

# Symfony 3.3: Upgrade, Autowiring & Autoconfigure



With <3 from SymfonyCasts

# Chapter 1: Upgrading to Symfony 3.3!

Yo peeps! Wow, Symfony 3.3! Actually, that version number *sounds* kinda boring. I mean, 3.3? Really? I expect cool stuff to happen in version 3.3?

Well..... normally I'd agree! But Symfony 3.3 is special: it has some big, awesome, exciting, dinosauric changes to how you configure services. And that's why we made this crazy tutorial in the first place: to take those head on.

If you're not already comfortable with how services work in Symfony, stop, drop and roll... and go watch our [Symfony Fundamentals](#) course. Then come back.

## [Symfony's Backwards-Compatibility Awesomeness Promise](#)

So, how can we upgrade to Symfony 3.3? I have no idea. No, no, no - upgrading Symfony is always *boring* and simple. That's because of Symfony's backwards compatibility promise: we do not break things when you upgrade a minor version, like 3.2 to 3.3. It's Symfony's secret super power. I *will* show you how to upgrade, but it's easy.

As always, you should totally bake cookies while watching this tutorial... and also, code along with me! Click download on this page and unzip the file. Inside, you'll find a `start/` directory with the same code you see here. Open the `README.md` file to find poetic prose... which doubles as setup instructions.

## [Upgrading](#)

Time to upgrade! Open up `composer.json`:

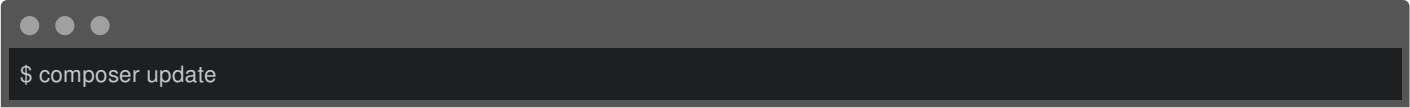
```
68 lines | composer.json
1  {
    ... lines 2 - 15
16  "require": {
    ... line 17
18      "symfony/symfony": "3.1.*",
    ... lines 19 - 30
31  },
    ... lines 32 - 66
67  }
```

This is actually a Symfony 3.1 project, but that's no problem. I'll change the version to 3.3.0-RC1 because Symfony 3.3 has not been released yet at the time of this recording. You should use 3.3.\*:

```
68 lines | composer.json
1  {
    ... lines 2 - 15
16  "require": {
    ... line 17
18      "symfony/symfony": "3.3.0-RC1",
    ... lines 19 - 30
31  },
    ... lines 32 - 66
67  }
```

By the way, we *could* update other packages too, but that's optional. Often, I'll visit the Symfony Standard Edition, make sure I'm on the correct branch, and see what versions it has in its [composer.json file](#).

For now, we'll *just* change the Symfony version. To upgrade, find your favorite terminal, move into the project, and run:



```
$ composer update
```

This will update *all* of your packages to the latest versions allowed in composer.json. You could also run `composer update symfony/symfony` to *only* update that package.

Then.... we wait! While Jordi and the Packagists work on our new version.

## [Removing trusted\\_proxies](#)

It downloads the new version then... woh! An error! Occasionally, if we have a *really* good reason, Symfony will change something that *will* break your app on upgrade... but only in a huge, obvious and unavoidable way:

### Tip

The "framework.trusted\_proxies" configuration key has been removed in Symfony 3.3. Use the `Request::setTrustedProxies()` method in your front controller instead.

This says that a `framework.trusted_proxies` configuration was removed. Open up `app/config/config.yml`: there it is! Just take that out:




```
80 lines | app/config/config.yml
```

```
... lines 1 - 11
12 framework:
... lines 13 - 26
27   trusted_hosts: ~
28   session:
... lines 29 - 80
```

The option was removed for security reasons. If you *did* have a value there, check out the docs to see the replacement.

Ok, *even* though `composer update` exploded, it *did* update our `composer.lock` file and download the new version. Just to be sure, I'll run `composer install` to make sure everything is happy:




```
$ composer install
```

Yes!

## [Enabling the WebServerBundle](#)

Ok, start the built-in web server:



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

Ahhh!

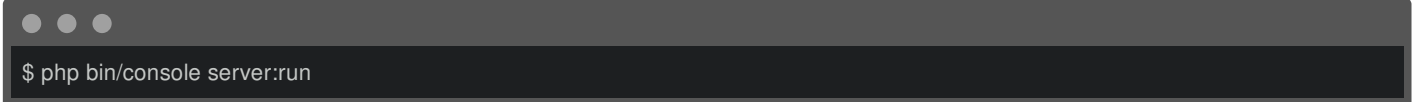
There are no commands defined in the "server" namespace

What happened to `server:run`? Well actually, Symfony is becoming more and more decoupled. That command now lives in its own bundle. Open `app/AppKernel.php`. And in the dev environment only, add `$bundles[] = new WebServerBundle();`

```
60 lines | app/AppKernel.php
... lines 1 - 2
3 use Symfony\Bundle\WebServerBundle\WebServerBundle;
... lines 4 - 6
7 class AppKernel extends Kernel
8 {
9     public function registerBundles()
10    {
11    ... lines 11 - 27
28        if (in_array($this->getEnvironment(), array('dev', 'test'), true)) {
12    ... lines 29 - 33
34            $bundles[] = new WebServerBundle();
35        }
13    ... lines 36 - 37
38    }
14    ... lines 39 - 58
59 }
```

That bundle still *comes* with the symfony/symfony package, but it wasn't enabled in our project. A *new* Symfony 3.3 project already has this line.

Flip back to your terminal and try again:



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

Got it! Find your browser and open up <http://localhost:8000> to find the famous Aquanaut project that we've been working on in this Symfony series.

Okay, we're set up and we are on Symfony 3.3. But we are *not* yet using any of the cool, new dependency injection features. Let's fix that!

## Chapter 2: `_defaults`, autowire & autoconfigure

If you started a brand new Symfony 3.3 project, its `services.yml` file will look like [this](#).

You're actually seeing at least *four* new features all at once! Wow. All of this is built on top of the existing service configuration system and you need to "opt in" to any of the new features. That means that the traditional way of configuring services that you've been using until now still works and always will. Winning!

But even if you ultimately choose *not* to use some of these new features, you need to understand how they work, because you'll see them a lot. For example, the Symfony documentation has already been updated to assume you're using these.

First, a word of warning: using the new features is *fun*. But, upgrading an existing project to use them... well... it's *less* fun. It takes some work, and when you're done... your project works... the same as before. I'm also going to show you the ugliest parts of the new system so you can handle them in your project. But stick with me! At the end, we'll use the new features to build some new code. And that, is a *blast*.

### `_defaults`: File-side Service Defaults

Let's look at this `_defaults` thing first. Open up your `app/config/services.yml` file. At the top of the services section - though order isn't important - add `_defaults`. Then below that, `autowire: true` and `autoconfigure: true`:

```
41 lines | app/config/services.yml
... lines 1 - 5
6  services:
7    _defaults:
8      autowire: true
9      autoconfigure: true
... lines 10 - 41
```

Let's unpack this. First, `_defaults` is a new special keyword that allows you to set default configuration for all services in *this* file. It's equivalent to adding `autowire` and `autoconfigure` keys under every service in this file only. And of course, any config from `_defaults` can be overridden by a specific service.

### Autowiring

Autowiring is *not* new: we talk about it in our Symfony series. When a service is autowired, it means that its constructor arguments are automatically configured when possible by reading type-hints. For example, the `MarkdownExtension` is autowired:

```
43 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 10
11  app.markdown_extension:
12    class: AppBundle\Twig\MarkdownExtension
13    tags:
14      - { name: twig.extension }
15    #arguments: ['@app.markdown_transformer']
16    autowire: true
... lines 17 - 43
```

And its first constructor argument is type-hinted with `MarkdownTransformer`:

```

35 lines | src/AppBundle/Twig/MarkdownExtension.php
... lines 1 - 2
3 namespace AppBundle\Twig;
4
5 use AppBundle\Service\MarkdownTransformer;
6
7 class MarkdownExtension extends \Twig_Extension
8 {
... lines 9 - 10
11     public function __construct(MarkdownTransformer $markdownTransformer)
12     {
... line 13
14     }
... lines 15 - 33
34 }

```

Thanks to that, Symfony determines which service to pass here.

The way that autowiring works *has* changed in Symfony 3.3. But more on that later.

Since we have autowire under `_defaults`, we can remove it from everywhere else: it's redundant. And yes, this *does* mean that some services that were *not* autowired before are *now* set to `autowire: true`. For example, `app.markdown_transformer` is now being autowired:

```

43 lines | app/config/services.yml
... lines 1 - 5
6 services:
7     app.markdown_transformer:
8         class: AppBundle\Service\MarkdownTransformer
9         arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
... lines 10 - 43

```

But... that's no problem! Both of its arguments are being *explicitly* set:

```

35 lines | src/AppBundle/Service/MarkdownTransformer.php
... lines 1 - 2
3 namespace AppBundle\Service;
4
5 use Doctrine\Common\Cache\Cache;
6 use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
7
8 class MarkdownTransformer
9 {
... lines 10 - 12
13     public function __construct(MarkdownParserInterface $markdownParser, Cache $cache)
14     {
... lines 15 - 16
17     }
... lines 18 - 33
34 }

```

so autowiring simply doesn't do anything. Setting `autowire: true` under `_defaults` is safe to add to an existing project.

## Autoconfigure

Next, this `autoconfigure` key *is* a brand new feature. When a service is autoconfigured, it means that Symfony will automatically *tag* it when possible. For example, our `MarkdownExtension` extends `\Twig_Extension`:

```

35 lines | src/AppBundle/Twig/MarkdownExtension.php
... lines 1 - 6
7  class MarkdownExtension extends \Twig_Extension
8  {
... lines 9 - 33
34 }

```

Which implements Twig\_ExtensionInterface:

```

abstract class Twig_Extension implements Twig_ExtensionInterface
{
    // ...
}

```

That's actually the important part. When a service is autoconfigured and its class implements Twig\_ExtensionInterface, the twig.extension tag is *automatically* added for you:

```

43 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 10
11  app.markdown_extension:
... line 12
13      tags:
14          - { name: twig.extension }
... lines 15 - 43

```

Basically, Symfony is saying:

Hey! I see you configured a service that implements Twig\_ExtensionInterface. Obviously, that's a Twig extension, so let me configure it for you.

```

41 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 14
15  app.markdown_extension:
16      class: AppBundle\Twig\MarkdownExtension
17      #arguments: ['@app.markdown_transformer']
... lines 18 - 41

```

This works for many - but not all tags. It does *not* work for doctrine.event\_subscriber or form.type\_extension:

```

41 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 21
22  app.doctrine.hash_password_listener:
23      class: AppBundle\Doctrine\HashPasswordListener
24      tags:
25          - { name: doctrine.event_subscriber }
26
27  app.form.help_form_extension:
28      class: AppBundle\Form\TypeExtension\HelpFormExtension
29      tags:
30          - { name: form.type_extension, extended_type: Symfony\Component\Form\Extension\Core\Type\FormType }
... lines 31 - 41

```

Because it has an `extended_type` tag option... which the system can't guess for you. When you're developing a feature, the docs will tell you whether or not you need to add the tag manually. If you *do* add a tag, even though you didn't need to, no problem! Your tag takes precedence.

So, all our services are autowired and autoconfigured! But, it doesn't make any difference, besides shortening our config *just* a little. And when we refresh, everything still works!



# Chapter 3: Service Class Name as Service id

Traditionally, our service ids looked like this: some underscored, lowercased version of the class name:

```
41 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 10
11 app.markdown_transformer:
12   class: AppBundle\Service\MarkdownTransformer
13   arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
... lines 14 - 41
```

That's still *totally* legal. But in Symfony 3.3, the best practice has changed: a service's id should *now* be equal to its *class name*. I know, crazy, right! This makes life simpler. First, you don't need to invent an arbitrary service id. And second, class names as ids will work much better with autowiring and service auto-registration... as you'll see soon.

The only complication is when you have multiple services that point to the same class. We *will* handle this:

```
41 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 31
32 app.encouraging_message_generator:
33   class: AppBundle\Service\MessageGenerator
... lines 34 - 36
37 app.discouraging_message_generator:
38   class: AppBundle\Service\MessageGenerator
... lines 39 - 41
```

## Using Class Service ids

So, let's start changing service ids! I'll copy the old `app.markdown_transformer` id and replace it with `AppBundle\Service\MarkdownTransformer`. When your service id is your class name, you can actually remove the class key to shorten things:

```
39 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 13
14 AppBundle\Service\MarkdownTransformer:
15   arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
... lines 16 - 39
```

Nice!

But when we did that... we broke our app! Any code referencing the old service id will explode! We could hunt through our code and find those now, but let's save that for later. There's a faster, *safer* way to upgrade to the new format.

## Create Legacy Aliases to not Break Things

Create a new file in config called `legacy_aliases.yml`. Inside, add the normal services key, then paste the old service id set to `@`. Copy the new service id - the class name - and paste it: `@AppBundle\Service\MarkdownTransformer`:

```

7 lines | app/config/legacy_aliases.yml
1  services:
2    app.markdown_transformer: '@AppBundle\Service\MarkdownTransformer'
... lines 3 - 7

```

This creates something called a service *alias*. This is *not* a new feature, though this shorter syntax *is*. A service alias is like a symbolic link: whenever some code asks for the `app.markdown_transformer` service, the `AppBundle\Service\MarkdownTransformer` service will be returned. This will keep our old code working with *no* effort.

To load that file, at the top of `services.yml`, add an `imports` key with `resource` set to `legacy_aliases.yml`:

```

39 lines | app/config/services.yml
1  # Learn more about services, parameters and containers at
2  # http://symfony.com/doc/current/book/service_container.html
3  imports:
4    - { resource: legacy_aliases.yml }
... lines 5 - 39

```

Now, our app is happy again.

## Updating all of the Service

Let's repeat that for the rest of our services. Honestly, when we upgraded KnpUniversity, this was the most tedious step: going one-by-one, copying each service id, copying the class name, and then adding it to `legacy_aliases.yml`. We wrote a dirty script that used the `Yaml` class to load `services.yml` and at least create the `legacy_aliases.yml` file for us.

Notice that `LoginFormAuthenticator` has *no* configuration anymore! Cool! Just set it to a `~`:

```

39 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 19
20  AppBundle\Security\LoginFormAuthenticator: ~
... lines 21 - 39

```

Perfect! Let's finish the rest:

```

39 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 13
14  AppBundle\Service\MarkdownTransformer:
15    arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
16
17  AppBundle\Twig\MarkdownExtension:
18    #arguments: ['@app.markdown_transformer']
19
20  AppBundle\Security\LoginFormAuthenticator: ~
21
22  AppBundle\Doctrine\HashPasswordListener:
23    tags:
24      - { name: doctrine.event_subscriber }
25
26  AppBundle\Form\TypeExtension\HelpFormExtension:
27    tags:
28      - { name: form.type_extension, extended_type: Symfony\Component\Form\Extension\Core\Type\FormType }
... lines 29 - 39

```

7 lines | [app/config/legacy\\_aliases.yml](#)

```
1  services:
2    app.markdown_transformer: '@AppBundle\Service\MarkdownTransformer'
3    app.markdown_extension: '@AppBundle\Twig\MarkdownExtension'
4    app.security.login_form_authenticator: '@AppBundle\Security\LoginFormAuthenticator'
5    app.doctrine.hash_password_listener: '@AppBundle\Doctrine\HashPasswordListener'
6    app.form.help_form_extension: '@AppBundle\Form\TypeExtension\HelpFormExtension'
```

## Multiple Services for the Same Class

39 lines | [app/config/services.yml](#)

... lines 1 - 8

```
9  services:
```

... lines 10 - 29

```
30  app.encouraging_message_generator:
31    class: AppBundle\Service\MessageGenerator
```

... lines 32 - 34

```
35  app.discouraging_message_generator:
36    class: AppBundle\Service\MessageGenerator
```

... lines 37 - 39

The last two services are a problem: we *can't* set the id to the class name, because the class is the same for each! When this happens, use the old id naming convention. We're going to talk more about this situation soon.

And, we're done! We just changed all of the service ids to class names... but thanks to `legacy_aliases.yml`, our code won't break. This may not have felt significant, but it was actually a big step forward. Now, we can talk about private services.

# Chapter 4: Making all Services Private

There is *one* more key that lives under `_defaults` in a new Symfony 3.3 project: `public: false`:

```
42 lines | app/config/services.yml
... lines 1 - 8
9  services:
10  _defaults:
... lines 11 - 12
13      public: false
... lines 14 - 42
```

The idea of public versus private services is *not* new in Symfony 3.3... but defaulting all services to private *is* new, and it's a *critical* change. Thanks to this, *every* service in this file is now *private*. What does that mean? One simple thing: when a service is private, you *cannot* fetch it directly from the container via `$container->get()`. So, for example, `$container->get('AppBundle\Service\MarkdownTransformer')` will *not* work.

## Tip

In Symfony 3.3, *sometimes* you can fetch a private service directly from the container. But doing this has been deprecated and will be removed in Symfony 4.

But, *everything* else works the same: I can pass this service as an argument and it can be autowired into an argument. *Only* the `$container->get()` usage changed.

In our app, making this change is safe! We *just* created all of these service ids a minute ago and they're not being used anywhere yet, definitely not with `$container->get()`. The exception is the last two services: we *might* be fetching these services directly from the container. And in fact, I know we are: in `src/AppBundle/Controller/Admin/GenusAdminController.php`. Down in `editAction()`, we're fetching both services via `$this->get()`, which is a shortcut for `$this->container->get()`:

```

96 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
... lines 1 - 15
16 class GenusAdminController extends Controller
17 {
... lines 18 - 63
64     public function editAction(Request $request, Genus $genus)
65     {
... lines 66 - 69
70         if ($form->isSubmitted() && $form->isValid()) {
... lines 71 - 76
77             $this->addFlash(
78                 'success',
79                 $this->get('app.encouraging_message_generator')->getMessage()
80             );
... lines 81 - 84
85         } elseif ($form->isSubmitted()) {
86             $this->addFlash(
87                 'error',
88                 $this->get('app.discouraging_message_generator')->getMessage()
89             );
90         }
... lines 91 - 94
95     }
96 }

```

That means, for now, to keep our app working, add `public: true` under each service:

```

42 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 30
31  app.encouraging_message_generator:
... lines 32 - 34
35      public: true
... line 36
37  app.discouraging_message_generator:
... lines 38 - 40
41      public: true

```

Aliases can *also* be private... but in `legacy_aliases.yml`, there is no `_defaults` key with `public: false`. So, these are all *public* aliases... which is exactly what we want, ya know, because we're trying *not* to break our app!

Like with *every* step so far, our app should still work fine. Woohoo! But... you may be wondering *why* we made the services private. Doesn't this just make our services harder to use? As we'll learn soon, when you use private services, it becomes *impossible* to make a mistake and accidentally reference a non-existent service. Private services are going to make our app *even* more dependable than before. And also, a little bit faster.

Next, let's talk about the *biggest* change to this file: auto-registration of all classes in `src/AppBundle` as services. Woh.

# Chapter 5: Auto-Registering All Services

Go back to the Symfony Standard Edition's [services.yml file for Symfony 3.3](#). These two sections are the most *fundamental* change to the Symfony 3.3 service configuration. Copy the first section, then find our services.yml and, after `_defaults`, paste:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
10  _defaults:
... lines 11 - 14
15  # makes classes in src/AppBundle available to be used as services
16  # this creates a service per class whose id is the fully-qualified class name
17  AppBundle\:
18    resource: '../src/AppBundle/*'
19    # you can exclude directories or files
20    # but if a service is unused, it's removed anyway
21    exclude: '../src/AppBundle/{Entity,Repository}'
... lines 22 - 57
```

Woh.

This auto-registers *each* class in `src/AppBundle` as a service. There are a few important things to know. First, the id for each service is the full *class name*, just like we've been doing with *our* services. And second, thanks to `_defaults`, these new services are autowired and autoconfigured. And that means... in a lot of cases, you won't need to manually register your services at all anymore. Nope, as soon as you put a class in `src/AppBundle`, Symfony will autowire and autoconfigure it. That means you can start using it with *zero* config. And as we'll see soon, if the service *can't* be autowired, you'll get a clear error with details on what to do.

## Auto-registering ALL Classes as Services? Are you Insane?

Now... I *bet* I know what you're thinking:

Ryan, are you completely insane!? You can't auto-register everything in `src/AppBundle` as a service!? Some classes - like `DataFixtures`! - are simply *not* meant to be services! You've gone mad sir!

Ok, this *seems* like a fair argument... but actually, it's not. What if I told you that the total number of services in the container before *and after* adding this section is the same. Yep! To prove it, comment out this auto-registration code. Then, go to your terminal, open a new tab, and run:

```
$ php bin/console debug:container | wc -l
```

This will basically count the number of services returned. Ok - 262! *Uncomment* that code. Now, I'm registering *all* classes from `src/AppBundle` as services. Try the command again:

```
$ php bin/console debug:container | wc -l
```

It's the same! How is that possible?

Remember, all of these new services are *private*. And that's *very* important. It means that none of the services can be referenced via `$container->get()`. And Symfony's container is *so* incredible that - right before it dumps the cached container, it finds all private services that have not been referenced, and *removes* them from the container. This means that even though it *looks* like we're registering *every* class inside of `src/AppBundle` as a service, that's actually not true!

A better way to think of it is this: each class in `src/AppBundle` is *available* to be used as a service. This means we can reference it as argument in `services.yml` or type-hint its class in a constructor so that it's autowired. But if you do *not* reference one of these classes, that service is automatically removed.

## [Excluding some Paths](#)

You've probably also noticed this `exclude` key:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 16
17  AppBundle\:
... line 18
19      # you can exclude directories or files
20      # but if a service is unused, it's removed anyway
21      exclude: ['../src/AppBundle/{Entity,Repository}']
... lines 22 - 57
```

Actually, for the reasons we just discussed... this isn't that important. You *can* exclude certain files or directories if you want. But most of the time, that's not needed: if you don't reference a class, it's removed from the container for you.

However, if you *do* have entire directories that should *not* be auto-registered, adding it here is nice. It'll give you a slight performance boost in the dev environment because Symfony won't need to watch those files for changes. And for a subtle technical reason, the `Entity` directory *must* be excluded. We also excluded the `Repository` directory. You actually *can* register these as services... but you need to configure them manually to use a factory. Basically, auto-registering and autowiring doesn't work, so we might as well ignore them.

## [Overriding Auto-Registered Services](#)

Phew! So the idea is that we *start* by auto-registering each class as a service with these 3 lines. Then, if you *do* need to add some more configuration - like a tag, or an argument that can't be autowired, you can do that! Just override the auto-registered service below: use the class name as the key, then do whatever you need. Symfony automates as much configuration as possible so that you only need to fill in the rest.

# Chapter 6: Controllers as Services

There is one other piece of auto-registration code in the [new services.yml file for Symfony 3.3](#) involving the Controller/ directory. Copy that!

Then, paste it in our file:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 16
17 AppBundle\:
... lines 18 - 22
23     # controllers are imported separately to make sure they're public
24     # and have a tag that allows actions to type-hint services
25 AppBundle\Controller\:
26     resource: '../src/AppBundle/Controller'
27     public: true
28     tags: ['controller.service_arguments']
... lines 29 - 57
```

This auto-registers each class in src/AppBundle/Controller as a service... which was already done above:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 14
15     # makes classes in src/AppBundle available to be used as services
16     # this creates a service per class whose id is the fully-qualified class name
17 AppBundle\:
18     resource: '../src/AppBundle/**'
19     # you can exclude directories or files
20     # but if a service is unused, it's removed anyway
21     exclude: '../src/AppBundle/{Entity,Repository}'
22
23     # controllers are imported separately to make sure they're public
24     # and have a tag that allows actions to type-hint services
25 AppBundle\Controller\:
... lines 26 - 57
```

This overrides those services to make sure that anything in Controller/ is public and has this very special tag. In Symfony 3.3 - controllers are the *one* service that *must* be public. And the tag gives us a special controller argument autowiring super power that we'll see soon.

## Controllers are Services!?

But wait, our controllers are services!? Yes! In Symfony 3.3, we recommend that all of your controllers be registered as a service. And it's so awesome! You can still use all of your existing tricks. You can still extend Symfony's base Controller class and use its shortcuts. You can even still fetch public services directly from the container. But now, you can *also* use proper dependency injection if you want to. *And*, as long as your service's id is the class name, all of the existing routing config formats will automatically know to use your service. In other words, this just works.

## Removing Unnecessary Services



Now that we're auto-registering each class as a service, we can *remove* these two services:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 32
33  AppBundle\Twig\MarkdownExtension:
34      #arguments: ['@app.markdown_transformer']
35
36  AppBundle\Security\LoginFormAuthenticator: ~
... lines 37 - 57
```

They're *still* being registered, but since we don't need to add any further configuration, they're redundant!

Woohoo! And when we refresh our app, everything still works! Controllers as services with *four* lines of code!

# Chapter 7: Understanding Autowiring Logic

All this auto-registration stuff works smoothly: unused services are removed, you can override a service to configure it further, and if there are any problems autowiring a service, you get a clear exception when you try to load *any* page.

But, there's one place where it's not as clean: when you have multiple services that point to the same class:

```
57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 45
46  app.encouraging_message_generator:
47      class: AppBundle\Service\MessageGenerator
... lines 48 - 51
52  app.discouraging_message_generator:
53      class: AppBundle\Service\MessageGenerator
... lines 54 - 57
```

In other words, when you have services where the id *can't* be the class name. Right now, there are *two* services for the MessageGenerator class, right? Actually, there are *three*.

Find your terminal and run:

```
$ php bin/console debug:container --show-private | grep Message
```

Surprise! This lists our two services *plus* a third service that was automatically registered. Now technically, that's not a problem. Nobody is referencing the new auto-registered service, so it's removed. But, it *may* cause an issue with autowiring in the future.

## [How Autowiring Works](#)

First, we need to talk about how autowiring works. It's *super* simple... or at least, it *will* be simple in Symfony 4.

Let's look at HashPasswordListener. As you can see, this is type-hinted with UserPasswordEncoder:

```
65 lines | src/AppBundle/Doctrine/HashPasswordListener.php
... lines 1 - 7
8  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoder;
9
10 class HashPasswordListener implements EventSubscriber
11 {
... lines 12 - 13
14     public function __construct(UserPasswordEncoder $passwordEncoder)
15     {
... line 16
17     }
... lines 18 - 63
64 }
```

In services.yml, this argument is *not* specified:

```

57 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 37
38  AppBundle\Doctrine\HashPasswordListener:
39      tags:
40      - { name: doctrine.event_subscriber }
... lines 41 - 57

```

It works because the container is autowiring it.

But how does it know *which* service to pass here? First, autowiring looks for a service whose id *exactly* matches the type-hint. In other words, it looks for a service whose id is `Symfony\Component\Security\Core\Encoder\UserPasswordEncoder`:

```

65 lines | src/AppBundle/Doctrine/HashPasswordListener.php
... lines 1 - 7
8  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoder;
9
10 class HashPasswordListener implements EventSubscriber
11 {
... lines 12 - 13
14     public function __construct(UserPasswordEncoder $passwordEncoder)
15     {
... line 16
17     }
... lines 18 - 63
64 }

```

If that exists, it's used... always. This is the *main* way that autowiring works. It's not magic: you explicitly configure a service or alias for each type-hint.

That's also why we started using class names as our service ids: this allows all of our services to be autowired into other classes.

Before we keep going: I want to repeat this *one* more time: autowiring works by looking for a service - or alias - whose id exactly matches the type-hint. It's that simple.

I wanted to repeat that, because - if there is *not* a service whose id matches the type-hint, autowiring tries two other things. But, but but! One of these things will be removed in Symfony 4 and the other thing will never happen in practice.

### The Deprecated way Autowiring Works

But, we need to talk about these two things so that your 3.3 app makes sense. If there is *not* a service whose id matches the type-hint, autowiring will look at *every* service in the container and see which services have the class or interface. If two or more services have it, Symfony will throw a clear exception. We'll see that later. But if exactly *one* service is found, that service is used. However, *this* is deprecated and will *not* work in Symfony 4.

If autowiring finds *zero* services that have the class, it will auto-register that class as a new autowired service. But, this will *never* happen for us... because we've auto-registered *all* of our classes as services. And this magic auto-registration is not done for vendor classes.

So basically, autowiring looks for a service whose id exactly matches the type-hint. And in Symfony 4, if that service doesn't exist, it will throw a clear exception. Until then, some of your type-hints *may* still be autowired via the old deprecated logic. We'll see and fix this in a few minutes.

Next, let's see how this relates to our MessageGenerator services, and how we can fix them.

## Chapter 8: Problematic Multi-Class Services

Since autowiring works by finding a service whose id exactly matches the type-hint, it means that in the future, if someone type-hints `MessageGenerator` as an argument, this *first*, auto-registered service will be used! And actually, in this case, that's not a *huge* problem: `MessageGenerator` has a required constructor argument:

```
19 lines | src/AppBundle/Service/MessageGenerator.php
... lines 1 - 4
5 class MessageGenerator
6 {
... lines 7 - 8
9     public function __construct(array $messages)
10    {
... line 11
12    }
... lines 13 - 17
18 }
```

So our app would explode with an exception. But still, that's a little unexpected... we really *don't* want this third service.

### [Fix 1\) Don't auto-register the Service](#)

There are three ways to fix this, depending on your app. First, you can explicitly exclude it. At the end of exclude, add `Service/MessageGenerator.php`:

```
52 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 16
17  AppBundle:
... lines 18 - 20
21      exclude: ['../src/AppBundle/{Entity,Repository,Service/MessageGenerator.php}']
... lines 22 - 52
```

Now, go back to your terminal:

```
$ php bin/console debug:container --show-private
```

Boom! Back to just the two services. And because neither id is set to the class name, if someone tries to autowire using the `MessageGenerator` type-hint, they'll see a clear exception asking them to explicitly wire the value they want for that argument. In other words, `MessageGenerator` cannot be used for autowiring... but that's ok!

### [Fix 2\) Choose one Service for Autowiring](#)

Another solution is to choose one of your services to be the one that's used for autowiring. For example, suppose the `app.encouraging_message_generator` is used *much* more often, so you want that to be autowired for the `MessageGenerator` type-hint:

```

52 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 40
41  app.encouraging_message_generator:
42      class: AppBundle\Service\MessageGenerator
43      arguments:
44          - ['You can do it!', 'Dude, sweet!', 'Woot!']
45      public: true
... lines 46 - 52

```

Cool! Copy the old service id. Then, open `legacy_aliases.yml` and do the same thing we did before. I'll use the class as the service id, then setup an alias:

```

50 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 40
41  AppBundle\Service\MessageGenerator:
... lines 42 - 50

```

```

8 lines | app/config/legacy_aliases.yml
1  services:
... lines 2 - 6
7  app.encouraging_message_generator: '@AppBundle\Service\MessageGenerator'

```

We *still* have two services in the container for the same class. But the first will be used for autowiring. If you need to pass the second instance as an argument, you'll need to explicitly configure that. We'll see how in a few minutes.

### [Fix 3\) Refactor to a Single Class](#)

The final option - which is really practical, but a bit more controversial - is to refactor your application to avoid this situation. For example, if you downloaded the start code, you should have a `tutorial/` directory with a `MessageManager.php` file inside. Copy that and paste it into the `Service/` directory:

26 lines | [src/AppBundle/Service/MessageManager.php](#)

... lines 1 - 2

```
3 namespace AppBundle\Service;
4
5 class MessageManager
6 {
7     private $encouragingMessages = array();
8     private $discouragingMessages = array();
9
10    public function __construct(array $encouragingMessages, array $discouragingMessages)
11    {
12        $this->encouragingMessages = $encouragingMessages;
13        $this->discouragingMessages = $discouragingMessages;
14    }
15
16    public function getEncouragingMessage()
17    {
18        return $this->encouragingMessages[array_rand($this->encouragingMessages)];
19    }
20
21    public function getDiscouragingMessage()
22    {
23        return $this->discouragingMessages[array_rand($this->discouragingMessages)];
24    }
25 }
```

This class has two arguments - `$encouragingMessages` and `$discouragingMessages` - and a method to fetch a message from each. It's basically a combination of our two `MessageGenerator` services.

Technically, this class is already registered as a service. But of course, these two arguments can't be autowired. So, configure the service explicitly: `AppBundle\Service\MessageManager`: and under arguments, pass the encouraging messages and the discouraging messages:

46 lines | [app/config/services.yml](#)

... lines 1 - 8

```
9 services:
10
11    ... lines 10 - 40
41    AppBundle\Service\MessageManager:
42        arguments:
43            - ['You can do it!', 'Dude, sweet!', 'Woot!']
44            - ['We are *never* going to figure this out', 'Why even try again?', 'Facepalm']
45        public: true
```

Now that we have `MessageManager`, let's *remove* all the `MessageGenerator` stuff completely! Copy the old discouraging service id. Then, search for it:

```
$ git grep app.discouraging_message_generator
```

Ah, this is *only* used in `GenusAdminController`. In fact, both `MessageGenerator` services are *only* used there. Let's use the new `MessageManager` service - which I've *purposely* made public for now:

46 lines | [app/config/services.yml](#)

... lines 1 - 8

9 services:

... lines 10 - 40

41 AppBundle\Service\MessageManager:

42 arguments:

43 - ['You can do it!', 'Dude, sweet!', 'Woot!']

44 - ['We are \*never\* going to figure this out', 'Why even try again?', 'Facepalm']

45 public: true

In GenusAdminController, use that: `$this->get(MessageGenerator::class)->getEncouragingMessage()`:

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

... lines 1 - 16

17 class GenusAdminController extends Controller

18 {

... lines 19 - 64

65 public function editAction(Request \$request, Genus \$genus)

66 {

... lines 67 - 70

71 if (\$form->isSubmitted() && \$form->isValid()) {

... lines 72 - 77

78 \$this->addFlash(

79 'success',

80 \$this->get(MessageManager::class)->getEncouragingMessage()

81 );

... lines 82 - 90

91 }

... lines 92 - 95

96 }

97 }

Then the same below: `$this->get(MessageGenerator::class)->getDiscouragingMessage()`:

```

97 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
... lines 1 - 16
17 class GenusAdminController extends Controller
18 {
... lines 19 - 64
65     public function editAction(Request $request, Genus $genus)
66     {
... lines 67 - 70
71         if ($form->isSubmitted() && $form->isValid()) {
... lines 72 - 77
78             $this->addFlash(
79                 'success',
80                 $this->get(MessageManager::class)->getEncouragingMessage()
81             );
... lines 82 - 85
86         } elseif ($form->isSubmitted()) {
87             $this->addFlash(
88                 'error',
89                 $this->get(MessageManager::class)->getDiscouragingMessage()
90             );
91         }
... lines 92 - 95
96     }
97 }

```

Time to celebrate! Delete the discouraging service, and go remove the legacy alias too: neither is being used. This refactoring was optional, but it makes life a little bit easier: I can now safely type-hint any argument with `MessageManager` and let autowiring do its magic.

And, our app isn't broken, which is always nice. Oh, and I'll delete the unused `MessageGenerator` class.

Next, we'll try out a special new type of dependency injection for controllers and clean up the rest of our service config.



# Chapter 9: Autowiring Controller Arguments

Our app is now *fully* using the new config features. It's time to start enjoying it a little!

Let's start by cleaning up `legacy_aliases.yml`:

```
7 lines | app/config/legacy_aliases.yml
1  services:
2      app.markdown_transformer: '@AppBundle\Service\MarkdownTransformer'
3      app.markdown_extension: '@AppBundle\Twig\MarkdownExtension'
4      app.security.login_form_authenticator: '@AppBundle\Security\LoginFormAuthenticator'
5      app.doctrine.hash_password_listener: '@AppBundle\Doctrine\HashPasswordListener'
6      app.form.help_form_extension: '@AppBundle\Form\TypeExtension\HelpFormExtension'
```

Remember, we created this because these old service ids might still be used in our app. Let's eliminate these one-by-one.

Copy the first id and go see where it's used:

```
$ git grep app.markdown_transformer
```

## [Private Services and \\$container->get\(\)](#)

Ok! This is used in `GenusController`. Open that up - there it is!

```
143 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 14
15  class GenusController extends Controller
16  {
... lines 17 - 77
78      public function showAction(Genus $genus)
79      {
... lines 80 - 81
82          $markdownTransformer = $this->get('app.markdown_transformer');
... lines 83 - 95
96      }
... lines 97 - 141
142 }
```

Easy fix! Let's change this to use the new service id: `MarkdownTransformer::class`:

```

143 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 14
15 class GenusController extends Controller
16 {
... lines 17 - 77
78 public function showAction(Genus $genus)
79 {
... lines 80 - 81
82     $markdownTransformer = $this->get(MarkdownTransformer::class);
... lines 83 - 95
96 }
... lines 97 - 141
142 }

```

Awesome! Let's try it: navigate to /genus. That code was on the show page, so click any of the genres and... explosion!

You have requested a non-existent service AppBundle\Service\MarkdownTransformer.

But... we *know* that's a service: it's even *explicitly* configured:

```

46 lines | app/config/services.yml
... lines 1 - 8
9 services:
... lines 10 - 29
30 AppBundle\Service\MarkdownTransformer:
31     arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
... lines 32 - 46

```

What's going on!?

Remember, *all* of these services are *private*... which means that we *cannot* fetch them via `$container->get()`:

```

46 lines | app/config/services.yml
... lines 1 - 8
9 services:
10     _defaults:
... lines 11 - 12
13     public: false
... lines 14 - 46

```

This is actually one of the *big* motivations behind all of these autowiring and auto-registration changes: we do not want you to fetch things out of the container directly anymore. Nope, we want you to use dependency injection.

Not only is dependency injection a better practice than calling `$container->get()`, using it will actually give you better *errors*!

For example, if you use `$container->get()` and accidentally fetch a service that doesn't exist, you'll only get an error if you visit a page that runs that code. But if you use dependency injection and reference a service that doesn't exist, you'll get a huge error when you access *any* page or try to do *anything* in your app. If you make all services private, the new config system is actually more stable than the old one.

## Controller Action Injection

To fix our error, we need to use classic dependency injection. And actually, there are *two* ways to do this in a controller. First, we could of course, go to the top of our class, add a `__construct` function, type-hint a `MarkdownTransformer` argument set that on a property, and use it below. Thanks to autowiring, we wouldn't need to touch any config files.

But, because this is a bit tedious and so common to do in a controller, we've added a shortcut. In controllers *only*, you can autowire a service into an argument of your action method. We'll add `MarkdownTransformer $markdownTransformer`:

```

142 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 14
15 class GenusController extends Controller
16 {
... lines 17 - 77
78     public function showAction(Genus $genus, MarkdownTransformer $markdownTransformer)
79     {
... lines 80 - 94
95     }
... lines 96 - 140
141 }

```

Now, remove the `$this->get()` line... which is a shortcut for `$this->container->get()`.

This fixes things... because we've eliminated the `$container->get()` call that does *not* work with private services.

Celebrate by removing the first alias! The rest are easy!

```

6 lines | app/config/legacy\_aliases.yml
1 services:
2     app.markdown_extension: '@AppBundle\Twig\MarkdownExtension'
... lines 3 - 6

```

The `app.markdown_extension` id isn't referenced anywhere, so remove that. The `app.security.login_form_authenticator` is used in *two* places: `security.yml` and also `UserController`:

```

40 lines | app/config/security.yml
... lines 1 - 2
3 security:
... lines 4 - 14
15     firewalls:
... lines 16 - 20
21         main:
... line 22
23         guard:
24             authenticators:
25                 - app.security.login_form_authenticator
... lines 26 - 40

```

```

81 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 11
12 class UserController extends Controller
13 {
... lines 14 - 16
17     public function registerAction(Request $request)
18     {
... lines 19 - 21
22         if ($form->isValid()) {
... lines 23 - 30
31             return $this->get('security.authentication.guard_handler')
32                 ->authenticateUserAndHandleSuccess(
... lines 33 - 34
35                 $this->get('app.security.login_form_authenticator'),
... line 36
37             );
38         }
... lines 39 - 42
43     }
... lines 44 - 79
80 }

```

Copy the new service id - the class name. In security.yml, just replace the old with the new:

```

40 lines | app/config/security.yml
... lines 1 - 2
3 security:
... lines 4 - 14
15     firewalls:
... lines 16 - 20
21         main:
... line 22
23             guard:
24                 authenticators:
25                     - AppBundle\Security\LoginFormAuthenticator
... lines 26 - 40

```

Next, in UserController, I'll search for "authenticator". Ah, we're fetching it out of the container directly! We know the fix: type-hint a new argument with LoginFormAuthenticator \$authenticator and use that below:

```

82 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 7
8  use AppBundle\Security\LoginFormAuthenticator;
... lines 9 - 12
13 class UserController extends Controller
14 {
... lines 15 - 17
18     public function registerAction(Request $request, LoginFormAuthenticator $authenticator)
19     {
... lines 20 - 22
23         if ($form->isValid()) {
... lines 24 - 31
32             return $this->get('security.authentication.guard_handler')
33                 ->authenticateUserAndHandleSuccess(
... lines 34 - 35
36                 $authenticator,
... line 37
38             );
39         }
... lines 40 - 43
44     }
... lines 45 - 80
81 }

```

Almost done! The `app.doctrine.hash_password_listener` service isn't being used anywhere, and neither is `app.form.help_form_extension`.

And... that's it! At the top of `services.yml`, remove the import:

```

42 lines | app/config/services.yml
1  # Learn more about services, parameters and containers at
2  # http://symfony.com/doc/current/book/service_container.html
3  parameters:
4  #   parameter_name: value
5
6  services:
... lines 7 - 42

```

Then, delete `legacy_aliases.yml`.

Our *last* public service is `MessageManager`:

```

46 lines | app/config/services.yml
... lines 1 - 8
9  services:
... lines 10 - 40
41  AppBundle\Service\MessageManager:
42      arguments:
43      - ['You can do it!', 'Dude, sweet!', 'Woot!']
44      - ['We are *never* going to figure this out', 'Why even try again?', 'Facepalm']
45  public: true

```

Now we know how to fix this. In `GenusAdminController`, find `editAction()` and add an argument: `MessageManager $messageManager`:

```

97 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
... lines 1 - 6
7 use AppBundle\Service\MessageManager;
... lines 8 - 16
17 class GenusAdminController extends Controller
18 {
... lines 19 - 64
65 public function editAction(Request $request, Genus $genus, MessageManager $messageManager)
66 {
... lines 67 - 95
96 }
97 }

```

Use that below in both places:

```

97 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
... lines 1 - 6
7 use AppBundle\Service\MessageManager;
... lines 8 - 16
17 class GenusAdminController extends Controller
18 {
... lines 19 - 64
65 public function editAction(Request $request, Genus $genus, MessageManager $messageManager)
66 {
... lines 67 - 70
71 if ($form->isSubmitted() && $form->isValid()) {
... lines 72 - 77
78 $this->addFlash(
79     'success',
80     $messageManager->getEncouragingMessage()
81 );
... lines 82 - 85
86 } elseif ($form->isSubmitted()) {
87     $this->addFlash(
88         'error',
89         $messageManager->getDiscouragingMessage()
90     );
91 }
... lines 92 - 95
96 }
97 }

```

Back in services.yml, make that service private:

```

42 lines | app/config/services.yml
... lines 1 - 5
6 services:
... lines 7 - 37
38 AppBundle\Service\MessageManager:
39     arguments:
40         - ['You can do it!', 'Dude, sweet!', 'Woot!']
41         - ['We are *never* going to figure this out', 'Why even try again?', 'Facepalm']

```

This is reason to celebrate! *All* our services are now private! We're using proper dependency injection on everything! Thanks to this, the container will optimize itself for performance *and* give us clear errors if we make a mistake... anywhere.

Next, we need to talk more about aliases: the key to unlocking the full potential of autowiring.

# Chapter 10: Aliases & When Autowiring Fails

Let's talk more about what happens when autowiring goes wrong. Right now, the `MarkdownTransformer` class has two arguments, type-hinted with `MarkdownParserInterface` and `Cache` from Doctrine:

```
35 lines | src/AppBundle/Service/MarkdownTransformer.php
... lines 1 - 4
5 use Doctrine\Common\Cache\Cache;
6 use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
7
8 class MarkdownTransformer
9 {
... lines 10 - 12
13     public function __construct(MarkdownParserInterface $markdownParser, Cache $cache)
14     {
... lines 15 - 16
17     }
... lines 18 - 33
34 }
```

But, these are *not* being autowired: we're explicitly specifying each argument:

```
42 lines | app/config/services.yml
... lines 1 - 5
6 services:
... lines 7 - 26
27     AppBundle\Service\MarkdownTransformer:
28         arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
... lines 29 - 42
```

## [Letting Symfony tell you about Autowiring Failures](#)

If I were creating this service today, I actually wouldn't specify *any* configuration at first. Nope, I'd just create the class, add the type-hints, and let Symfony tell me if it had any problems autowiring my arguments.

To show off one of the ways autowiring can fail, I'm going to remove the first argument and set it to an empty string:

```
41 lines | app/config/services.yml
... lines 1 - 5
6 services:
... lines 7 - 26
27     AppBundle\Service\MarkdownTransformer:
28         arguments: ['', '@doctrine_cache.providers.my_markdown_cache']
... lines 29 - 41
```

When you do this, it means that you *do* want Symfony to try to autowire it. Symfony will try to autowire the first argument, but not the second. Actually, if this looks ugly to you, I agree! In a few minutes, I'll show you a cleaner way of explicitly wiring only *some* arguments.

Anyways, let's see if the `MarkdownParserInterface` type-hint can be autowired. Refresh the page... or *any* page.

Explosion!

Cannot autowire service AppBundle\Service\MarkdownTransformer: argument \$markdownParser.



It's very clearly saying that there is a problem with *this* specific argument.

The error continues:

... it references interface MarkdownParserInterface but no such service exists.

This is because it's looking for a service with *exactly* this id. But, none was found! So, it tries to help out:

You should maybe alias this interface to one of these existing services.

and it lists one, two, three, four, *five* services in the container that implement MarkdownParserInterface. This is autowiring the Symfony way: there's no guess work.

## Using Aliases to add Valid Autowiring Types

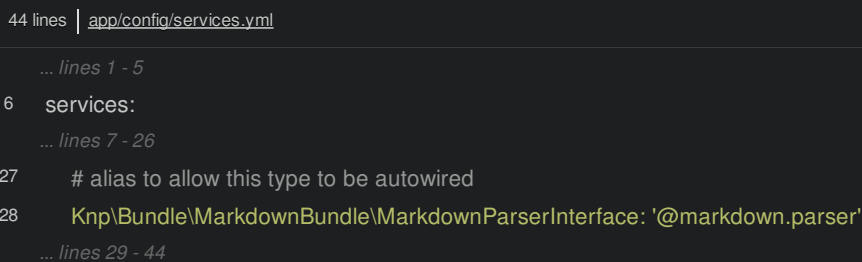
Previously, we were using a service called markdown.parser. Find your terminal and get some information about this service:



```
$ php bin/console debug:container markdown.parser
```

Interesting! This is actually an alias to markdown.parser.max. This service comes from KnpMarkdownBundle, and it ships with a few different markdown parsers. It then creates an alias from markdown.parser to whatever parser we configured as the default.

So to fix the error, we have two options. First, we could of course explicitly specify the argument, just like we were doing before. Or, as the error suggests, we can create an *alias*. Let's do that: alias Knp\Bundle\MarkdownBundle\MarkdownParserInterface to @markdown.parser:



```
44 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 26
27  # alias to allow this type to be autowired
28  Knp\Bundle\MarkdownBundle\MarkdownParserInterface: '@markdown.parser'
... lines 29 - 44
```

We just told Symfony *exactly* what service to autowire when it sees the MarkdownParserInterface type-hint. In theory, KnpMarkdownBundle would come with this alias already, and it probably will in the future.

Try the page! It works!

# Chapter 11: Autowiring Deprecations

On the web debug toolbar, I've got a little yellow icon that says 10 deprecation warnings. Rude! Let's click that!

These are all the ways that my code is using old, deprecated, uncool functionality. It's basically a list of stuff we need to update before upgrading to Symfony 4. And there are a few deprecations related to autowiring:

Autowiring services based on the types they implement is deprecated since Symfony 3.3 and won't be supported in version 4.0. You should rename or alias `security.user_password_encoder.generic` to ... long class name... `UserPasswordEncoder` instead.

## Tip

The text in the deprecation may look slightly different for you: we updated it to be a bit more clear in Symfony 3.3.1.

Um... what????

This is saying that *somewhere*, we are type-hinting an argument with `Symfony\Component\Security\Core\Encoder\UserPasswordEncoder...` but there is *no* service in the container with that exact *id*. So, autowiring got busy: it looked at *every* service and found exactly *one* - `security.user_password_encoder.generic` - that has this class. It passed *this* service to that argument.

And *that* is the part of autowiring that is deprecated. Looking across every service for a matching class or interface was a little more magic than we wanted in Symfony.

How do we fix this? Actually, there are *two* solutions!

## [Solution 1\) Fixing the Type-Hint](#)

Here's the first question: is there a different type-hint that we should be using instead for this service? Let's find out!

Head to your terminal. We already know about the `debug:container` command:

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

This gives us a big list of every *public* service in the container. The blue text is the *id* of each service. But guess what? Service id's are *much* less important in Symfony 3.3... because we almost always rely on type-hints and autowiring.

Re-run the command again with a new `--types` option:

```
$ php bin/console debug:container --types
```

Voilà! This is a list of all valid type-hints that you can use for autowiring. This is *awesome*. If you search for "encoder", you'll find one called `UserPasswordEncoderInterface`. *This* is the type-hint we should use! Symfony ships with this alias to enable autowiring.

Cool! Let's find out where we're using this:

```
$ git grep UserPasswordEncoder
```

Two places: `HashPasswordListener` and `LoginFormAuthenticator`. Open up `HashPasswordListener`. Then, add `Interface` to the end of the use statement, and also the type-hint:

```

65 lines | src/AppBundle/Doctrine/HashPasswordListener.php
... lines 1 - 7
8  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
9
10 class HashPasswordListener implements EventSubscriber
11 {
... lines 12 - 13
14     public function __construct(UserPasswordEncoderInterface $passwordEncoder)
15     {
... line 16
17     }
... lines 18 - 63
64 }

```

That's it.

Open up LoginFormAuthenticator and do the exact same thing: update the use... and the argument:

```

80 lines | src/AppBundle/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
... lines 12 - 15
16 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
17 {
... lines 18 - 22
23     public function __construct(FormFactoryInterface $formFactory, EntityManager $em, RouterInterface $router, UserPasswordEncoderInterface $passwordEncoder)
24     {
... lines 25 - 28
29     }
... lines 30 - 78
79 }

```

Ok, go back to the browser! Refresh, and watch those 10 deprecations. Bam! 8 deprecations!

If you check the list now, we still have *one* more autowiring deprecation. This time, apparently, it's unhappy about an EntityManager type-hint.

Same question as before: is there a better type-hint to use? Let's find out:

```

$ php bin/console debug:container --types

```

Search for EntityManager and... boom! There is an EntityManagerInterface alias. *This* is the officially supported type-hint.

## [Solution 2: Adding an Alias](#)

Ok, we know the fix: update our EntityManager type hints to EntityManagerInterface! But... there's another solution! If you want, it is *totally* ok to type-hint EntityManager. To make this work with autowiring, we can create an *alias*.

Copy the Doctrine\ORM\EntityManager class name. Then, find your editor and open up services.yml. Add the alias: Doctrine\ORM\EntityManager aliased to @, and then copy the target service id: @doctrine.orm.default\_entity\_manager:

```
45 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 26
27  # alias to allow this type to be autowired
... line 28
29  Doctrine\ORM\EntityManager: '@doctrine.orm.default_entity_manager'
... lines 30 - 45
```

We have *full* control over autowiring. With aliases, we can configure *exactly* which service we want to use for each type-hint. No magic.

Ok, refresh the page one more time! Got it! 8 deprecations now down to 7. The rest of the deprecations are related to other parts of our code. I'll leave those as homework.

# Chapter 12: Configuring Specific (Named) Arguments

We saw earlier that sometimes you only *need* to pass *one* argument... and the rest can be autowired. For MarkdownTransformer, we *only* need to configure the *second* argument:

```
45 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 30
31  AppBundle\Service\MarkdownTransformer:
32    arguments: ['@doctrine_cache.providers.my_markdown_cache']
... lines 33 - 45
```

To allow the first to be continue to be autowired, we set it to an empty string. That works... but it's just weird. So, in Symfony 3.3, there's a better way.

In MarkdownTransformer, the argument *names* are \$markdownParser and \$cache:

```
35 lines | src/AppBundle/Service/MarkdownTransformer.php
... lines 1 - 7
8  class MarkdownTransformer
9  {
... lines 10 - 12
13  public function __construct(MarkdownParserInterface $markdownParser, Cache $cache)
14  {
... lines 15 - 16
17  }
... lines 18 - 33
34 }
```

The \$cache argument is the one we need to configure. Back in services.yml, copy the cache service id, clear out the arguments, and add \$cache: '@doctrine\_cache.providers.my\_markdown\_cache':

```
46 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 30
31  AppBundle\Service\MarkdownTransformer:
32    arguments:
33    $cache: '@doctrine_cache.providers.my_markdown_cache'
... lines 34 - 46
```

The dollar sign is the important part: it tells Symfony that we're actually configuring an argument by its *name*. \$cache here must match \$cache here.

And it works beautifully - refresh now! Everything is happy!

## Named Arguments are Validated

But wait! Isn't this dangerous!? I mean, normally, if I were coding in MarkdownTransformer, I could safely rename this variable to \$cacheDriver:

```

35 lines | src/AppBundle/Service/MarkdownTransformer.php
... lines 1 - 7
8  class MarkdownTransformer
9  {
... lines 10 - 12
13  public function __construct(MarkdownParserInterface $markdownParser, Cache $cacheDriver)
14  {
... line 15
16      $this->cache = $cacheDriver;
17  }
... lines 18 - 33
34 }

```

That change shouldn't make any difference to any code outside of this class.

But suddenly now... it *does* make a difference! The `$cache` in `services.yml` no longer matches the argument name. Isn't this a huge problem!? Actually, no. Yes, you *will* get an error - you can see it when you refresh. But this error will happen if you try to refresh *any* page across your entire system. Symfony validates *all* of your services and configuration. And as soon as it saw `$cache` in `services.yml` with no corresponding argument, it screamed.

So yes, changing the argument in your service class *will* break your app. But the error is unignorable: nothing works until you fix it. Update the configuration to `$cacheDriver`:

```

46 lines | app/config/services.yml
... lines 1 - 5
6  services:
... lines 7 - 30
31  AppBundle\Service\MarkdownTransformer:
32      arguments:
33          $cacheDriver: '@doctrine_cache.providers.my_markdown_cache'
... lines 34 - 46

```

And refresh again.

Back working! This is what makes Symfony's autowiring special. If there is *any* question or problem wiring the arguments to *any* service, Symfony throws an exception at *compile* time... meaning, it throws an exception when you try to refresh *any* page. This means no surprises: it's *not* possible to have an autowiring error on only *one* page and not notice it.

# Chapter 13: RAD with Symfony 3.3

We've done a *lot* of work, and I showed you the *ugliest* parts of the new system so that you can solve them in your project. That's cool... but so far, coding hasn't been much fun!

And that's a shame! Once you're done upgrading, using the new configuration system is a *blast*. Let's take it for a legit test drive.

## [Creating an Event Subscriber](#)

Here's the goal: create an event listener that adds a header to every response. Step 1: create an EventSubscriber directory - though this could live anywhere - and a file inside called AddNiceHeaderEventSubscriber:

```
24 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 2
3  namespace AppBundle\EventSubscriber;
... lines 4 - 8
9  class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
... lines 11 - 22
23 }
```

Event subscribers always look the same: they must implement EventSubscriberInterface:

```
24 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 2
3  namespace AppBundle\EventSubscriber;
4
5  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
... lines 6 - 8
9  class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
... lines 11 - 22
23 }
```

I'll go to the Code-Generate menu, or Command+N on a Mac - and select "Implement Methods" to add the one required function: public static getSubscribedEvents():

```
24 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 2
3  namespace AppBundle\EventSubscriber;
4
5  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
... lines 6 - 8
9  class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
... lines 11 - 16
17     public static function getSubscribedEvents()
18     {
... lines 19 - 21
22     }
23 }
```

To listen to the kernel.response event, return KernelEvents::RESPONSE set to onKernelResponse:

```

24 lines | src/AppBundle\EventSubscriber\AddNiceHeaderEventSubscriber.php
... lines 1 - 2
3 namespace AppBundle\EventSubscriber;
... lines 4 - 6
7 use Symfony\Component\HttpKernel\KernelEvents;
8
9 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
... lines 11 - 16
17     public static function getSubscribedEvents()
18     {
19         return [
20             KernelEvents::RESPONSE => 'onKernelResponse'
21         ];
22     }
23 }

```

On top, create that method: `onKernelResponse()` with a `FilterResponseEvent` object: that's the argument passed to listeners of *this* event:

```

24 lines | src/AppBundle\EventSubscriber\AddNiceHeaderEventSubscriber.php
... lines 1 - 5
6 use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
... lines 7 - 8
9 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
11     public function onKernelResponse(FilterResponseEvent $event)
12     {
... lines 13 - 14
15     }
... lines 16 - 22
23 }

```

Inside, add a header: `$event->getResponse()->headers->set()` with `X-NICE-MESSAGE` set to `That was a great request`:

```

24 lines | src/AppBundle\EventSubscriber\AddNiceHeaderEventSubscriber.php
... lines 1 - 5
6 use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
... lines 7 - 8
9 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
10 {
11     public function onKernelResponse(FilterResponseEvent $event)
12     {
13         $event->getResponse()
14             ->headers->set('X-NICE-MESSAGE', 'That was a great request!');
15     }
... lines 16 - 22
23 }

```

Ok, we've touched only *one* file and written 23 lines of code. And... we're done! Yep, this will *already* work. I'll open up my network tools, then refresh one more time. For the top request, click "Headers", scroll down and... there it is! We just added an event subscriber to Symfony... by *just* creating the event subscriber. Yea... it kinda makes sense.

The new class was automatically registered as a service and automatically tagged thanks to autoconfigure.

[Grab some Dependencies](#)



But what if we want to log something from inside the subscriber? What type-hint should we use for the logger? Let's find out! Find your terminal and run:

```
$ php bin/console debug:container --types
```

And search for "logger". Woh, nothing!? So, there is no way to type-hint the logger for autowiring!?

Actually... since the autowiring stuff is new, some bundles are still catching up and adding aliases for their interfaces. An alias *has* been added to MonologBundle... but only in version 3.1. In composer.json, I'll change its version to ^3.1:

```
68 lines | composer.json
1  {
    ... lines 2 - 15
16  "require": {
    ... lines 17 - 22
23  "symfony/monolog-bundle": "^3.1",
    ... lines 24 - 30
31  },
    ... lines 32 - 66
67 }
```

Then, run:

```
$ composer update
```

to pull down the latest changes.

If you have problems with other bundles that don't have aliases yet... don't panic! You can always add the alias yourself. In this case, the type-hint should be Psr\Log\LoggerInterface, which you could alias to @logger:

```
services:
# ...
Psr\Log\LoggerInterface: '@logger'
```

You always have full control over how things are autowired.

Ok, done! Let's look at the types list again:

```
$ php bin/console debug:container --types
```

There it is! Psr\Log\LoggerInterface.

Back in the code, add public function \_\_construct() with a LoggerInterface \$logger argument:

```

34 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 4
5 use Psr\Log\LoggerInterface;
... lines 6 - 9
10 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
11 {
... lines 12 - 13
14     public function __construct(LoggerInterface $logger)
15     {
... line 16
17     }
... lines 18 - 32
33 }

```

I'll hit Option+Enter and initialize my field:

```

34 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 4
5 use Psr\Log\LoggerInterface;
... lines 6 - 9
10 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
11 {
12     private $logger;
13
14     public function __construct(LoggerInterface $logger)
15     {
16         $this->logger = $logger;
17     }
... lines 18 - 32
33 }

```

That's just a shortcut to add the property and set it.

In the main method, use the logger: `$this->logger->info('Adding a nice header')`:

```

34 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 9
10 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
11 {
... lines 12 - 18
19     public function onKernelResponse(FilterResponseEvent $event)
20     {
21         $this->logger->info('Adding a nice header!');
... lines 22 - 24
25     }
... lines 26 - 32
33 }

```

Other than the one-time composer issue, we've *still* only touched *one* file. Find your browser and refresh. I'll click one of the web debug toolbar links at the bottom and then go to "Logs". There it is! Autowiring passes us the logger without any configuration.

## [Adding more Arguments](#)

Let's keep going! Instead of hard-coding the message, let's use our MessageManager. Add it as a second argument, then create the property and set it like normal:

```

39 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 4
5 use AppBundle\Service\MessageManager;
... lines 6 - 10
11 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
12 {
... line 13
14     private $messageManager;
15
16     public function __construct(LoggerInterface $logger, MessageManager $messageManager)
17     {
... line 18
19         $this->messageManager = $messageManager;
20     }
... lines 21 - 37
38 }

```

In the method, add `$message = $this->messageManager->getEncouragingMessage();`. Use that below:

```

39 lines | src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php
... lines 1 - 10
11 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
12 {
... lines 13 - 21
22     public function onKernelResponse(FilterResponseEvent $event)
23     {
... lines 24 - 25
26         $message = $this->messageManager->getEncouragingMessage();
27
28         $event->getResponse()
29             ->headers->set('X-NICE-MESSAGE', $message);
30     }
... lines 31 - 37
38 }

```

Once again, autowiring will work with zero configuration. The MessageManager service id is equal to its class name... so autowiring works immediately:

```

46 lines | app/config/services.yml
... lines 1 - 5
6 services:
... lines 7 - 41
42     AppBundle\Service\MessageManager:
43         arguments:
44             - ['You can do it!', 'Dude, sweet!', 'Woot!']
45             - ['We are *never* going to figure this out', 'Why even try again?', 'Facepalm']

```

Refresh to try it! Click the logs icon, go to "Request / Response", then the "Response" tab. Yea! This is another way to see our response header.

Say hello to the new workflow: focus on your business logic and ignore configuration. *If you do* need to configure something, let Symfony tell you.

## [Manually Wiring when Necessary](#)

Let's see an example of that: add a third argument: `$showDiscouragingMessage`. I'll use Alt+Enter again to set this on a new property:

43 lines | [src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php](#)

```
... lines 1 - 10
11 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
12 {
... lines 13 - 14
15     private $showDiscouragingMessage;
16
17     public function __construct(LoggerInterface $logger, MessageManager $messageManager, $showDiscouragingMessage)
18     {
... lines 19 - 20
21         $this->showDiscouragingMessage = $showDiscouragingMessage;
22     }
... lines 23 - 41
42 }
```

This argument is *not* an object: it's a boolean. And that means that autowiring *cannot* guess what to put here.

But... ignore that! In `onKernelResponse()`, add some logic: if `$this->showDiscouragingMessage`, then call `getDiscouragingMessage()`. Else, call `getEncouragingMessage()`:

43 lines | [src/AppBundle/EventSubscriber/AddNiceHeaderEventSubscriber.php](#)

```
... lines 1 - 10
11 class AddNiceHeaderEventSubscriber implements EventSubscriberInterface
12 {
... lines 13 - 23
24     public function onKernelResponse(FilterResponseEvent $event)
25     {
... lines 26 - 27
28         $message = $this->showDiscouragingMessage
29             ? $this->messageManager->getDiscouragingMessage()
30             : $this->messageManager->getEncouragingMessage();
... lines 31 - 33
34     }
... lines 35 - 41
42 }
```

Just like before, we're focusing *only* on this class, not configuration. And this class is done! So, let's try it! Error!

Cannot autowire service AddNiceHeaderEventSubscriber: argument \$showDiscouragingMessage of method `__construct()` must have a type-hint or be given a value explicitly.

Yes! Symfony can automate *most* configuration. And as soon as it *can't*, it will tell you *what* you need to do.

Copy the class name, then open `services.yml`. To explicitly configure this service, paste the class name and add arguments. We *only* need to specify `$showDiscouragingMessage`. So, add `$showDiscouragingMessage: true`:

50 lines | [app/config/services.yml](#)

```
... lines 1 - 5
6 services:
... lines 7 - 46
47     AppBundle\EventSubscriber\AddNiceHeaderEventSubscriber:
48         arguments:
49             $showDiscouragingMessage: true
```

Refresh now! The error is gone! And in the profiler... yep! The message is much more discouraging. Boooo.

Ok guys that is it! Behind the scenes, the way that you configure services is still the same: Symfony *still* needs to know the class name and arguments of every service. But before Symfony 3.3, all of this *had* to be done explicitly: you needed to register every single service and specify every argument and tag. But if you use the new features, *a lot* of this is automated.

Instead of filling in *everything*, only configure what you need.

And there's *another* benefit to the new stuff. Now that our services are *private* - meaning, we no longer use `$container->get()` - Symfony will give us more immediate errors *and* will automatically optimize itself. Cool!

Your turn to go play! I hope you love this new stuff: faster development without a ton of WTF moments! And let me know what you think!

All right guys, see you next time.

