! We may *not* need to change all of these... some services *are* still public. But let's clean it all up!

Start in SecurityController. Ah, here it is. So: what type-hint should we use to replace this? Well, you *could* just guess! Honestly, that works a lot. Or try the brand new console command:

```
$ ./bin/console debug:autowiring
```

Sweet! This is a *full* list of *all* type-hints you can use for autowiring. Search for "authentication" and... there it is! This type-hint is an alias to the service we want.

That means, back in SecurityController, delete this line and add a new AuthenticationUtils \$authenticationUtils argument. Done.

```
43 lines | src/AppBundle/Controller/SecurityController.php
... lines 1 - 9
10 class SecurityController extends Controller
11 {
... lines 12 - 14
15 public function loginAction(AuthenticationUtils $authenticationUtils)
16 {
... lines 17 - 33
34 }
... lines 35 - 42
43 }
```

The last spot is in UserController: we're using security.authentication.guard\_handler. This time, let's guess the type-hint! Add a new argument: Guard... GuardAuthenticationHandler. That's probably it! And if we're *wrong*, Symfony will tell us. Use that value below.

```

83 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 13
14 class UserController extends Controller
15 {
... lines 16 - 18
19     public function registerAction(Request $request, LoginFormAuthenticator $authenticator, GuardAuthenticatorHandler $guardHandler)
20     {
... lines 21 - 32
33         return $guardHandler
... lines 34 - 39
40     }
... lines 41 - 44
45 }
... lines 46 - 81
82 }

```

And yep, you *can* see the GuardAuthenticationHandler class in the debug:autowiring list. But... what if it *weren't* there? What if we were trying to autowire a service that was *not* in this list?

Well... you would get a *huge* error. And *maybe* you should ask yourself: is this *not* a service I'm supposed to be using?

But anyways, if it's not in the list, there's a simple solution: go to services.yml and add your *own* alias. At the bottom, paste the class you want to use as the type-hint, then copy the service id, and say @ and paste.

```

53 lines | app/config/services.yml
... lines 1 - 5
6 services:
... lines 7 - 50
51 # example of adding aliases, if one does not exist
52 # Symfony\Component\Security\Guard\GuardAuthenticatorHandler: '@security.authentication.guard_handler'

```

Yep, that is *all* you need to do in order to define your *own* autowiring rules. Since we don't need it in this case, comment it out.

Ok, refresh! At this point, our goal is to hunt for deprecations until we're *pretty* confident they're gone: it's not an exact science. If you have a test suite, you can use the [symfony/phpunit-bridge](#) to get a report of deprecated code paths that are hit in your tests.

### [Adding \\$form->isSubmitted\(\)](#)

There is one more deprecation on the registration page. Look at the details:

Call Form::isValid() on an unsubmitted form is deprecated. Use Form::isSubmitted() before Form::isValid().

This comes from UserController. Open that class and search for isValid(). Before \$form->isValid(), add \$form->isSubmitted(). Find again and fix the other spot. This isn't very important... you just need both in Symfony 4.

```
83 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 13
14 class UserController extends Controller
15 {
... lines 16 - 18
19     public function registerAction(Request $request, LoginFormAuthenticator $authenticator, GuardAuthenticatorHandler $guardHandle
20     {
... lines 21 - 23
24         if ($form->isSubmitted() && $form->isValid()) {
... lines 25 - 44
45     }
... lines 46 - 59
60     public function editAction(User $user, Request $request)
61     {
... lines 62 - 64
65         if ($form->isSubmitted() && $form->isValid()) {
... lines 66 - 80
81     }
82 }
```

And now... I think we're done! All the deprecations I could find are *gone*.

It's time to upgrade to Symfony 4. Which, by the way, is the fastest Symfony version ever! Zoom!



## Chapter 3: Upgrade to Symfony 4.0

With the deprecations gone... yeah! It's time to upgrade to Symfony 4! If you were hoping this was going to be really cool and difficult... sorry. It's *super* easy... well... *mostly* easy.

Open `composer.json` and change `symfony/symfony` to `^4.0`. There are a few other libraries that start with `symfony/`, but they're independent and follow different release cycles. Oh, except for `symfony/phpunit-bridge`: change that to `^4.0` also.

```
68 lines | composer.json
... lines 1 - 15
16     "require": {
... line 17
18         "symfony/symfony": "^4.0",
... lines 19 - 30
31     },
32     "require-dev": {
... line 33
34         "symfony/phpunit-bridge": "^4.0",
... lines 35 - 36
37     },
... lines 38 - 68
```

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

```
$ composer update
```

Yep, upgrading is *just* that easy! Except... there are almost *definitely* some libraries in our `composer.json` file that are *not* yet compatible with Symfony 4. The best way to find out is just to try it! And then wait for an explosion!

### Removing Alice

Ah! Here is our first! Look closely... this is coming from `nelmio/alice`: the version in our project is *not* compatible with Symfony 4. If we did some digging, we would find out that there *is* a new version of Alice that *is* compatible. But, that version also contains a lot of changes to Alice... and I don't like the library's new version very much. At least, not at this moment.

So, instead of upgrading, remove `alice` from `composer.json`. This will break our fixtures: we'll fix them later.

Update again!

```
$ composer update
```

### Removing Outdated Libraries

Our next explosion! This comes from `incenteev/composer-parameter-handler`. This library helps you manage your `parameters.yml` file and... guess what? When we finish upgrading, that file will be *gone*! Yep, we do not need this library anymore.

Remove it from `composer.json`. Oh, also remove the `distribution bundle`: it helps support the *current* directory structure, and isn't needed with Flex. And below, remove the `generator bundle`. We'll install the new `MakerBundle` later.

Ok, update again!

```
$ composer update
```

## When a Library is not Ready: StofDoctrineExtensionsBundle

It works! I'm just kidding - it totally exploded again. This time the culprit is StofDoctrineExtensionsBundle: the version in our project is not compatible with Symfony

1. Now... we become detectives! Maybe the library supports it in a new version? Let's find out.

Google for StofDoctrineExtensionsBundle to find its [GitHub](#) page. Check out the [composer.json file](#). It *does* support Symfony 4! Great! Maybe there's a new version that has this! Check out the releases. Oof! No releases for a *long*, long time.

This means that Symfony 4 support *was* added, but there is not *yet* a release that contains that code. Honestly, by the time you're watching this, the bundle probably *will* have a new release. But this is likely to happen with other libraries.

Actually, another common problem is when a library does *not* have Symfony 4 support, but there is an open pull request that adds it. In both situations, we have a problem, and *you* have a choice to make.

First... you can wait. This is the most responsible decision... but the least fun. I hate waiting!

Second, if there is a pull request, you can use that *fork* as a custom composer repository and temporarily use that until the library merges the pull request and tags a release. For example, imagine this pull request was *not* merged. We could add this as a vcs repository in composer.json, and then update the version constraint to dev-master, because the branch on the fork is master.

And third, since the pull request *is* merged, but there is no tag, we can simply change our version to dev-master. Believe me: I am *not* happy about this. But I'll update it later when there *is* a release.

```
70 lines | composer.json
... lines 1 - 15
16     "require": {
... lines 17 - 27
28         "stof/doctrine-extensions-bundle": "dev-master"
29     },
... lines 30 - 70
```

Try to update again:

```
$ composer update
```

Ha! Look! It's *actually* working! Say hello to our new Symfony 4 app! Woohoo!

## Upgrading old Packages

Oh, but check out that warning: the symfony/swiftmailer-bridge is abandoned. I don't like that! Hmm, I don't see that package in our composer.json file. Run:

```
$ composer why symfony/swiftmailer-bridge
```

Ah! It's required by symfony/swiftmailer-bundle. We're using version 2.3.8, which is *apparently* compatible with Symfony 4. But I wonder if there's a newer version?

### Tip

Actually, version 2.3.8 is *not* compatible with Symfony 4. But due to an old issue with its composer.json file, it *appears* compatible. Be careful with old libraries!

Google for the package to find its [GitHub](#) page. Click releases.

Woh! There is a new version 3 of the bundle. And I bet it fixes that abandoned packages issue. Change our version to ^3.1.

```
70 lines | composer.json
... lines 1 - 15
16     "require": {
... lines 17 - 21
22         "symfony/swiftmailer-bundle": "^3.1",
... lines 23 - 28
29     },
... lines 30 - 70
```

And now, update!

```
$ composer update
```

Because we're upgrading to a new *major* version, you'll want to check out the CHANGELOG on the project to make sure there aren't any major, breaking changes.

Yes! Abandoned package warning gone! And our project is on Symfony 4. Not bad!

But... get ready... because now the *real* work starts. And the fun! It's time to migrate our project to the Flex project structure!

# Chapter 4: Installing Flex

Our project is now on Symfony 4.0, and it *still* works! Well, it *almost* works: we would just need to remove a few references to SensioDistributionBundle and SensioGeneratorBundle.

The point is this: if you want, you can upgrade to Symfony 4, but *not* migrate to the new Flex project structure. That's fine.

But... since Flex is *awesome*... let's do it!

## [Flex: Composer Plugin & New BFF](#)


Flex is a Composer *plugin*, and, it's pretty simple: when you install a package, it checks to see if there is a *recipe* for that package. A recipe can add configuration files, auto-enable the bundle, add paths to your .gitignore file and more. But, for Flex to work, you need to use the Flex directory structure.

## [Upgrade to Flex: The Plan](#)

So here's the plan: we're going to bootstrap a new Flex application *right* inside our *existing* project. Then, little-by-little, we'll move *our* code and configuration *into* it. It's going to be pretty freakin' cool.

## [Upgrade Composer. For Real](#)

Before we start, make sure that your Composer is at the *latest* version:




```
$ composer self-update
```

Seriously, *do* this. Composer recently released a bug fix that helps Flex.

## [Installing Flex](#)

Ok, so... let's install Flex!




```
$ composer require symfony/flex
```

As *soon* as this is in our project, it will find and install recipes each time we add a new library to our project. In fact, check it out!

Configuring symfony/flex

Ha! Flex even installed a recipe for *itself*! What an over-achiever! Let's find out what it did:



```
$ git status
```

Of course, it modified composer.json and composer.lock. But there are two *new* files: .env.dist and symfony.lock. Open the first.

How did this get here? It was added by the symfony/flex recipe! More about this file later.

Next, look at symfony.lock. This file is managed by Flex: it keeps track of which recipes were installed. You should commit it, but not think about it.

## [Installing Missing Recipes](#)

Because this is an existing project, our app already contains a bunch of vendor libraries... and a lot of these *might* have

recipes that were never installed because Flex wasn't in our project yet! Lame! No problem! Empty the vendor/ directory and run composer install

```
$ rm -rf vendor  
$ composer install
```

Normally, Flex only installs a recipe when you *first* composer require a library. But Flex knows that the recipes for these libraries were *never* installed. So it runs them now.

Yea! 11 recipes! Woh! And one of them is from the "contrib" repository. There are two repositories for recipes. The official one is heavily guarded for quality. The "contrib" one also has some checks, but the quality is not guaranteed. That's why you see this question. I'll type "p" to permanently allow recipes from contrib.

Run git status to see what changed:

```
$ git status
```

Woh! We have a new config/ directory and a lot more! Starting with nothing, Flex is scaffolding the new project around us! It's even auto-enabling all the bundles in a new bundles.php file.

Sweet!

### [The Flex composer.json](#)

When you start a *new* Flex project, you actually clone this [symfony/skeleton](#) repository... which is literally one file: composer.json. This has a few *really* important things in it, including the fact that it requires symfony/framework-bundle but *not* symfony/symfony.

Let's work on that next!

# Chapter 5: The Flex composer.json File

We need to make *our* composer.json file look like the one from symfony/skeleton. Actually, go to "Releases", find the latest release, and then click to browse the files. Now we can see the *stable* composer.json contents.

So... yea, this *one* file is all you need to start a *new* project. That's crazy! Flex *builds* the project structure around it.

## Tip

Before copying the composer.json, make sure to change the branch on GitHub to the latest released version you want to upgrade to (e.g. 4.1)

## [Bye Bye symfony/symfony](#)

Anyways, the *most* important change is that, with Flex, you *stop* requiring symfony/symfony. Yep, you require *only* the specific packages that you need. Copy all of the require lines, find our composer.json file, and paste over the php and symfony/symfony lines. Oh, and remove symfony/flex from the bottom: it's up here now.

```
74 lines | composer.json
1  {
  ... lines 2 - 17
18  "require": {
19    "php": "^7.1.3",
20    "symfony/console": "^4.0",
21    "symfony/flex": "^1.0",
22    "symfony/framework-bundle": "^4.0",
23    "symfony/lts": "^4@dev",
24    "symfony/yaml": "^4.0",
  ... lines 25 - 34
35  },
  ... lines 36 - 72
73 }
```

The symfony/framework-bundle package is the most important: this is the *core* of Symfony: it's *really* the only required package for a Symfony app.

Go back and also copy the dotenv package from require-dev and put it in our composer.json file. This package is responsible for reading the new .env file.

```
74 lines | composer.json
1  {
  ... lines 2 - 35
36  "require-dev": {
37    "symfony/dotenv": "^4.0",
  ... lines 38 - 39
40  },
  ... lines 41 - 72
73 }
```

## [Synchronizing the rest of Composer.json](#)

Go back and also copy the config line and paste that here too.

```

74 lines | composer.json
1  {
  ... lines 2 - 40
41  "config": {
42    "preferred-install": {
43      "**": "dist"
44    },
45    "sort-packages": true
46  },
  ... lines 47 - 72
73 }

```

Skip the autoload sections for now, but copy the rest of the file. Replace the existing scripts and extras sections with this new, shiny stuff.

```

74 lines | composer.json
1  {
  ... lines 2 - 46
47  "scripts": {
48    "auto-scripts": {
49      "cache:clear": "symfony-cmd",
50      "assets:install --symlink --relative %PUBLIC_DIR%": "symfony-cmd"
51    },
52    "post-install-cmd": [
53      "@auto-scripts"
54    ],
55    "post-update-cmd": [
56      "@auto-scripts"
57    ]
58  },
59  "conflict": {
60    "symfony/symfony": "*"
61  },
62  "extra": {
63    "symfony": {
64      "allow-contrib": true
65    }
66  },
  ... lines 67 - 72
73 }

```

Brilliant!

## [Autoloading src/ & src/AppBundle](#)

Let's talk about autoload. In Symfony 3, everything lived in `src/AppBundle` and had an `AppBundle` namespace. But in Symfony 4, as you can see, everything should live *directly* in `src/`. And even though we don't have any examples yet, the namespace will start with `App\`, even though there's no `App/` directory.

Eventually, we *are* going to move all of our files into `src/` and refactor all of the namespaces. With PhpStorm, that won't be as scary as you think. But, it *is* a big change, and you may *not* be able to do that all at once in a real project.

So, I'm going to show you a more "gentle", gradual way to upgrade to Flex. Yep, for now, we're going to leave our files in `AppBundle` and make them work. But *new* files will live directly in `src/`.

Right now, the `autoload` key in `composer.json` says to look for *all* namespaces in `src/`. Make this more specific: the `AppBundle\` namespace prefix should live in `src/AppBundle`. Do the same in `autoload-dev`: `Tests\AppBundle\` will live in

tests/AppBundle.

```
74 lines | composer.json
1  {
    ... lines 2 - 4
5    "autoload": {
6      "psr-4": {
7        "AppBundle\\": "src/AppBundle",
    ... line 8
9      },
    ... line 10
11   },
12   "autoload-dev": {
13     "psr-4": {
14       "Tests\\AppBundle\\": "tests/AppBundle",
    ... line 15
16     }
17   },
    ... lines 18 - 72
73 }
```

Why are we doing this? Because *now* we can go copy the autoload entry from the official composer.json file and add it below our AppBundle\\ line. Copy the new autoload-dev line also.

```
74 lines | composer.json
1  {
    ... lines 2 - 4
5    "autoload": {
6      "psr-4": {
    ... line 7
8        "App\\": "src/"
9      },
    ... line 10
11   },
12   "autoload-dev": {
13     "psr-4": {
    ... line 14
15       "App\\Tests\\": "tests/"
16     }
17   },
    ... lines 18 - 72
73 }
```

Thanks to this, Composer can autoload our *old* classes *and* any new classes!

## Scaffold the Full Structure

Ok, that was a *huge* step. Run composer update:

```
$ composer update
```

The *biggest* change is that we're not relying on symfony/symfony anymore. And, yep! You can see it remove symfony/symfony and start adding individual libraries.

Ah, it explodes! Don't worry about that yet: Composer *did* finish and Flex configured 3 new recipes!



At this point, the new Flex project is *fully* built... and it already works! I'll prove it to you next.

# Chapter 6: Your Flex Project is Alive!

Thanks to the Flex recipe for symfony/framework-bundle, we now have a fully-functional Symfony Flex app living *right* inside our directory! `public.` is the new document root, `config/` has all of the configuration, and our PHP code lives in `src/`, including the new Kernel class.

Yep, we have our *old* app with all our stuff, and a *new*, Flex, app, which is basically empty and *waiting* for us to move our code into it.

## Re-Order .env

Open up `.env.dist`. Woh! This has more stuff now! That's thanks to the recipes from DoctrineBundle, SwiftmailerBundle and FrameworkBundle. Copy the FrameworkBundle section and move that to the top. Do the same thing to `.env`.

```
25 lines | .env.dist
1  # This file is a "template" of which env vars need to be defined for your application
2  # Copy this file to .env file for development, create environment variables when deploying to production
3  # https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
4
5  ###> symfony/framework-bundle ###
6  APP_ENV=dev
7  APP_SECRET=12c008ecf65c043dc2b14b5eb9a115ef
8  #TRUSTED_PROXIES=127.0.0.1,127.0.0.2
9  #TRUSTED_HOSTS=localhost,example.com
10 ###
... lines 11 - 25
```

We don't need to do this, but `APP_ENV` is so important, I want to see it first. If you start a *new* Flex app, it's on top.

## Re-Ordering the Libs

Next, this will sound weird, but run:

```
$ composer require symfony/flex
```

We *already* have this library. I know. So... why are we doing this? It's a little trick: one of the *new* keys in our `composer.json` is `sort-packages`, which is set to `true`. Thanks to this, whenever you run `composer require`, it orders the packages alphabetically. By requiring a package we already have, Composer just re-ordered my packages.

Thanks Jordi!

## Fixing the console

But... we still have this *giant* error: attempted to load SecurityBundle from AppKernel. Bummer! This happens because `bin/console` is still trying to boot the *old* app.

When you start a *new* Flex project, the `symfony/console` recipe creates the `bin/console` file. But, since our project already *had* this file, the recipe couldn't do its job.

No worries! Let's go find the new file! Go to [github.com/symfony/recipes](https://github.com/symfony/recipes). Welcome to the official recipes repository!

Navigate to `symfony`, `console`, then `bin`. There it is! Copy its contents. Then, completely replace our version.

40 lines | [bin/console](#)

```
1  #!/usr/bin/env php
2
3
4  use App\Kernel;
5  use Symfony\Bundle\FrameworkBundle\Console\Application;
6  use Symfony\Component\Console\Input\ArgvInput;
7  use Symfony\Component\Debug\Debug;
8  use Symfony\Component\Dotenv\Dotenv;
9
10 set_time_limit(0);
11
12 require __DIR__.'../vendor/autoload.php';
13
14 if (!class_exists(Application::class)) {
15     throw new \RuntimeException("You need to add \"symfony/framework-bundle\" as a Composer dependency.");
16 }
... lines 17 - 40
```

This will boot the *new* application! So... does it work? Run:



```
$ ./bin/console
```

No! But that's a new error: we are closer! This says that the autoloader expects `App\AppBundle\AppBundle` to be defined in `AppBundle.php`, but it wasn't found. That's strange... that is *not* the correct namespace for that class! If you look closer, it says the error is coming from a new `config/services.yaml` file.

Our old code - the stuff in `src/AppBundle` - should not be used at *all* by the new app yet. Open that new `config/services.yaml` file. It has the same auto-registration code that we're familiar with. And, ah ha! Here's the problem: it is auto-registering *everything* in `src/` as a service, but it's telling Symfony that the namespace of each class will start with `App\`. But, our stuff starts with `AppBundle`!

For now, completely ignore `AppBundle`: let's get the new project working and *then* migrate our code.

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

```
... lines 1 - 4
5  services:
... lines 6 - 15
16  App\:
... line 17
18      exclude: '../src/{Entity,Migrations,Tests,AppBundle}'
... lines 19 - 28
```

Ok, try `bin/console` again:



```
$ bin/console
```

It's alive! We just hacked a fully-functional Flex app *into* our project! Now let's move our code!

# Chapter 7: Migrating framework Config

Most of our old code lives in `app/config`, and also `src/AppBundle`. We'll talk about that directory later - it's easier.

Yep, *most* of the work of migrating our code to the new app involves moving each piece of config into the new location. Honestly, it's tedious and slow. But you're going to learn a lot, and the end result is totally worth it.

## Moving Parameters

Start in `config.yml`. Ignore imports: we'll look at each file one-by-one. The first key is parameters. Copy those, delete them, and open `config/services.yml`. This is where your parameters and services will live. Paste them here. Oh, but remove the `strict_mode` line: autowiring *always* works in "strict mode" in Symfony 4.

```
30 lines | config/services.yml
... lines 1 - 2
3  parameters:
4    locale: en
5    cache_type: file_system
... lines 6 - 30
```

## Environment Variables

Keep going! Back to `config.yml`. The keys under `framework` will be the *most* work to migrate... by *far*. In Flex, this configuration will live in `config/packages/framework.yml`: each package has its *own* config file.

Remove the `esi` line from the old file: it's *also* commented out in the *new* one: nothing to migrate.

Check out the secret config: it's set to `%env(APP_SECRET)%`. That's a relatively new syntax that reads from *environment* variables. In the dev environment, Symfony loads the `.env` file, which sets all these keys as environment variables, including `APP_SECRET`.

Delete the old secret key. The *way* this value is set is a bit different, but, the point is, it's handled.

## Requiring translator

The next key is `translator`. Are you ready? Because *this* is where things get fun! You *might* think that all we need to do is copy this line into `framework.yml`. But no!

Many of the keys under `framework` represent *components*. In Symfony 3, by adding the `translator` key, you *activated* that component.

But with Flex, the `Translator` component isn't even *installed* yet. Yep, if you want a translator, you need to install it. In your terminal, run:

```
$ composer require translator
```

If that package name looks funny... it should! There is *no* package called `translator`! But look! It added a new `symfony/translation` key to `composer.json`.

This is another superpower of Flex. Go to <https://symfony.sh/>. This is a list of *all* of the packages that have a recipe. Search for "translation" to find `symfony/translation`. See those Aliases? Yep, we can reference `translation`, `translations` or `translator` and Flex will, um, translate that into `symfony/translation` automatically.

## The translator Recipe

Back to the terminal! Before I started recording, I committed all of our changes so far. That was no accident: Flex just installed a recipe and I want to see *exactly* what it did! Run:

```
$ git status
```

Cool! It created a new translation.yaml file and a translation/ *directory*.

```
8 lines | config/packages/translation.yaml
```

```
1 framework:
2   default_locale: '%locale%'
3   translator:
4     paths:
5       - '%kernel.project_dir%/translations'
6     fallbacks:
7       - '%locale%'
```

That is where translation files should live in Symfony 4. And even though the translator config lives under framework, in Flex, it has its *own* configuration file. Oh, and this is one of my *favorite* things about Flex. Why should my translation files live in a translations/ directory? Is that hardcoded somewhere deep in core? Nope: it's right here in *your* configuration file. Want to put them somewhere else? Just update that line or add a second path.

So, do we need to move the translator config from our old project? Actually, no! It's already in the new file. Delete it.

And since we now know that translations should live in this new translations/ directory, let's move our existing files... well file. In app/Resources/translations, move validators.en.yml down into translations/.

## [Migrating router Config](#)

We're on a roll! What about the router config? It told Symfony to load routing.yml. All of that is taken care of in the new app: it loads a routes.yaml file and anything in the routes/ directory, like annotations.yaml.

There's also a config/packages/routing.yaml file, and even another one in dev/ to tweak that strict\_requirements setting.

The point is this: routing is handled. Delete that stuff!

## [Migrating Forms and Validator](#)

Next, forms! Like with translations, this activates a component that is not installed yet. We *do* have forms in our app, so we need this and validation. Let's get them installed:

```
$ composer require form validator
```

Yep! More aliases! Perfect! This time, it did not install *any* recipes. That's cool: not all packages need a recipe.

So, do we need to move these 2 lines of config into framework.yaml? Actually, no!

Go back to your terminal and run:

```
$ ./bin/console debug:config framework
```

This prints out the current framework configuration. Search for form. Nice! It's *already* enabled, even without *any* config! This is really common with Flex: as soon as a component is installed, FrameworkBundle automatically enables it. No configuration is needed unless you want to change something. Delete the form line.

Search for "validation" next: it's even *more* interesting! It's *also* enabled, and enable\_annotations is set to true. Great! Delete the validation line! What's *really* interesting is that enable\_annotations is set to true because it detected that we have the Doctrine annotations package installed. This is the flow with Flex: install a package and you're done.

Ok! It might not look like it, but we're almost done with the framework stuff. Let's finish it next!

## Chapter 8: Finishing framework Config

Let's finish this! Back on `debug:config`, search for `default_locale`. Apparently, it's already set to `en`. Cool! Let's remove that from `config.yml`. You *can* move it if you want: the translator *did* add a locale parameter. I'll delete it.

### Migrating `csrf_protection`

Close a few files: I want to keep comparing `config.yml` and `framework.yml`. In `config.yml`, we have `csrf_protection` activated. Ok, so uncomment it in `framework.yml`. Then, remove it from `config.yml`. Let's also remove `serializer`: I wasn't using it before. If you *are*, run `composer require serializer` to activate it. No config is needed.

```
16 lines | config/packages/framework.yml
1  framework:
    ... lines 2 - 3
4  csrf_protection: ~
    ... lines 5 - 16
```

Ok, let's see if we broke anything! Run:

```
$ ./bin/console
```

Woh! We're busted!

CSRF support cannot be enabled as the Security CSRF component is not installed.

Ohhhh. Like translation and form, `csrf_protection` activates a component that we don't have installed! No problem! Go back to `symfony.sh` and search for "csrf". There it is! Run:

```
$ composer require security-csrf
```

By the way, once this is installed, the `csrf_protection` key in `framework.yml` should *not* be needed... well, starting in Symfony 4.0.2... there was a small bug. Since I'm using 4.0.1, I'll keep it.

Let's go check on Composer. It downloads the package and... explodes!

CSRF protection needs sessions to be enabled

### Enabling Sessions

Ah, sessions. They are *off* by default. Uncomment this config to activate them. Sessions are a bit weird, because, unlike translator or `csrf_protection`, you can't activate them simply by requiring a package. You need to *manually* change this config. It's no big deal, but it's the *one* part of framework config that isn't super smooth.

```
16 lines | config/packages/framework.yml
1  framework:
    ... lines 2 - 7
8  session:
9      # With this config, PHP's native session handling is used
10     handler_id: ~
    ... lines 11 - 16
```

Oh, and notice that this config *is* a bit different than before. In Symfony 3, we stored sessions in `var/sessions`. And you can still *totally* do this. But the *new* default tells PHP to store it on the filesystem wherever it wants. It's just one less thing to worry

about: PHP will handle all the file permissions.

### Tip

Just remember: when you change your session storage location, your users will lose their current session data when you first deploy!

Remove the old session configuration. Let's see if the app works!

```
$ ./bin/console
```

Yes!

## Migrate Twig

We're getting close! Next is templating. This component still exists, but isn't recommended anymore. Instead, you should use twig directly. So, delete it.

### Tip

If your app references the templating service, you'll need to change that to twig.

But our app *does* use Twig. So find your terminal. Oh, let's commit first: I want to see what the Twig recipe does. Create a calm and sophisticated commit message. Now run:

```
$ composer require twig
```

Yay aliases! This added TwigBundle, and Flex installed its Recipe. Run:

```
$ git status
```

Ah, this made some cool changes! First, in config/bundles.php, it automatically enabled the bundle. Flex does this for *every* bundle. I love that!

It also added a config/packages/twig.yaml file. Where do templates live in a Flex app? You can see it right here! In templates/ at the root of our project. And hey! It even *created* that directory for us with base.html.twig inside.

The config in twig.yaml is *almost* the same as our old app. Copy the extra form\_themes and number\_format keys, delete the old config, and paste them at the bottom of twig.yaml.

10 lines | [config/packages/twig.yaml](#)

```
1 twig:
2   paths: ['%kernel.project_dir%/templates']
3   debug: '%kernel.debug%'
4   strict_variables: '%kernel.debug%'
5   number_format:
6     thousands_separator: ','
7   form_themes:
8     - bootstrap_3_layout.html.twig
9     - _formTheme.html.twig
```

Oh, and the recipe gave us something else for free! Any routes in config/routes/dev are automatically loaded, but only in the dev environment. The recipe added a twig.yaml file there with a route import. This helps you debug and design your error pages. All of this stuff is handled automatically.

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

Now that we know that template files should live in templates/, let's move them there! Open app/Resources/views. Copy *all* of the files and paste them. And yes, we *do* want to override the default base.html.twig.

Perfect! Now, celebrate: *completely* remove app/Resources/views. Actually, woh! We can delete *all* of Resources/! Our app/ directory is getting *really* small!

### [Migrating trusted\\_hosts, fragments & http\\_method\\_override](#)

We're now down to the *final* parts of framework. So what about trusted\_hosts, fragments and http\_method\_override? Remove all of those. And in framework.yaml, uncomment fragments.

```
16 lines | config/packages/framework.yaml
1  framework:
    ... lines 2 - 12
13  fragments: ~
    ... lines 14 - 16
```

If you run:

```
$ bin/console debug:config framework
```

you'll see that the other keys already default to the old values. Yep, http\_method\_override is still true and trusted\_hosts is already empty.

### [Migrating assets](#)

This leaves us with *one* last key: assets. And guess what? This enables a component. And right now, in debug:config, you can see that assets is enabled: false.

Install it:

```
$ composer require asset
```

It installs the component, but this time, there is no recipe. But run debug:config again:

```
$ ./bin/console debug:config framework
```

Search for "asset". Ha, yes! It enabled itself.

Ok: delete the framework key. This is *huge*! I know I know, it *feels* like we still have a lot of work to do. But that's not true! With framework out of the way, we are in the home stretch!



# Chapter 9: Migrating Services & Security

Ok, remember our goal: to move our code - which mostly lives in `config/` - into the *new* directory structure.

## Migrating the doctrine Config

The next section is doctrine... and there's nothing special here: this is the default config from Symfony 3. Compare this with `config/packages/doctrine.yaml`. If you look closely, they're almost the same - but with a few improvements!

Instead of having multiple config entries for the database host, username and password, it's all combined into one url. The `DATABASE_URL` environment variable is waiting to be configured in the `.env` file.

But there is *one* important difference: mappings. In a Flex project, we expect your entities to live in `src/Entity`. But currently, *our* classes live in `src/AppBundle/Entity`.

And yes yes, we *are* going to move them... eventually. But let's pretend like moving them is too big of a change right now: I want to make my files work where they are. How can we do that? Add a second mapping! This one will look in the `src/AppBundle/Entity` directory for classes that start with `AppBundle\`. Update the alias to `AppBundle` - that's what lets you say `AppBundle:Genus`.

```
34 lines | config/packages/doctrine.yaml
... lines 1 - 7
8  doctrine:
... lines 9 - 16
17  orm:
... lines 18 - 20
21  mappings:
... lines 22 - 27
28      AppBundle:
29          is_bundle: false
30          type: annotation
31          dir: '%kernel.project_dir%/src/AppBundle/Entity'
32          prefix: 'AppBundle\Entity'
33          alias: AppBundle
```

Simple & explicit. I love it! Go delete the old doctrine config!

## Migrating doctrine\_cache and stof\_doctrine\_extensions

The last two sections are for `doctrine_cache` and `stof_doctrine_extensions`. Both bundles are installed, so we just need to move the config. Huh, but the `DoctrineCacheBundle` did *not* create a config file. That's normal: some bundles don't *need* configuration, so their recipes don't add a file. Create it manually: `doctrine_cache.yaml`. And move all the config into it.

```
7 lines | config/packages/doctrine_cache.yaml
1  doctrine_cache:
2    providers:
3      my_markdown_cache:
4        type: '%cache_type%'
5        file_system:
6          directory: '%kernel.cache_dir%/markdown_cache'
```

All of the files in this directory are automatically loaded, so we don't need to do anything else.

Then, for `stof_doctrine_extensions`, it *does* have a config file, but we need to paste our custom config at the bottom.

```

8 lines | config/packages/stof_doctrine_extensions.yaml
... lines 1 - 2
3  stof_doctrine_extensions:
  ... line 4
5    orm:
6      default:
7        sluggable: true

```

And... that's it! Delete config.yml. Victory!

## Migrating Services

Close a few files, but keep the new services.yaml open... because this is our next target! Open the *old* services.yml file. This has the normal autowiring and auto-registration stuff, as well as some aliases and custom service wiring.

Because we're *not* going to move our classes out of AppBundle yet, we need to *continue* to register those classes as services. But in the new file, to get things working, we *explicitly* excluded the AppBundle directory, because those classes do not have the App\ namespace.

No problem! Copy the 2 auto-registration sections from services.yml and paste them into the new file. And I'll add a comment: when we eventually move everything *out* of AppBundle, we can delete this. Change the paths: we're now one level *less* deep.

```

68 lines | config/services.yaml
... lines 1 - 6
7  services:
  ... lines 8 - 27
28  # REMOVE when AppBundle is removed
29  AppBundle\:
30    resource: '../src/AppBundle/**'
31    # you can exclude directories or files
32    # but if a service is unused, it's removed anyway
33    exclude: '../src/AppBundle/{Entity,Repository}'
34  AppBundle\Controller\:
35    resource: '../src/AppBundle/Controller'
36    public: true
37    tags: ['controller.service_arguments']
38  # END REMOVE
  ... lines 39 - 68

```

Next, copy the existing aliases and services and paste them into the new file.

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

... lines 1 - 6


7 services:

... lines 8 - 39

```
40 # add more service definitions when explicit configuration is needed
41 # please note that last definitions always *replace* previous ones
42
43 Knp\Bundle\MarkdownBundle\MarkdownParserInterface: '@markdown.parser'
44 Doctrine\ORM\EntityManager: '@doctrine.orm.default_entity_manager'
45
46 AppBundle\Service\MarkdownTransformer:
47   arguments:
48     $cacheDriver: '@doctrine_cache.providers.my_markdown_cache'
49
50 AppBundle\Doctrine\HashPasswordListener:
51   tags: [doctrine.event_subscriber]
52
53 AppBundle\Form\TypeExtension\HelpFormExtension:
54   tags:
55     - { name: form.type_extension, extended_type: Symfony\Component\Form\Extension\Core\Type\FormType }
56
57 AppBundle\Service\MessageManager:
58   arguments:
59     - ['You can do it!', 'Dude, sweet!', 'Woot!']
60     - ['We are *never* going to figure this out', 'Why even try again?', 'Facepalm']
61
62 AppBundle\EventSubscriber\AddNiceHeaderEventSubscriber:
63   arguments:
64     $showDiscouragingMessage: true
65
66 # example of adding aliases, if one does not exist
67 # Symfony\Component\Security\Guard\GuardAuthenticatorHandler: '@security.authentication.guard_handler'
```

And... ready? Delete services.yml! That was a *big* step! Suddenly, almost *all* of our existing code is being used: we just hooked our old code into the new app.


But, does it work! Maybe....? Try it!



```
$ ./bin/console
```

## Migrating Security

Ah! Not *quite*: a class not found error from Symfony's Guard security component. Why? Because we haven't installed security yet! Let's do it:



```
$ composer require security
```

It downloads and then... another error! Interesting:

LoginFormAuthenticator contains 1 abstract method

Ah! I think we *missed* a deprecation warning, and now we're seeing a fatal error. Open AppBundle/Security/LoginFormAuthenticator.php.

PhpStorm agrees: class must implement method onAuthenticationSuccess. Let's walk through this change together. First, remove getDefaultSuccessRedirectUrl(): that's not used anymore. Then, go to the Code->Generate menu - or Command+N

on a Mac - select "Implement methods" and choose onAuthenticationSuccess.

Previously, this method was handled by the base class for you. But now, it's your responsibility. No worries: it's pretty simple. To help, at the top, use a trait called TargetPathTrait.

```
89 lines | src/AppBundle/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
22     use TargetPathTrait;
... lines 23 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
... lines 77 - 81
82     }
... lines 83 - 87
88 }
```

Back down in onAuthenticationSuccess, this allows us to say if \$targetPath = \$this->getTargetPath() with \$request->getSession() and main.

```
89 lines | src/AppBundle/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), 'main')) {
... line 78
79         }
80
... line 81
82     }
... lines 83 - 87
88 }
```

Let's break this down. First, the main string is just the name of our firewall. In both the old *and* new security config, that's its key.

Second, what does getTargetPath() do? Well, suppose the user originally tried to go to /admin, and then they were redirected to the login page. After they login, we should probably send them back to /admin, right? The getTargetPath() method returns the URL that the user *originally* tried to access, if any.

So if there *is* a target path, return new RedirectResponse(\$targetPath). Else, return new RedirectResponse and generate a URL to the homepage.

```

89 lines | src/AppBundle/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), 'main')) {
78             return new RedirectResponse($targetPath);
79         }
80
81         return new RedirectResponse($this->router->generate('homepage'));
82     }
... lines 83 - 87
88 }

```

PhpStorm thinks this isn't a real route, but it is!

Problem solved! Is that enough to make our app happy? Find out!



It is! But before we move on, we need to migrate the security config. Copy *all* of the old security.yml, and *completely* replace the new security.yml. To celebrate, delete the old file!

```

41 lines | config/packages/security.yml
... lines 1 - 2
3 security:
4     encoders:
5         AppBundle\Entity\User: bcrypt
6
7     role_hierarchy:
8         ROLE_ADMIN: [ROLE_MANAGE_GENUS, ROLE_ALLOWED_TO_SWITCH]
... lines 9 - 14
15     firewalls:
... lines 16 - 20
21     main:
22         anonymous: ~
23         guard:
24             authenticators:
25                 - AppBundle\Security\LoginFormAuthenticator
26
27         logout:
28             path: /logout
29             switch_user: ~
30             logout_on_user_change: true
... lines 31 - 38
39     access_control:
40         # - { path: ^/admin, roles: ROLE_ADMIN }

```

And... ah! We're *super* close. Only a *few* more files to deal with! By the end of the next chapter, our app/config/ directory will be gone!

# Chapter 10: Final config/ Migration

We are in the Flex home stretch! These last config/ files are the easiest. Start with config\_dev.yml.

## Dev Environment Parameters

Ok, we have a cache\_type parameter. This is meant to override the value that lives in services.yml whenever we're in the dev environment.

How can we have dev-specific parameters or services in Flex? By creating a new services\_dev.yml file. Copy the parameter, remove it and paste it here.

```
3 lines | config/services_dev.yml
1  parameters:
2    cache_type: array
```

Symfony will automatically load this file in the dev environment only.

## Migrating config\_dev.yml

For the rest of this file... we haven't really changed anything: these are the original default values. So there's a good chance that we can just use the *new* files without doing anything.

And yea! If you investigated, you would find that the framework config is already represented in the new files. And the profiler... well actually... that's not even installed yet. Let's fix that:

```
$ composer require profiler
```

Go back and remove the framework and web\_profiler sections. When Composer finishes... yes! This installed a recipe. The new web\_profiler.yml file contains exactly what we just removed. It even added config for the test environment *and* loaded the routes it needs. Thanks profiler!

```
7 lines | config/packages/dev/web_profiler.yml
1  web_profiler:
2    toolbar: true
3    intercept_redirects: false
4
5  framework:
6    profiler: { only_exceptions: false }
```

```
7 lines | config/packages/test/web_profiler.yml
1  web_profiler:
2    toolbar: false
3    intercept_redirects: false
4
5  framework:
6    profiler: { collect: false }
```

```

8 lines | config/routes/dev/web_profiler.yml
1  web_profiler_wdt:
2    resource: '@WebProfilerBundle/Resources/config/routing/wdt.xml'
3    prefix: /_wdt
4
5  web_profiler_profiler:
6    resource: '@WebProfilerBundle/Resources/config/routing/profiler.xml'
7    prefix: /_profiler

```

The last key in config\_dev.yml is monolog. Monolog *is* installed... and its recipe added config for the dev and prod environments.

I haven't made any changes to my monolog config that I really care about - just this firephp section, which I could re-add if I want. So I'll use the new default config and just... delete config\_dev.yml! We can also delete config\_prod.yml.

### [doctrine Config in config\\_prod.yml](#)

Oh, by the way, if you have some doctrine caching config in config\_prod.yml, I would recommend *not* migrating it. The DoctrineBundle recipe gives you prod configuration that is *great* for production out-of-the-box. Booya!

### [Migrating config\\_test.yml](#)

Next: config\_test.yml. And yea... this is *still* just default config. But there *is* one gotcha: in config/packages/test/framework.yml, uncomment the session config.

```

5 lines | config/packages/test/framework.yml
1  framework:
... line 2
3  session:
4    storage_id: session.storage.mock_file

```

I mentioned earlier that the session config is not perfect smooth: if you need sessions, you need to uncomment some config in the main framework.yml and here too.

Ok, delete config\_test.yml!

### [Migrating paramters.yml?](#)

What about parameters.yml? In Flex, this file does *not* exist. Instead of referencing parameters, we reference *environment variables*. And in the dev environment, we set these in the .env file.

We also had a parameters.yml.dist file, which kept track of all the parameters we need. In Flex, yea, we've got the same: .env.dist.

The parameters.yml file in this project only holds database config and secret... and both of these are already inside .env and .env.dist.

The only difference between the files is how you *reference* the config. In doctrine.yml, instead of using %DATABASE\_URL% to reference a paramter, you reference environment variables with a strange config: %env(DATABASE\_URL)%.

But other than that, it's the same idea. Oh, the resolve: part is optional: it allows you to put parameters *inside* of your environment variable values.

So... we're good! Delete parameters.yml and parameters.yml.dist. If *you* have other keys in parameters.yml, add them to .env and .env.dist and then go update where they're referenced to use the new syntax. Easy peasy.

While we're on the topic, in .env, update your database config: I'll use root with no password and call the database symfony4\_tutorial.

Copy that and repeat it in .env.dist: I want this to be my default value.

25 lines | [.env.dist](#)

... lines 1 - 15

```
16 DATABASE_URL=mysql://root:@127.0.0.1:3306/symfony4_tutorial
```

... lines 17 - 25

## [Migrating routing Files](#)

Back to the mission! What about routing.yml? Copy its contents. I'll close a few directories... then open config/routes.yml. Paste here!

9 lines | [config/routes.yml](#)

```
1 app:
2   resource: "@AppBundle/Controller/"
3   type:    annotation
4
5 homepage:
6   path: /
7   defaults:
8     _controller: AppBundle:Main:homepage
```

We *already* have a config/routes/dev/annotations.yml file that loads annotation routes from src/Controller. But for now, we still need *our* import because it loads routes from AppBundle.

But we *do* need to make two small changes. *Even* though we'll keep the AppBundle directory for now, we are *not* going to actually *register* it as a bundle anymore. Yep, AppBundle.php can be deleted: we just *don't* need bundles anymore.

But to make this work, we need to replace @AppBundle with a normal path: ../src/AppBundle/Controller.

And for the homepage route, remove the weird three-part colon syntax and just use the full class name: AppBundle\Controller\MainController::homepageAction.

9 lines | [config/routes.yml](#)

```
1 app:
2   resource: "../src/AppBundle/Controller/"
3   ... lines 3 - 4
5 homepage:
6   ... line 6
7   defaults:
8     _controller: AppBundle\Controller\MainController::homepageAction
```

I am so happy to be done with those two Symfony-specific syntaxes! Delete routing.yml. And... routing\_dev.yml? Yep, delete it too! The Flex recipes handle this stuff too.

In fact, delete the config/ directory!

Does our app work? Try to list the routes:

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

Ha! Yes! We have our routes back!

Next, let's *delete* some files - that's always fun - and then welcome our new Flex app!




# Chapter 11: Hello Flex: Moving Final Files

We're on a mission to remove the *last* parts of our Symfony 3 structure!

## [Moving DoctrineMigrations](#)

So what about the DoctrineMigrations directory? Look in src/. Interesting... the DoctrineMigrationsBundle recipe added a Migrations/ directory. So, I guess that's where they go!

Copy all of the migration files and paste them there. I guess that worked? Let's find out:

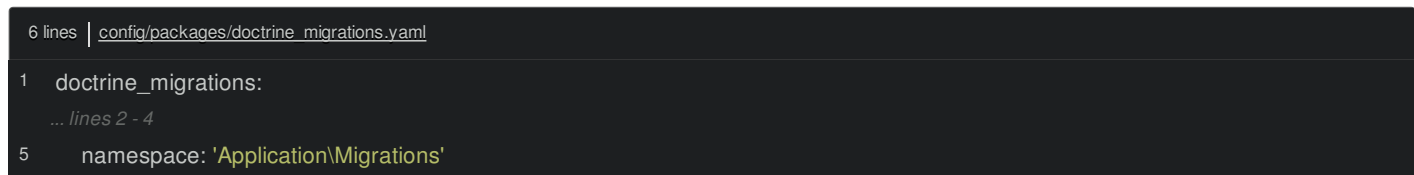


```
$ ./bin/console doctrine:migrations:status
```

Ah! I guess not! It says that my migration class wasn't found: Is it placed in a DoctrineMigrations namespace? Bah! I don't know!


Our files have an Application\Migrations namespace. What's going on? Open the config/packages/doctrine\_migrations.yaml file.

Ah ha! The recipe installed config that told the bundle to expect a DoctrineMigrations namespace. Easy fix! Copy the current namespace, and paste it here.



```
6 lines | config/packages/doctrine_migrations.yaml
1  doctrine_migrations:
  ... lines 2 - 4
5  namespace: 'Application\Migrations'
```

Try the command again:



```
$ ./bin/console doctrine:migrations:status
```

Life is good! Well, we don't have a database - but life is still *pretty* good.

## [Removing the app/ Directory](#)

At this point, app/ only has 3 files left: AppKernel, AppCache and autoload.php. And unless you made some crazy customizations to these, you don't need any of them. Yes, I'm telling you to delete the app/ directory!

And in composer.json, remove the classmap line: those files are gone!

## [Moving & Delete Files](#)

Let's reset our files and look at each directory one by one. We need bin/, config/ and public/ is the new document root. src/ holds our code, and templates/, tests/ and translations/ are all valid Flex directories. Oh, and tutorial/? Ignore that: I added that for this course - it has a file we'll use later.

But expand var/. Delete everything except for cache and log: the default logs directory was renamed in Flex. And that bootstrap file is a relic of the past!

And finally.... web! This directory should *not* exist. Select the files we need: css/, images/, js/ and vendor/: move these into public/. Let's also move robots.txt.

And that's it! The favicon is from Symfony and we don't need the app files anymore. What about .htaccess? You only need that if you use Apache. And if you *do*, Flex can add this file for you! Just run `composer require symfony/apache-pack`. The recipe will add this inside public.

Anyways, delete web/! This is it! Our app is *fully* in Flex! And we didn't even *need* to move all our files from src/AppBundle... though we *will* do that soon. And as far as bin/console is concerned, the app works!

But to really prove it's alive, let's try this in a browser and handle a few last details. That's next!

# Chapter 12: The Server & New IsGranted

Time to try our app! First, in `.env`, change the database name to `symfony3_tutorial`, or whatever the database name was called when you *first* setup the project. Now when we run `doctrine:migrations:status`... yes! We have a full database!

## Installing the Server

Let's start the built-in web server:



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

Surprise!

There are no commands defined in the "server" namespace.

Remember: with Flex, you opt *in* to features. Run:



```
$ composer require server
```

When it finishes, run:



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

Interesting - it started on `localhost:8001`. Ah, that's because the *old* server is still running and hogging port 8000! And woh! It's *super* broken: we've removed a *ton* of files it was using. Hit `Ctrl+C` to stop the server. Ah! It's so broken it doesn't want to stop! It's taking over! Close that terminal!

Start the server again:



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

It still starts on port 8001, but that's fine! Go back to your browser and load `http://localhost:8001`. Ha! It works! Check it out: Symfony 4.0.1.

Surf around to see if everything works: go to `/genus`. Looks great! Now `/admin/genus`. Ah! Looks terrible!

To use the `@Security` tag, you need to use the Security component and the ExpressionLanguage component.

## The New @IsGranted

Hmm. Let's do some digging! Open `src/AppBundle/Controller/Admin/GenusAdminController.php`. Yep! Here is the `@Security` annotation from FrameworkExtraBundle. The string we're passing to it is an *expression*, so we need to install the ExpressionLanguage.

But wait! I have a better idea. Google for SensioFrameworkExtraBundle and find its [GitHub page](#). Click on releases: the latest is 5.1.3. What version do we have? Open `composer.json`: woh! We're using version 3! Ancient!

Let's update this to `^5.0`.

82 lines | [composer.json](#)

```
1  {  
    ... lines 2 - 16  
17  "require": {  
    ... lines 18 - 23  
24    "sensio/framework-extra-bundle": "^5.0",  
    ... lines 25 - 42  
43  },  
    ... lines 44 - 80  
81 }
```

Then, run:



```
$ composer update sensio/framework-extra-bundle
```

to update *just* this library. Like with any major upgrade, look for a CHANGELOG to make sure there aren't any insane changes that will break your app.

So... why are we upgrading? So glad you asked: because the new version has a feature I *really* like! As soon as Composer finishes, go back to GenusAdminController. Instead of using @Security, use @IsGranted.

97 lines | [src/AppBundle/Controller/Admin/GenusAdminController.php](#)

```
    ... lines 1 - 12  
13  /**  
14   * @IsGranted("ROLE_MANAGE_GENUS")  
    ... line 15  
16   */  
17  class GenusAdminController extends Controller  
    ... lines 18 - 97
```

This is *similar*, but *simpler*. For the value, you *only* need to say: ROLE\_MANAGE\_GENUS.

Try it - refresh! Yes! We're sent to the login page - that's good! Sign in with password iliketurtles.

At this point... we're done! Unless... you want to move all of your classes from AppBundle directly into src/. I do! And it's much easier than you might think.

# Chapter 13: Bye Bye AppBundle

If you want to stop now, you can! Your old code lives in `src/AppBundle`, but it works! Over time, you can slowly migrate it directly into `src/`.

Or! We can keep going: take this final challenge head-on and move all our files at once! If you're not using PhpStorm... this will be a nightmare. Yep, this is one of those rare times when you really *need* to use it.

## [Moving your Files](#)

Open `AppBundle.php`. Then, right click on the `AppBundle` namespace and go to Refactor -> Move. The new namespace will be `App`. And below... yea! The target destination should be `src/`.

This says: change all `AppBundle` namespaces to `App` and move things into the `src/` directory. Try it! On the big summary, click OK!

```
10 lines | src/AppBundle.php
... lines 1 - 2
3 namespace App;
4
5 use Symfony\Component\HttpKernel\Bundle\Bundle;
6
7 class AppBundle extends Bundle
8 {
9 }
```

In addition to changing the namespace at the top of each file, PhpStorm is also searching for *references* to the namespaces and changing those too. Will it be perfect? Of course not! But that last pieces are pretty easy.

Woh! Yes! Everything is directly in `src/`. `AppBundle` is now empty, except for a `fixtures.yml` file. We're going to replace that file soon anyways.

Delete `AppBundle`! That felt *amazing*!

## [Refactoring tests/](#)

Let's do the same thing for the `tests/` directory... even though we only have one file. Open `DefaultControllerTest.php` and Refactor -> Move its namespace. In Flex, the namespace should start with `App\Tests`. Then, press F2 to change the directory to `tests/Controller`.

```
19 lines | tests/Controller/DefaultControllerTest.php
... lines 1 - 2
3 namespace App\Tests\Controller;
... lines 4 - 6
7 class DefaultControllerTest extends WebTestCase
8 {
... lines 9 - 17
18 }
```

Ok, Refactor! Nice! Now delete that `AppBundle`.

## [Cleaning up AppBundle](#)

With those directories gone, open `composer.json` and find the `autoload` section. Remove *both* `AppBundle` parts.

So... will it work? Probably not - but let's try! Refresh! Ah!

The file ../src/AppBundle does not exist in config/services.yaml

Ah, that makes sense. Open that file: we're still trying to import services from the old directory. Delete those two sections. And, even though it doesn't matter, remove AppBundle from the exclude above.

In routes.yaml, we *also* have an import. Remove it! Why? Annotations are already being loaded from src/Controller. And *now*, that's where our controllers live!

Oh, and change AppBundle to App for the homepage route - I can now even Command+Click into that class. Love it!

```
5 lines | config/routes.yaml
1 homepage:
2   path: /
3   defaults:
4     _controller: App\Controller\MainController::homepageAction
```

Back in services.yaml, we still have a lot of AppBundle classes in here: PhpStorm is *not* smart enough to refactor YAML strings. But, the fix is easy: Find all AppBundle and replace with App.

```
56 lines | config/services.yaml
... lines 1 - 17
18 App\:
... lines 19 - 23
24 App\Controller\:
... lines 25 - 33
34 App\Service\MarkdownTransformer:
... lines 35 - 37
38 App\Doctrine\HashPasswordListener:
... lines 39 - 40
41 App\Form\TypeExtension\HelpFormExtension:
... lines 42 - 44
45 App\Service\MessageManager:
... lines 46 - 49
50 App\EventSubscriber\AddNiceHeaderEventSubscriber:
... lines 51 - 56
```

Done! There is one last thing we need to undo: in config/packages/doctrine.yaml. Remove the AppBundle mapping we added.

So, what other AppBundle things haven't been updated yet? It's pretty easy to find out. At your terminal, run:

```
$ git grep AppBundle
```

Hey! Not too bad. And most of these are the same: calls to getRepository(). Start in security.yaml and do the same find and replace. You could do this for your *entire* project, but I'll play it safe.

```

41 lines | config/packages/security.yaml
... lines 1 - 2
3  security:
4      encoders:
5          App\Entity\User: bcrypt
... lines 6 - 10
11  providers:
12      our_users:
13          entity: { class: App\Entity\User, property: email }
... line 14
15  firewalls:
... lines 16 - 20
21  main:
... line 22
23      guard:
24          authenticators:
25              - App\Security\LoginFormAuthenticator
... lines 26 - 41

```

Now, *completely* delete the AppBundle.php file: we're *already* not using that. Next is GenusAdminController. Open that class. But instead of replacing everything, which *would* work, search for AppBundle. Ah! It's a getRepository() call!

Our project has a lot of these... and... well... if you're lazy, there's a secret way to fix it! Just change the alias in doctrine.yaml from App to AppBundle. Cool... but let's do it the right way! Use Genus::class.

```

97 lines | src/Controller/Admin/GenusAdminController.php
... lines 1 - 16
17  class GenusAdminController extends Controller
18  {
... lines 19 - 21
22      public function indexAction()
23      {
24          $genuses = $this->getDoctrine()
25              ->getRepository(Genus::class)
... lines 26 - 30
31  }
... lines 32 - 96
97  }

```

We have a few more in GenusController. Use SubFamily::class, User::class, Genus::class, GenusNote::class and GenusScientist::class.

```

144 lines | src/Controller/GenusController.php
... lines 1 - 17
18 class GenusController extends Controller
19 {
... lines 20 - 22
23     public function newAction()
24     {
... lines 25 - 26
27         $subFamily = $em->getRepository(SubFamily::class)
... lines 28 - 42
43         $user = $em->getRepository(User::class)
... lines 44 - 60
61     }
... lines 62 - 65
66     public function listAction()
67     {
... lines 68 - 69
70         $genuses = $em->getRepository(Genus::class)
... lines 71 - 75
76     }
... lines 77 - 80
81     public function showAction(Genus $genus, MarkdownTransformer $markdownTransformer, LoggerInterface $logger)
82     {
... lines 83 - 88
89         $recentNotes = $em->getRepository(GenusNote::class)
... lines 90 - 96
97     }
... lines 98 - 127
128     public function removeGenusScientistAction($genusId, $userId)
129     {
... lines 130 - 131
132         $genusScientist = $em->getRepository(GenusScientist::class)
... lines 133 - 141
142     }
143 }

```

Ok, back to the list! Ah, a few entities still have AppBundle. Start with Genus. The repositoryClass, of course! Change AppBundle to App. There's another reference down below on a relationship. Since all the entities live in the same directory, this can be shortened to just SubFamily.



```

223 lines | src/Entity/Genus.php
... lines 1 - 12
13  /**
14   * @ORM\Entity(repositoryClass="App\Repository\GenusRepository")
... line 15
16   */
17   class Genus
18   {
... lines 19 - 37
38   /**
... line 39
40   * @ORM\ManyToOne(targetEntity="App\Entity\SubFamily")
... line 41
42   */
43   private $subFamily;
... lines 44 - 221
222  }

```

Make the same change in GenusNote, SubFamily and User.

```

101 lines | src/Entity/GenusNote.php
... lines 1 - 6
7  /**
8   * @ORM\Entity(repositoryClass="App\Repository\GenusNoteRepository")
... line 9
10  */
11  class GenusNote
... lines 12 - 101

```

```

45 lines | src/Entity/SubFamily.php
... lines 1 - 6
7  /**
8   * @ORM\Entity(repositoryClass="App\Repository\SubFamilyRepository")
... line 9
10  */
11  class SubFamily
... lines 12 - 45

```

```

223 lines | src/Entity/User.php
... lines 1 - 11
12  /**
13   * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
... lines 14 - 15
16  */
17  class User implements UserInterface
... lines 18 - 223

```

Almost done! Next is GenusFormType: open that and change the data\_class to Genus::class.

```

109 lines | src/Form/GenusFormType.php
... lines 1 - 21
22 class GenusFormType extends AbstractType
23 {
... lines 24 - 60
61     public function configureOptions(OptionsResolver $resolver)
62     {
63         $resolver->setDefaults([
64             'data_class' => Genus::class
65         ]);
66     }
... lines 67 - 107
108 }

```

Then, finally, LoginFormAuthenticator. Update AppBundle:User to User::class.

```

90 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 20
21 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
22 {
... lines 23 - 56
57     public function getUser($credentials, UserProviderInterface $userProvider)
58     {
... lines 59 - 60
61         return $this->em->getRepository(User::class)
... line 62
63     }
... lines 64 - 88
89 }

```

Phew! Search for AppBundle again:

```

$ git grep AppBundle

```

They're gone! So... ahh... let's try it! Refresh! Woh! An "Incomplete Class" error? Fix it by manually going to /logout. What was that? Well, because we changed the User class, the User object in the session couldn't be deserialized. On production, your users shouldn't get an error, but they *will* likely be logged out when you first deploy.

Go back to /admin/genus, then login with weaverryan+1@gmail.com, password iliketurtles. Guys, we're done! We have a Symfony 4 app, built on the Flex directory structure, and with *no* references to AppBundle! And it was all done in a safe, gradual way.

To celebrate, I've added one last video with a few reasons to be *thrilled* that you've made it this far.

# Chapter 14: Flex Extras

Now that we're on Symfony 4 with Flex, I have *three* cool things to show you.

## [Repositories as a Service](#)

Start by opening GenusController: find listAction. Ah yes: this is a very classic setup: get the entity manager, get the repository, then call a method on it.

One of the annoying things is that - unless you add a bunch of extra config - repositories are *not* services and can *not* be autowired. Boo!

Well... that's not true anymore! Want your repository to be a service? Just make two changes. First, extend a new base class: ServiceEntityRepository.

```
52 lines | src/Repository/GenusRepository.php
... lines 1 - 5
6   use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
... lines 7 - 10
11  class GenusRepository extends ServiceEntityRepository
12  {
... lines 13 - 50
51 }
```

And second, override the `__construct()` function. But remove the `$entityClass` argument.

### Tip

Make sure the type-hint for the first argument is `RegistryInterface` not `ManagerRegistry`.

In the parent call, use `Genus::class`.

```
52 lines | src/Repository/GenusRepository.php
... lines 1 - 12
13  public function __construct(ManagerRegistry $registry)
14  {
15      parent::__construct($registry, Genus::class);
16  }
... lines 17 - 52
```

That might look weird at first... but with those *two* small changes, your repository is *already* being auto-registered as a service! Yep, back in listAction, add a new argument: `GenusRepository $genusRepository`. Use that below *instead* of fetching the EntityManager.

```

143 lines | src/Controller/GenusController.php
... lines 1 - 18
19 class GenusController extends Controller
20 {
... lines 21 - 66
67 public function listAction(GenusRepository $genusRepository)
68 {
69     $genuses = $genusRepository
70         ->findAllPublishedOrderedByRecentlyActive();
... lines 71 - 74
75 }
... lines 76 - 141
142 }

```

And that's it! Go to that page in your browser: `/genus`. Beautiful! Make that same change to your other repository classes when you want to.

## Fixtures as Services

Ok, cool thing #2: our fixtures are broken. Well... that's not the cool part. They're broken because we removed Alice, so everything explodes:

But, there's even *more* going on. Find your `composer.json` file and make sure the version constraint is `^3.0`.

```

81 lines | composer.json
1 {
... lines 2 - 42
43 "require-dev": {
... lines 44 - 45
46     "doctrine/doctrine-fixtures-bundle": "^3.0"
47 },
... lines 48 - 79
80 }

```

Then, run:

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

Version 3 of this bundle is *all* new... but not in a "broke everything" kind of way. Before, fixture classes were loaded because they lived in an *exact* directory: usually `DataFixtures\ORM` in your bundle. And if you needed to access services, you extended `ContainerAwareFixture` and fetched them directly from the container.

Well, no more! In the new version, your fixtures are *services*, and so they act like *everything* else. You can even put them *anywhere*.

When Composer finishes, download one more package:

```
$ composer require fzaninotto/faker
```

### Tip

Even better would be `composer require fzaninotto/faker --dev`!

This isn't needed by `DoctrineFixturesBundle`, but we *are* going to use it. In fact, if you downloaded the course code, you should have a `tutorial/` directory with an `AllFixtures.php` file inside. Copy that and put it directly into `DataFixtures`.

```

142 lines | src/DataFixtures/AllFixtures.php
... lines 1 - 15
16 class AllFixtures extends Fixture
17 {
... lines 18 - 22
23 public function load(ObjectManager $manager)
24 {
25     $this->faker = Factory::create();
26     $this->addSubFamily($manager);
27     $this->addGenus($manager);
28     $this->addGenusNote($manager);
29     $this->addUser($manager);
30     $this->addGenusScientist($manager);
31
32     $manager->flush();
33 }
... lines 34 - 140
141 }

```

Then, delete the old ORM directory. This is our new fixture class: all we need to do is extend Fixture from the bundle, and the command instantly recognizes it. If you need services, just add a constructor and use autowiring!

Let's go check on Faker. Ah, it's done! Inside the class, Faker allows me to generate really nice, random values. Does it work? Reload the fixtures:

```
$ ./bin/console doctrine:fixtures:load
```

It sees our class immediately and... it works! Fixtures are services... and they work great.

## MakerBundle

Ready for one last cool thing? Run:

```
$ composer require maker --dev
```

This installs the MakerBundle: Symfony's new code generator. Code generation is of course optional. But with this bundle, you'll be able to develop new features faster than ever. Need a console command, an event subscriber or a Twig extension? Yep, there's a command for that.

What's everything it can do? Run:

```
$ ./bin/console list make
```

Right now, it has about 10 commands - but there are a lot more planned: this bundle is only about 1 month old!

Let's try one of these commands!

```
$ ./bin/console make:voter
```

Call it RandomAccessVoter: we'll create a voter that randomly gives us access. Fun! Open the new class in `src/Security/Voter`. This comes pre-generated with real-world example code. In `supports()`, return `$attribute === 'RANDOM_ACCESS'`. Our voter will vote when someone calls `isGranted()` with `RANDOM_ACCESS`.

```

21 lines | src/Security/Voter/RandomAccessVoter.php
... lines 1 - 8
9  class RandomAccessVoter extends Voter
10 {
11     protected function supports($attribute, $subject)
12     {
13         return $attribute === 'RANDOM_ACCESS';
14     }
... lines 15 - 19
20 }

```

Then, for `voteOnAttribute()`, return `random_int(0, 10) > 5`.

```

21 lines | src/Security/Voter/RandomAccessVoter.php
... lines 1 - 15
16     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
17     {
18         return random_int(0, 10) > 5;
19     }
... lines 20 - 21

```

Now we need to go and update some configuration, right? No! This class is *already* being used! Open `GenusController` and... above `newAction()`, add `@IsGranted("RANDOM_ACCESS")`.

```

145 lines | src/Controller/GenusController.php
... lines 1 - 19
20  class GenusController extends Controller
21  {
22      /**
... line 23
24      * @IsGranted("RANDOM_ACCESS")
25      */
26      public function newAction()
... lines 27 - 143
144 }

```

Done! Try it: go to `/genus/new`. Ha! It sent us to the login page - that proves its working. Login with `iliketurtles` and... access granted! Refresh - granted! Refresh - denied!

All that by running 1 command and changing about 3 lines. Welcome to Symfony 4.

## [Let's go Symfony 4!](#)

Hey, we're done! Upgrading to the Flex structure *is* work, but I hope you're as happy as I am about the result! To go further with Flex and Symfony 4, check out our [Symfony Track](#): we're going to *start* a project with Flex and *really* do things right.

All right guys. Seeya next time!

