Symfony 5 Fundamentals: Services, Config & Environments

With <3 from SymfonyCasts

Chapter 1: Bundles!

Hey friends! Welcome to Symfony 5 Fundamentals! I *cannot* emphasize enough, how *important* this course is to make you *super* productive in Symfony. And, well, I *also* think you're going to love it. Because we're going to *really* explore how your app works: services, configuration, environment, environment variables and more! These will be the tools that you'll need for *everything* else that you'll do in Symfony. After putting some work in now, anything else you build will feel *much* easier.

Ok! Let's go unlock some potential! The *best* way to do that - of course - is to download the course code from this page and code along with me. If you followed our first course, you rock! The code is basically where that course finished. But I recommend downloading the *new* code because I *did* make a few small tweaks.

After you unzip the download, you'll find a start/ directory with the same code that you see here. Head down to the fancy README.md for all the instructions on how you get your project set up... *and*, of course, a poem about magic.

The last step in the setup will be to find a terminal, move into the project and use the symfony executable to start a handy development web server. If you don't have this symfony binary, you can download it at https://symfony.com/download. I'll run:

```
symfony serve -d
```

to start a web server at localhost:8000. The -d means run as a "daemon" - a fancy way of saying that this runs in the background and I can keep using my terminal. You can run symfony server:stop later to stop it.

Ok! Spin over to your browser and go to https://localhost:8000 to see... Cauldron Overflow! Our question & answer site dedicated to Witches and Wizards. It's a *totally* untapped market.

Services do Everything

One of the things we learned at the end of the first course is that all the work in a Symfony app - like rendering a template, logging something, executing database queries, making API calls - *everything* is done by one of many useful objects floating around. We call these objects *services*. There's a router service, logger service, service for rendering Twig templates and many more. Simply put, a service is a fancy word for an object that does work.

And because services do work, they're *tools*! If you know how to get access to these objects, then you're very powerful. How *do* we access them? The primary way is by something called autowiring. Open up src/Controller/QuestionController.php and find the homepage() method:

44 lines src/Controller/QuestionController.php

```
... lines 1 - 9
class QuestionController extends AbstractController
{
/**
* @Route("/", name="app_homepage")
*/
public function homepage(Environment $twigEnvironment)
{
/*
// fun example of using the Twig service directly!
$html = $twigEnvironment->render('question/homepage.html.twig');
return new Response($html);
*/
return $this->render('question/homepage.html.twig');
}
... lines 26 - 42
}
```

We commented out the code that used it, but by adding an argument type-hinted with Environment, we signaled to Symfony that we wanted it to pass us the Twig service object:

44 lines src/Controller/QuestionController.php

```
... lines 1 - 7
use Twig\Environment;
class QuestionController extends AbstractController
{
... lines 12 - 14
public function homepage(Environment $twigEnvironment)
{
... lines 17 - 24
}
... lines 26 - 42
}
```

That's called autowiring.

And how did we know to use this exact Environment type hint to get the Twig service? And what *other* service objects are floating around waiting for us to use them and claim ultimate programming glory? Find your terminal and run:

```
php bin/console debug:autowiring
```

Boom! *This* is our guide. Near the bottom, it says that if you type-hint a controller argument with Twig\Environment, it will give us the Twig service. Another one is CacheInterface: use that type-hint to get a useful caching object. This is your *menu* of what service objects are available and what type-hint to use in a controller argument to get them.

Hello Bundles!

But where do these services come from? Like, who added these to the system? The answer to that is... *bundles*. Back in your editor, open a new file: config/bundles.php:

13 lines config/bundles.php

```
... lines 1 - 2
return [
Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true, 'test' => true],
Symfony\WebpackEncoreBundle\WebpackEncoreBundle::class => ['all' => true],
];
```

We'll see who uses this file later, but it returns an array with 8 class names that all have the word "Bundle" in them.

Ok, first, whenever you install one of these "bundle" things, the Flex recipe system automatically updates this file *for* you and adds the new bundle. For example, in the first course, when we installed this WebpackEncoreBundle, its recipe added this line:

13 lines config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 10
Symfony\WebpackEncoreBundle\WebpackEncoreBundle::class => ['all' => true],
];
```

The point is, this is *not* a file that you *normally* need to think about.

But... what *is* a bundle? Very simply: bundles are Symfony *plugins*. They're PHP libraries with special integration with Symfony.

And, the *main* reason that you add a bundle to your app is because bundles give you services! In fact, *every* single service that you see in the debug:autowiring list comes from one of these eight bundles. You can kind of guess that the Twig\Environment service down here comes from TwigBundle:

13 lines config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 5
Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
... lines 7 - 11
];
```

So if I removed that TwigBundle line and ran the command again, the Twig service would be gone.

And yes, bundles can give you other things like routes, controllers, translations and more. But the *main* point of a bundle is that it gives you more services, more *tools*.

Need a new tool in your app to... talk to an API or parse Markdown into HTML? If you can find a bundle that does that, you get that tool for free.

In fact, let's do *exactly* that next.

Chapter 2: KnpMarkdownBundle & Service

Fun fact! Witches & wizards *love* writing markdown. I have no idea why... but darnit! We're going to give the people what they want! We're going to allow the question text to be written in Markdown. For now, we'll focus on this "show" page.

Open up QuestionController and find the show() method:

44 lines src/Controller/QuestionController.php

```
... lines 1 - 9
class QuestionController extends AbstractController
{
... lines 12 - 26
/**
* @Route("/questions/{slug}", name="app_question_show")
*/
public function show($slug)
{
$answers = [
'Make sure your cat is sitting purrrfectly still  ',
'Honestly, I like furry shoes better than MY cat',
'Maybe... try saying the spell backwards?',
];
return $this->render('question/show.html.twig', [
'question' => ucwords(str_replace('-', ' ', $slug)),
'answers' => $answers,
]);
}
}
```

Let's see, this renders show.html.twig... open up that template... and find the question text. Here it is:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
<div class="row">
<div class="col-12">
... line 9
<div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
<div class="q-container-show p-4">
<div class="row">
... lines 13 - 15
<div class="col">
... line 17
<div class="q-display p-3">
... line 19
<p class="d-inline">I've been turned into a cat, any thoughts on how to turn back? While I'm adorable, I don't really care for cat food.</p>
... line 21
</div>
</div>
</div>
</div>
</div>
</div>
</div>
... lines 29 - 56
</div>
{% endblock %}
```

Because we don't have a database yet, the question is hardcoded. Let's move this text into our controller, so we can write some code to transform it from Markdown to HTML.

Copy the question text, delete it, and, in the controller, make a new variable: $questionText = and paste. Pass this to the template as a new questionText variable:

46 lines src/Controller/QuestionController.php

```
... lines 1 - 9
class QuestionController extends AbstractController
{
... lines 12 - 29
public function show($slug)
{
... lines 32 - 36
$questionText = 'I\'ve been turned into a cat, any thoughts on how to turn back? While I\'m adorable, I don\'t really care for cat food.';
return $this->render('question/show.html.twig', [
... line 40
'questionText' => $questionText,
... line 42
]);
}
}
```

Back in show.html.twig, print that: {{ questionText }}:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
<div class="row">
<div class="col-12">
... line 9
<div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
<div class="q-container-show p-4">
<div class="row">
... lines 13 - 15
<div class="col">
... line 17
<div class="q-display p-3">
... line 19
<p class="d-inline">{{ questionText }}</p>
... line 21
</div>
</div>
</div>
</div>
</div>
</div>
</div>
... lines 29 - 56
</div>
{% endblock %}
```

Oh, and to make things a bit more interesting, let's add some markdown formatting - how about ** around "adorable":

46 lines src/Controller/QuestionController.php

```
... lines 1 - 9
class QuestionController extends AbstractController
{
... lines 12 - 29
public function show($slug)
{
... lines 32 - 36
$questionText = 'I\'ve been turned into a cat, any thoughts on how to turn back? While I\'m **adorable**, I don\'t really care for cat food.';
... lines 38 - 43
}
}
```

Perfect!

If we refresh the page now... no surprise - it literally prints **adorable**.

Transforming text from Markdown into HTML is clearly "work"... and we know that all work in Symfony is done by a service. And... who knows? Maybe Symfony *already* has a service that parses markdown. At your terminal, let's find out. Run:

```
php bin/console debug:autowiring markdown
```

Installing KnpMarkdownBundle

Nope! And that makes sense: Symfony starts small but makes it super easy to add more stuff. Since I don't want to write a markdown parser by hand - that would be *crazy* - let's find something that can help! Google for KnpMarkdownBundle and find its GitHub page. This isn't the only bundle that can parse markdown, but it's a good one. My *hope* is that it will add a service to our app that can handle all the markdown parsing for us.

Copy the Composer require line, find your terminal and paste:

composer require knplabs/knp-markdown-bundle

This installs and... it configured a recipe! Run:

git status

It updated the files we expect: composer.json, composer.lock and symfony.lock but it *also* updated config/bundles.php! Check it out: we have a new line at the bottom that initializes the new bundle:

14 lines config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 11
Knp\Bundle\MarkdownBundle\KnpMarkdownBundle::class => ['all' => true],
];
```

Finding the new Service

Ok, so if the *main* purpose of a bundle is to give us more services... then we *probably* have at least one new one! Find your terminal and run debug:autowiring markdown again:

php bin/console debug:autowiring markdown

Yes! There are *two* services. Well actually, both of these interfaces are a way to get the *same* service object. See this little blue text - markdown.parser.max? We'll talk more about this later, but each "service" in Symfony has a unique "id". This service's unique id is apparently markdown.parser.max and we can *get* that service by using either type-hint.

It doesn't really matter which one we use, but if you check back on the bundle's documentation... they use MarkdownParserInterface.

Let's do it! In QuestionController::show() add a second argument: MarkdownParserInterface $markdownParser:

48 lines src/Controller/QuestionController.php

```
... lines 1 - 4
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
... lines 6 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownParserInterface $markdownParser)
{
... lines 33 - 45
}
}
```

Down below, let's say $parsedQuestionText = $markdownParser->... I love this: we don't even need to look at documentation to see what methods this object has. Thanks to the type-hint, PhpStorm tells us *exactly* what's available. Use transformMarkdown($questionText). Now, pass *this* variable into the template:

48 lines src/Controller/QuestionController.php

```
... lines 1 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownParserInterface $markdownParser)
{
... lines 33 - 37
$questionText = 'I\'ve been turned into a cat, any thoughts on how to turn back? While I\'m **adorable**, I don\'t really care for cat food.';
$parsedQuestionText = $markdownParser->transformMarkdown($questionText);
return $this->render('question/show.html.twig', [
... line 42
'questionText' => $parsedQuestionText,
... line 44
]);
}
}
```

Twig Output Escaping: The "raw" Filter

Love it! Will it work? Who knows? Move over and refresh. It... sorta works! But it's dumping out the HTML tags! The reason... is *awesome*. If you inspect the HTML... here we go... Twig is using htmlentities to output escape the text. Twig does that automatically for security: it protects against XSS attacks - that's when users try to enter JavaScript inside a question so that it will render & execute on your site. In this case, we *do* want to allow HTML because it's coming from our Markdown process. To tell Twig to *not* escape, we can use a special filter |raw:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
<div class="row">
<div class="col-12">
... line 9
<div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
<div class="q-container-show p-4">
<div class="row">
... lines 13 - 15
<div class="col">
... line 17
<div class="q-display p-3">
... line 19
<p class="d-inline">{{ questionText|raw }}</p>
... line 21
</div>
</div>
</div>
</div>
</div>
</div>
</div>
... lines 29 - 56
</div>
{% endblock %}
```

By the way, in a real app, because the question text *will* be entered by users we don't trust, we *would* need to do a bit more work to prevent XSS attacks. I'll mention how in a minute.

Anyways, now when we refresh... it works! It's subtle, but that word is now bold.

The twig:debug Command

By the way, you can *of course* read the Twig documentation to learn that this raw filter exists. But Symfony *also* has a command that will tell you *everything* Twig can do. At your terminal, run:

```
php bin/console debug:twig
```

How cool is that? This shows us the Twig "tests", filters, functions - everything Twig can do in our app. Here's the raw filter.

The markdown Twig Filter

And... oh! Apparently there's a filter called markdown! If you go back to the bundle's documentation and search for |markdown... yeah! So, in addition to the MarkdownParserInterface service, this bundle *also* apparently gave us another service that added this markdown filter. At the end of the tutorial, we'll even learn how to add our *own* custom filters.

This filter is *immediately* useful because we might also want to process the *answers* through Markdown. We could do that in the controller, but it would be *much* easier in the template. I'll add some "ticks" around the word "purrrfectly":

48 lines src/Controller/QuestionController.php

```
... lines 1 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownParserInterface $markdownParser)
{
$answers = [
'Make sure your cat is sitting `purrrfectly` still  ',
... lines 35 - 36
];
... lines 38 - 45
}
}
```

Then, in show.html.twig, scroll down to where we loop over the answers. Here, say answer|markdown:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 36
<ul class="list-unstyled">
{% for answer in answers %}
<li class="mb-4">
<div class="d-flex justify-content-center">
... lines 41 - 43
<div class="mr-3 pt-2">
{{ answer|markdown }}
... line 46
</div>
... lines 48 - 52
</div>
</li>
{% endfor %}
</ul>
</div>
{% endblock %}
```

And because answers will eventually be added by users we don't trust, in a real app, I would use answer|striptags|markdown. Cool, right? That would remove any tags HTML added by the user and *then* processes it through Markdown.

Anyways, let's try it! Refresh and... got it! This filter is smart enough to automatically *not* escape the HTML, so we don't need |raw.

Next: I'm *loving* this idea of finding new tools - I mean *services* - and seeing what we can do with them. Let's find another service that's *already* in our app: a caching service. Because parsing Markdown on *every* request can slow things down.

Chapter 3: Cache Service

Parsing markdown on *every* request is going to make our app unnecessarily slow. So... let's cache that! Of course, caching something is "work"... and as I *keep* saying, all "work" in Symfony is done by a service.

Finding the Cache Service

So let's use our trusty debug:autowiring command to see if there are any services that include the word "cache". And yes, you can *also* just Google this and read the docs: we're learning how to do things the hard way to make you dangerous:

```
php bin/console debug:autowiring cache
```

And... cool! There is *already* a caching system in our app! Apparently, there are *several* services to choose from. But, as we talked about earlier, the blue text is the "id" of the service. So 3 of these type-hints are different ways to get the *same* service object, one of these is actually a logger, not a cache and the last one - TagAwareCacheInterface - *is* a different cache object: a more powerful one if you want to do something called "tag-based invalidation". If you don't know what I'm talking about, don't worry.

For us, we'll use the normal cache service... and the CacheInterface is my favorite type-hint because its methods are the easiest to work with.

Using the Cache Service

Head back to the controller and add another argument: CacheInterface - the one from Symfony\Contracts - and call it $cache:

52 lines src/Controller/QuestionController.php

```
... lines 1 - 8
use Symfony\Contracts\Cache\CacheInterface;
... lines 10 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 49
}
}
```

This object makes caching fun. Here's how it works: say $parsedQuestionText = $cache->get(). The first argument is a unique cache *key*. Let's pass markdown_ and then an md5() of $questionText. This will give every unique markdown text its own unique key.

Now, you *might* be thinking:

> Hey Ryan! Don't you need to *first* check to see if this key is *in* the cache already? Something like if ($cache->has())?

Yes... but no. This object works a bit different: the get() function has a *second* argument, a callback function. Here's the idea: *if* this key *is* already in the cache, the get() method will return the value immediately. But if it's *not* - that's a cache "miss" - then it will call our function, *we* will return the parsed HTML, and it will *store* that in the cache.

Copy the markdown-transforming code, paste it inside the callback and return. Hmm, we have two undefined variables because we need to get them into the function's scope. Do that by adding use ($questionText, $markdownParser):

52 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 40
$parsedQuestionText = $cache->get('markdown_'.md5($questionText), function() use ($questionText,
$markdownParser) {
return $markdownParser->transformMarkdown($questionText);
});
... lines 44 - 49
}
}
```

It's happy! I'm happy! Let's try it! Move over and refresh. Ok... it didn't *break*. Did it cache? Down on the web debug toolbar, for the *first* time, the cache icon - these 3 little boxes - shows a "1" next to it. It says: cache hits 0, cache writes 1. Right click that and open the profiler in a new tab.

Cool! Under cache.app - that's the "id" of the cache service - it shows one get() call to some markdown_ key. It was a cache "miss" because it didn't already exist in the cache. Close this then refresh again. This time on the web debug toolbar... yea! We have 1 cache hit! It's alive!

Where is the Cache Stored?

Oh, and if you're wondering *where* the cache is being stored, the answer is: on the filesystem - in a var/cache/dev/pools/ directory. We'll to talk more about that in a little while.

In the controller, make a tweak to our question - how about some asterisks around "thoughts":

52 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 38
$questionText = 'I\'ve been turned into a cat, any *thoughts* on how to turn back? While I\'m **adorable**, I don\'t really
care for cat food.';
... lines 40 - 49
}
}
```

If we refresh now and check the toolbar... yea! The key changed, it was a cache "miss" and the new markdown was rendered.

So the cache system *is* working and it's storing things inside a var/cache/dev/pools/ directory. But... that leaves me with a question. Having these "tools" - these services - automatically available is *awesome*. We're getting a lot of work done quickly.

But because something *else* is instantiating these objects, we don't really have any *control* over them. Like, what if, instead of caching on the filesystem, I wanted to cache in Redis or APCu? How can we do that? More generally, how can we control the *behavior* of services that are given to us by bundles.

That is what we're going to discover next.

Chapter 4: Configuring Bundles

In the show controller, we're using two services: MarkdownParserInterface - from a bundle we installed - and CacheInterface from Symfony itself:

52 lines src/Controller/QuestionController.php

```
... lines 1 - 4
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
... lines 6 - 8
use Symfony\Contracts\Cache\CacheInterface;
... lines 10 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 49
}
}
```

And... this was pretty easy: add an argument with the right type-hint then... use the object!

But I'm starting to wonder how can I control the *behavior* of these services? Like, what if I want Symfony's cache service to store in Redis instead of on the filesystem? Or maybe there are some options that I can pass to the markdown parser service to control its features.

Let's dd($markdownParser) - that's short for dump() and die() - to see what this object looks like:

54 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 40
$parsedQuestionText = $cache->get('markdown_'.md5($questionText), function() use ($questionText,
$markdownParser) {
return $markdownParser->transformMarkdown($questionText);
});
dd($markdownParser);
... lines 46 - 51
}
}
```

Move over and refresh. This is apparently an instance of some class called Max. And... inside, there's a $features property: it kind of looks like you can turn certain features on or off. Interesting.

If we *did* want to control one of these feature flags, we can't really do that right now. Why? Because *we* aren't responsible for creating this object: the bundle just gives it to us.

Bundle Configuration

This is a *super* common problem: a bundle wants to give you a service... but *you* want some control over what options are *passed* to that object when it's instantiated. To handle this, each bundle allows you to pass *configuration* to it where you *describe* the different behavior that you want its services to have.

Let me show you *exactly* what I'm talking about. Open config/bundles.php and copy the KnpMarkdownBundle class name:

14 lines config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 11
Knp\Bundle\MarkdownBundle\KnpMarkdownBundle::class => ['all' => true],
];
```

Now, close this file and find your terminal. We're going to run a special command that will tell us *exactly* what config we can pass to this bundle. Run:

```
php bin/console config:dump KnpMarkdownBundle
```

Boom! This dumps a bunch of example YAML that describes *all* the config that can be used to control the services in this bundle. Apparently, there's an option called parser which has an example value of markdown.parser.max... whatever that means.

Go back to the bundle's documentation and search for parser - better, search for parser:. Here we go... it says that we can apparently set this parser key to any of these 5 strings, which turn on or off different features. Copy the markdown.parser.light value: let's see if we can change the config to *this* value.

Look back at the YAML example at the terminal. The way you configure a bundle is via YAML. Well, you can also use PHP or XML - but *usually* it's done via YAML. We just need a knp_markdown key, a parser key below that and service below *that*. But what file should this live in?

Open up the config/packages/ directory. This is *already* full of files that are configuring *other* bundles. Create a file called knp_markdown.yaml. Inside, say knp_markdown:, enter, go in 4 spaces, then we need parser:, go in 4 spaces again and set service to markdown.parser.light:

4 lines config/packages/knp_markdown.yaml

```
knp_markdown:
parser:
service: markdown.parser.light
```

If you're wondering why I named this file knp_markdown.yaml, I did that to match this first key. But actually... the filename doesn't matter! It's the knp_markdown YAML key that tells Symfony to pass this config to *that* bundle. If we renamed this to i_love_harry_potter.yaml, it would work *exactly* the same.

So... what did this change? Well, find your browser and refresh!

Ah! An error! That was a *genuine* Ryan typo... but I love it!

> Unrecognized option services under knp_markdown.parser. Did you mean service?

Why yes I did! Let me change that:

4 lines config/packages/knp_markdown.yaml

```
knp_markdown:
parser:
service: markdown.parser.light
```

This is one of my *favorite* things about the bundle config system: it's validated. If you make a typo, it will tell you. That's awesome.

Refresh now. Woh! By changing that little config value, it changed the entire *class* of the service object!

The point is: bundles gives you services and every bundle gives you different configuration to help you control the *behavior* of those services. For example, find your terminal and run:

```
php bin/console config:dump FrameworkBundle
```

FrameworkBundle is the *main*, core, Symfony bundle and it gives us the most *foundational* services, like the cache service. This dumps a *huge* list of config options: there's a lot here because this bundle provides many services.

Go Deeper!

You can also pass sub-level key, e.g. cache as the second argument to reduce the output:

```
php bin/console config:dump FrameworkBundle cache
```

Try it!

Of course, if you *really* needed to configure something, you'll probably Google and find the config you need. But how cool is it that you can run this command to see the *full* list of possible config? Let's try another one for TwigBundle, which we installed in the first course:

```
php bin/console config:dump TwigBundle
```

Hello Twig config! Apparently you can create a global Twig variable by adding a globals key. Oh, and this command *also* works if you use the root key, like twig, as the argument:

```
php bin/console config:dump twig
```

That's the exact same list.

Ok! We *now* know that every bundle gives us configuration that allows us to control its services.

But I'm curious about this service config we used for knp_markdown. The docs told us we could use this markdown.parser.light value. But what *is* that string? Is it just some random string that the bundle decided to use for a "light" parser? Actually, that string has a bit *more* meaning. Let's talk about the *massively* important "service container" next.

Chapter 5: The Service Container & Autowiring

We found out that KnpMarkdownBundle allows us to control some of the *features* of the markdown parser by using this knp_markdown.parser.service key:

4 lines config/packages/knp_markdown.yaml

```
knp_markdown:
parser:
service: markdown.parser.light
```

We used their documentation to learn that there were a *few* valid values for this service key.

But what *is* this? What does markdown.parser.light *mean*? Is it just a string that someone invented when they were designing the config for this bundle?

Not exactly: in this case, markdown.parser.light happens to be the *id* of a *service* in the container.

Hello Service Container

Let's... back up. We know that there are many useful objects - called services - floating around in Symfony. What I *haven't* told you yet is that, behind the scenes, Symfony puts all of these services inside something called the "service container". You can think of the service container as *basically* an associative array of services, where each object has a unique id.

How can we see a list of *all* of the services in the container and their IDs? Just run debug:autowiring, right? Actually, not quite. Find your terminal and run a *new* command called:

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

And wow! *This* is the *full* list of *all* the services inside the service container! On the left is the service's id or "key" - like filesystem and on the right is the type of object you would get if you asked for this service. The filesystem service is an instance of Symfony\Component\Filesystem\Filesystem.

You can see that this is a *really* long list. But the truth is that you will probably only ever use a very *small* number of these. *Most* of these are low-level service objects that help *other* more *important* services do their work.

Not all Services are Autowireable

Because of this, *many* of these services *cannot* be accessed via autowiring. What I mean is, for most of these services, there is *no* type-hint that you could use in a controller to fetch that service. So how *would* you access a service if it can't be autowired? Don't worry about that yet, we'll talk about how later.

The point is: not all services can be autowired but the *most* useful ones *can*. To get that, shorter, list of autowireable services, we run:

```
php bin/console debug:autowiring
```

This is not the *full* list of services, but it's *usually* all you'll need to use.

How Autowiring Works

While we're looking at this list, I want to talk about *how* autowiring works. We know that we can use the type-hint on the left to get the service with the id on the right. For example, we can use this AdapterInterface type-hint to fetch some service called cache.app.

Cool. But... how does that *work*? How does Symfony know that the AdapterInterface should give us *that* exact service? When Symfony sees an argument type-hinted with Symfony\Component\Cache\Adapter\AdapterInterface, does... it loops over *every* service in the container and look for one that implements that interface?

Fortunately, no. The way autowiring works is *so* much simpler. When Symfony sees an argument type-hinted with Symfony\Component\Cache\Adapter\AdapterInterface, to figure out *which* service to pass, it does one simple thing: it looks for a service in the container with this *exact* id. Yes, there is a service in the container whose *id* is literally this long interface name.

Let me show you. Once again, run:

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

Most of the service ids have snake-case names: lower case letters and periods. But if you scroll up to the top, there are *also* some services whose ids are class or interface names. And... yea! Here's a service whose id is: Symfony\Component\Cache\Adapter\AdapterInterface! On the right, it says that it's an *alias* to cache.app.

Ok, so there are two important things. First, when you type-hint an argument with Symfony\Component\Cache\Adapter\AdapterInterface, Symfony figures out which service to pass to you by looking for a service in the container with *that* exact id. If it finds it, it uses it. If it doesn't, you get an error. Second, some services - like this one - aren't *real* services: they're *aliases* to another service. If you ask for the AdapterInterface service, Symfony will *actually* give you the cache.app service. It's kind of like a symlink.

This is primarily how the autowiring system works. Bundles add services to the container and typically they use this snake-case naming scheme, which means the services *can't* be autowired. Then, to add autowiring support for the most important services, they register an alias from the class or interface *to* that service.

If this went a little over your head... don't sweat it. The most important thing is this: autowiring isn't magic. When you add a type-hint to autowire a service, Symfony *simply* looks for a service in the container with that id. If it finds one, life is good. If not... error!

Next, let's use our bundle-config skills to figure out how to control where Symfony stores the cache... which we know means: let's *control* how the cache service behaves.

Chapter 6: Configuring the Cache Service

At your terminal, get a list of *all* the services in the container matching the word "markdown" by running:

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

Ah, recognize markdown.parser.light? *That* was what we used for our parser key! Select "1" and hit enter to get more info. No surprise: its class name is Light, which is the *exact* class that's being dump back in our browser, from our controller.

So, on a high level, by adding the parser config, we were basically telling the bundle that we want the "main" markdown parser service to be *this* one. In other words: when we autowire with MarkdownParserInterface, please give us markdown.parser.light.

Figuring out how to Configure the Cache

Anyways, one of our initial goals was to figure out how we could change the cache service to *stop* caching on the filesystem and instead cache somewhere else. In our controller, replace the dd() with dump($cache):

54 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 34 - 44
dump($cache);
... lines 46 - 51
}
}
```

I'm using dump() so that the page still renders - it'll make things easier.

Now, move over, refresh and... interesting. The cache object is an instance of TraceableAdapter but inside it... ah, there's something called FilesystemAdapter. So that kind of proves that the cache is being stored *somewhere* on the filesystem.

Ok, so how can we control that? In reality... you'll probably just Google that to find what config you need. But let's see if we can figure this out ourselves. But, hmm, we don't really know which *bundle* this service comes from.

Open up config/bundles.php. When we started the project, the *only* bundle here was FrameworkBundle - the core Symfony bundle:

14 lines config/bundles.php

```
... lines 1 - 2
return [
Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
... lines 5 - 12
];
```

Every other bundle was installed by us. And... since I don't really see any "CacheBundle", it's a good guess that the cache service comes from FrameworkBundle.

Let's test that theory! Find your terminal, pet your cat, and run:

```
php bin/console config:dump FrameworkBundle
```

Search this *giant* config for cache... I'm looking to see if there is maybe a cache section. Here it is! Under framework, this bundle has a sub-key called cache with quite a lot of example config. Because this is a bit hard to read, re-run this command with FrameworkBundle cache.

```
php bin/console config:dump FrameworkBundle cache
```

This *only* shows the cache section beneath framework.

So... this give us *some* nice information: we can see a key called app set to cache.adapter.filesystem... that *kind* of looks like something we might want to tweak... but I'm not sure... and I don't know what I would change it *to*.

So this is *helpful*... but not *that* helpful.

The debug:config Command

Another way that you can look at bundle configuration is to pass the *exact* same arguments to another command called debug:config:

```
php bin/console debug:config FrameworkBundle cache
```

The difference is subtle: config:dump shows you *examples* of all possible config whereas debug:config shows you *your* real, current values. Let's rerun this without the cache argument to see *all* our FrameworkBundle config:

```
php bin/console debug:config FrameworkBundle
```

Seeing our real values is cool... and we can see that under cache, the app key *is* set to cache.adapter.filesystem. But... we still don't really know what config we should change... or what to change it to!

Changing the Cache Adapter to APCu

Let's go see if the config file for this bundle can help. Logically, because we're configuring the framework key, open up config/packages/framework.yaml:

17 lines config/packages/framework.yaml

```
framework:
secret: '%env(APP_SECRET)%'
#csrf_protection: true
#http_method_override: true
# Enables session support. Note that the session will ONLY be started if you read or write from it.
# Remove or comment this section to explicitly disable session support.
session:
handler_id: null
cookie_secure: auto
cookie_samesite: lax
#esi: true
#fragments: true
php_errors:
log: true
```

Huh, I don't see a cache key! And it's *possible* that there *is* no cache key in our config and that the values *we* saw were the bundle's defaults. But actually, we *do* have some cache config... it's just hiding in its own file: cache.yaml. Inside, it has framework then cache:

20 lines config/packages/cache.yaml

```
framework:
cache:
# Unique name of your app: used to compute stable namespaces for cache keys.
#prefix_seed: your_vendor_name/app_name
# The "app" cache stores to the filesystem by default.
# The data in this cache should persist between deploys.
# Other options include:
# Redis
#app: cache.adapter.redis
#default_redis_provider: redis://localhost
# APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
#app: cache.adapter.apcu
# Namespaced pools use the above "app" backend by default
#pools:
#my.dedicated.cache: null
```

It's not very common for a bundle's config to be separated into two files like this, but it *is* totally legal. Remember: the names of these files are *not* important at all. The cache config was separated because it's complicated enough to have its own file.

Anyways, this file is *full* of useful comments: it tells us how we could use Redis for cache *or* how we could use APCu, which is a simple in-memory cache. Let's use that: uncomment the cache.adapter.apcu line:

20 lines config/packages/cache.yaml

```
framework:
cache:
... lines 3 - 13
# APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
app: cache.adapter.apcu
... lines 16 - 20
```

Before we even *try* that, find your terminal and run the debug:config command again:

```
php bin/console debug:config FrameworkBundle
```

Scroll up to the cache section: yes! This sees our new config! But... what difference does that make in our app? Find your browser, refresh, then hover over the target icon on the web debug toolbar to see the dump. *This* time the adapter object inside is ApcuAdapter! It's caching in memory! We made one little tweak and FrameworkBundle did all the heavy lifting to change the behavior of that service.

Oh, and if you get the error:

> APCu is not enabled

It means you need to install the APCu extension. *How* you do that varies on each system but it's *usually* installed with pecl - like:

```
pecl install apcu
```

After you install it, make sure to restart your web server. You can do that by running

```
symfony server:stop
```

And then re-run the command to start the server:

```
symfony server:start
```

If installing this is causing you problems, don't worry about it. For example purposes, you can use the key cache.adapter.array instead. That's a service that actually does *no* caching, but it will allow you to see how the class changes.

Next, we've started to modify files in this config/packages/ directory. Now I want to talk more about the structure of this directory - *specifically* about Symfony *environments*, which will explain these dev/, prod/ and test/ sub-folders.

Chapter 7: Environments

Your app - the PHP code you write - is a machine: it does whatever interesting thing you told it to do. But that doesn't mean your machine always has the same behavior: by giving that machine different config, you can make it work in different ways. For example, during development, you probably want your app to display errors and your logger to log *all* messages. But on production, you'll probably want to pass configuration to your app that tells it to *hide* exception messages and to only write *errors* to your log file.

To help with this, Symfony has a powerful concept called "environments". This has nothing to do with *server* environments - like your "production environment" or "staging environment". In Symfony, an environment is a set of configuration. And by default, there are two environments: dev - the set of config that logs everything and shows the big exception page - and prod, which is optimized for speed and hides error messages.

And we can *see* these environments in action! Open up public/index.php:

28 lines public/index.php

```
... lines 1 - 2
use App\Kernel;
use Symfony\Component\ErrorHandler\Debug;
use Symfony\Component\HttpFoundation\Request;
require dirname(__DIR__).'/config/bootstrap.php';
if ($_SERVER['APP_DEBUG']) {
umask(0000);
Debug::enable();
}
if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ?? $_ENV['TRUSTED_PROXIES'] ?? false) {
Request::setTrustedProxies(explode(',', $trustedProxies), Request::HEADER_X_FORWARDED_ALL ^
Request::HEADER_X_FORWARDED_HOST);
}
if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? $_ENV['TRUSTED_HOSTS'] ?? false) {
Request::setTrustedHosts([$trustedHosts]);
}
$kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

This is your "front controller": a fancy way of saying that it's the file that's always executed first by your web server.

Where the Environment String is Set

If you scroll down a bit - most things aren't too important - this eventually instantiates an object called Kernel and passes it $_SERVER['APP_ENV']:

28 lines public/index.php

```
... lines 1 - 22
$kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
... lines 24 - 28
```

That APP_ENV thing is configured in another file - .env - at the root of your project:

22 lines .env

```
# In all environments, the following files are loaded if they exist,
# the latter taking precedence over the former:
#
# * .env contains default values for the environment variables needed by the app
# * .env.local uncommitted file with local overrides
# * .env.$APP_ENV committed environment-specific defaults
# * .env.$APP_ENV.local uncommitted environment-specific overrides
#
# Real environment variables win over .env files.
#
# DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED FILES.
#
# Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
# https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=c28f3d37eba278748f3c0427b313e86a
#TRUSTED_PROXIES=127.0.0.0/8,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16
#TRUSTED_HOSTS='^(localhost|example\.com)$'
###
```

There it is: APP_ENV=dev:

22 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=dev
... lines 18 - 22
```

So right now, *we* are running our app in the dev environment. By the way, this entire file is a way to define *environment variables*. Despite the similar name, environment variables are a *different* concept than *Symfony* environments... and we'll talk about them later.

Right now, the important thing to understand is that when this Kernel class is instantiated, we're currently passing the string dev as its first argument:

28 lines public/index.php

```
... lines 1 - 22
$kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
... lines 24 - 28
```

If you want to execute your app in the prod environment, you would change the value in .env:

22 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=dev
... lines 18 - 22
```

We'll do *exactly* that in a few minutes.

Kernel: How Environments Affect things

Anyways, this Kernel class is actually *not* some core class hiding deep in Symfony. Nope! It lives in *our* app: src/Kernel.php. Open that up:

55 lines src/Kernel.php

```
... lines 1 - 2
namespace App;
use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\Config\Resource\FileResource;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\Kernel as BaseKernel;
use Symfony\Component\Routing\RouteCollectionBuilder;
class Kernel extends BaseKernel
{
use MicroKernelTrait;
private const CONFIG_EXTS = '.{php,xml,yaml,yml}';
public function registerBundles(): iterable
{
$contents = require $this->getProjectDir().'/config/bundles.php';
foreach ($contents as $class => $envs) {
if ($envs[$this->environment] ?? $envs['all'] ?? false) {
yield new $class();
}
}
}
public function getProjectDir(): string
{
return \dirname(__DIR__);
}
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
$container->addResource(new FileResource($this->getProjectDir().'/config/bundles.php'));
$container->setParameter('container.dumper.inline_class_loader', \PHP_VERSION_ID < 70400 || $this->debug);
$container->setParameter('container.dumper.inline_factories', true);
$confDir = $this->getProjectDir().'/config';
$loader->load($confDir.'/{packages}/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
protected function configureRoutes(RouteCollectionBuilder $routes): void
{
$confDir = $this->getProjectDir().'/config';
$routes->import($confDir.'/{routes}/'.$this->environment.'/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}'.self::CONFIG_EXTS, '/', 'glob');
}
}
```

The Kernel is the *heart* of your application. Well, you won't need to look at it often... or write code in it... maybe ever, but it *is* responsible for initializing and tying everything together.

What does that mean? You can kind of think of a Symfony app as just 3 parts. First, Symfony needs to know what bundles are in the app. That's the job of registerBundles():

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 17
public function registerBundles(): iterable
{
$contents = require $this->getProjectDir().'/config/bundles.php';
foreach ($contents as $class => $envs) {
if ($envs[$this->environment] ?? $envs['all'] ?? false) {
yield new $class();
}
}
}
... lines 27 - 53
}
```

Then, it needs to know what config to *pass* to those bundles to help them configure their services. That's the job of configureContainer():

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
$container->addResource(new FileResource($this->getProjectDir().'/config/bundles.php'));
$container->setParameter('container.dumper.inline_class_loader', \PHP_VERSION_ID < 70400 || $this->debug);
$container->setParameter('container.dumper.inline_factories', true);
$confDir = $this->getProjectDir().'/config';
$loader->load($confDir.'/{packages}/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
... lines 45 - 53
}
```

And *finally*, it needs to get a list of all the routes in your app. That's the job of configureRoutes():

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 45
protected function configureRoutes(RouteCollectionBuilder $routes): void
{
$confDir = $this->getProjectDir().'/config';
$routes->import($confDir.'/{routes}/'.$this->environment.'/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}'.self::CONFIG_EXTS, '/', 'glob');
}
}
```

By the way, if you start a Symfony 5.1 app, you probably won't see a registerBundles() method. That's because it was moved into a core trait, but it has the *exact* logic that you see here.

registerBundles()

Back up in registerBundles(), the flag that we passed to Kernel - the dev string - eventually becomes the property $this->environment:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 17
public function registerBundles(): iterable
{
... line 20
foreach ($contents as $class => $envs) {
if ($envs[$this->environment] ?? $envs['all'] ?? false) {
... line 23
}
}
}
... lines 27 - 53
}
```

This methods uses that. Open up config/bundles.php:

14 lines config/bundles.php

```
... lines 1 - 2
return [
Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true, 'test' => true],
Symfony\WebpackEncoreBundle\WebpackEncoreBundle::class => ['all' => true],
Knp\Bundle\MarkdownBundle\KnpMarkdownBundle::class => ['all' => true],
];
```

Notice that all of the bundles classes are set to an array, like 'all' => true or some have 'dev' => true and 'test' => true. This is declaring which *environments* that bundle should be enabled in. *Most* bundles will be enabled in *all* environments. But some - like DebugBundle or WebProfilerBundle - are tools for development. And so, they are *only* enabled in the dev environment. Oh, and there is *also* a third environment called test, which is used if you write automated tests.

Over in registerBundles(), this loops over the bundles and *uses* that info to figure out if that bundle should be enabled in the current environment or not:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 17
public function registerBundles(): iterable
{
... line 20
foreach ($contents as $class => $envs) {
if ($envs[$this->environment] ?? $envs['all'] ?? false) {
yield new $class();
}
}
}
... lines 27 - 53
}
```

This is why the web debug toolbar & profiler won't show up in the prod environment: the bundle that powers those isn't enabled in prod!

configureContainer: Environment-Specific Config Files

Anyways, bundles give us *services* and, as we've learned, *we* need the ability to pass config *to* those bundles to *control* those services. That's the job of configureContainer():

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
$container->addResource(new FileResource($this->getProjectDir().'/config/bundles.php'));
$container->setParameter('container.dumper.inline_class_loader', \PHP_VERSION_ID < 70400 || $this->debug);
$container->setParameter('container.dumper.inline_factories', true);
$confDir = $this->getProjectDir().'/config';
$loader->load($confDir.'/{packages}/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
... lines 45 - 53
}
```

I *love* this method. It's completely responsible for loading *all* the config files inside the config/ directory. Skip passed the first 4 lines, if you have them, which set a few low-level flags.

The *real* magic is this $loader->load() stuff, which in a Symfony 5.1 app will look like $container->import()... but it works the same. This code does one simple thing: loads config files. The first line loads all files in the config/packages/ directory. That self::CONFIG_EXTS thing refers to a constant that tells Symfony to load any files ending in .php, .xml, .yaml, .yml. Most people use YAML config, but you can also use XML or PHP.

Anyways, *this* is the line that loads all the YAML files inside config/packages. I mentioned earlier that the *names* of these files aren't important. For example, this file is called cache.yaml even though it's technically configuring the framework bundle:

20 lines config/packages/cache.yaml

```
framework:
cache:
... lines 3 - 20
```

This shows why:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
... lines 35 - 39
$loader->load($confDir.'/{packages}/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
... lines 45 - 53
}
```

Symfony loads *all* of the files - regardless of their name - and internally creates one giant, array of configuration. Heck, we could combine *all* the YAML files into one *big* file and everything would work fine.

But what I *really* want you to see is the *next* line. This says: load everything from the config/packages/ "environment" directory:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
... lines 35 - 40
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
... lines 42 - 43
}
... lines 45 - 53
}
```

Because we're in the dev environment, it's loading the 4 files in config/packages/dev. This allows us to *override* configuration in *specific* environments!

For example, in the prod/ directory, open the routing.yaml file. This configures the router and sets a strict_requirements key to null:

4 lines config/packages/prod/routing.yaml

```
framework:
router:
strict_requirements: null
```

It's not really important *what* this does. What *is* important is that the *default* value for this is true, but a better value for production is null. This override accomplishes that. I'll close that file.

So this *whole* idea of environments is, ultimately, nothing more than a configuration trick: Symfony loads everything from config/packages and *then* loads the files in the environment subdirectory... which lets us override the original values.

Oh, these last two lines load services.yaml and services_{environment}.yaml:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
... lines 35 - 41
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
... lines 45 - 53
}
```

That's where we add our *own* services to the container and we'll talk about them soon.

configureRoutes()

Ok, we've now initialized our bundles and loaded config. The *last* job of Kernel is to figure out what routes our app needs. Look down at configureRoutes():

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 45
protected function configureRoutes(RouteCollectionBuilder $routes): void
{
$confDir = $this->getProjectDir().'/config';
$routes->import($confDir.'/{routes}/'.$this->environment.'/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}/*'.self::CONFIG_EXTS, '/', 'glob');
$routes->import($confDir.'/{routes}'.self::CONFIG_EXTS, '/', 'glob');
}
}
```

Ah, it does... pretty much the exact same thing as configureContainer(): it loads all the files from config/routes - which is just one annotations.yaml file - and *then* loads any extra files in config/routes/{environment}.

Let's look at one of these: config/routes/dev/web_profiler.yaml:

8 lines config/routes/dev/web_profiler.yaml

```
web_profiler_wdt:
resource: '@WebProfilerBundle/Resources/config/routing/wdt.xml'
prefix: /_wdt
web_profiler_profiler:
resource: '@WebProfilerBundle/Resources/config/routing/profiler.xml'
prefix: /_profiler
```

This is what's responsible for importing the web debug toolbar and profiler routes into our app! At your terminal, run:

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

Yep! These /_wdt and /_profiler routes are here thanks to that file. This is *another* reason why the web debug toolbar & profiler won't be available in the prod environment.

Next, let's *change* environments: from dev to prod and see the difference. We're also going to use our new environment knowledge to *change* the cache configuration *only* in the prod environment.

Chapter 8: Controlling the prod Environment

Let's see what our app looks like if we change to the prod environment. To do that, open the .env file and change APP_ENV to prod:

22 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=prod
... lines 18 - 22
```

Clearing Cache in the prod Environment

Cool! Now, find your browser, refresh and... it works! Well, actually, we got lucky. Behind the scenes, when we load a page, Symfony caches configuration, templates and other things for performance. In the dev environment, if we *update* a config file, Symfony *automatically* rebuilds the cache. So it's not something we even need to think about.

But in the prod environment - which is *primed* for performance - Symfony does *not* automatically rebuild your cache files. For example, if we added a new route to our app and then went to that URL in the prod environment, it would give us a page not found error! Why? Because our app would be using outdated routing cache.

That's why, *whenever* you change to the prod environment, you need to find your terminal and run a special command:

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

This clears the cache so that, on our next reload, *new* cache will be built. The cache is stored in a var/cache/prod directory. Oh, and notice that bin/console is smart enough to know that we're in the prod environment.

In practice, I *rarely* switch to the prod environment on my local computer. The most common time I run cache:clear is when I'm deploying.

Now our app *definitely* works. And notice: no web debug toolbar!

Let's see Symfony's automatic caching system in action. Open up templates/question/show.html.twig and... let's make some small change - like Question::

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
<div class="row">
<div class="col-12">
<h2 class="my-4">Question:</h2>
... lines 10 - 26
</div>
</div>
... lines 29 - 56
</div>
{% endblock %}
```

This time, when we refresh, the change is *not* there. That's because Symfony caches Twig templates. Now find your terminal, run:

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

And come back to refresh. There's the change!

Different Cache Adapter in prod

Now that we understand environments, I have a challenge for us! At the top of the page, we're still dumping the cache service inside our controller. The class is ApcuAdapter because that's what we configured inside of config/packages/cache.yaml:

20 lines config/packages/cache.yaml

```
framework:
cache:
... lines 3 - 13
# APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
app: cache.adapter.apcu
... lines 16 - 20
```

APCu is great. But maybe for simplicity, because it requires you to have a PHP extension installed, we want to use the filesystem adapter in the dev environment and APCu *only* for prod. How could we do that?

Let's think about it: we know how to override configuration in a specific environment... so we *could* override just this *one* config key in the dev environment.

To do that, in the dev/ directory, create a new file. It *technically* doesn't matter what it's called, but because we value our sanity, call it cache.yaml. Inside, say framework:, cache:, app: and the name of the original default value for this: cache.adapter.filesystem:

4 lines config/packages/dev/cache.yaml

```
framework:
cache:
app: cache.adapter.filesystem
```

That's... all we need! Let's see if it works! Because we're still in the prod environment, find your terminal and clear the cache:

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

When it finishes, go refresh the page. Good: in prod it's *still* using ApcuAdapter. Now go find the .env file at the root of the project... change APP_ENV back to dev:

22 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=dev
... lines 18 - 20
###
```

And refresh the page.

Because the web debug toolbar is back, our dump is hiding inside its target icon. Let's see... yes! It's FilesystemAdapter!

Ok team: we've mastered environments and configuring services that are coming from bundles. So let's take things up to the next level: let's create our *own* service objects! That's next.

Chapter 9: Creating a Service

Okay, so bundles give us services and services do work. So... if we needed to write our *own* custom code that did work... can we create our own service class and put the logic there? Absolutely! And it's something that you're going to do *all* the time. It's a great way to organize your code, gives you the ability to re-use logic *and* allows you to write unit tests if you want. So... let's do it!

We're *already* doing some work. It may not look like a lot, but the logic of parsing the markdown and caching the result *is* work:

54 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 31
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
$answers = [
'Make sure your cat is sitting `purrrfectly` still  ',
'Honestly, I like furry shoes better than MY cat',
'Maybe... try saying the spell backwards?',
];
$questionText = 'I\'ve been turned into a cat, any *thoughts* on how to turn back? While I\'m **adorable**, I don\'t really care for cat food.';
$parsedQuestionText = $cache->get('markdown_'.md5($questionText), function() use ($questionText, $markdownParser) {
return $markdownParser->transformMarkdown($questionText);
});
dump($cache);
return $this->render('question/show.html.twig', [
'question' => ucwords(str_replace('-', ' ', $slug)),
'questionText' => $parsedQuestionText,
'answers' => $answers,
]);
}
}
```

It would be nice to move this into its *own* class. That would make the controller a bit easier to read *and* we could re-use this markdown caching logic somewhere else if we needed to, which we will later.

Creating the Service

So how do we create our very own service? Start by creating a class anywhere in src/. It doesn't matter where but I'll create a new sub-directory called Service/, which I often use when I can't think of a better directory to put my class in. Inside, add a new PHP class called, how about MarkdownHelper:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 2
namespace App\Service;
class MarkdownHelper
{
... lines 7 - 12
}
```

And cool! PhpStorm automatically added the correct namespace to the class. Thanks!

Unlike controllers, this class has *nothing* to do with Symfony... it's just a class *we* are creating for our own purposes. And so, it doesn't need to extend a base class or implement an interface: this class will look *however* we want.

Let's think: we're probably going to want a function called something like parse(). It will need a string argument - how about $source - and it will return a string, which will be the finished HTML:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
class MarkdownHelper
{
public function parse(string $source): string
{
... lines 9 - 11
}
}
```

Nice! Back in QuestionController, copy the three lines of logic and paste them into the new method. Let's fix a few things: return the value:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
class MarkdownHelper
{
public function parse(string $source): string
{
$parsedQuestionText = $cache->get('markdown_'.md5($questionText), function() use ($questionText,
$markdownParser) {
return $markdownParser->transformMarkdown($questionText);
});
}
}
```

then change $questionText to $source in three different places:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
class MarkdownHelper
{
public function parse(string $source): string
{
return $cache->get('markdown_'.md5($source), function() use ($source, $markdownParser) {
return $markdownParser->transformMarkdown($source);
});
}
}
```

Your Class is Already a Service! Use it!

We still have a few undefined variables... but... I want you to ignore them for now. Because, congratulations! This may not, ya know, "work" yet, but you just created your first service! Remember: a service is just a class that does work.

Ok, so, how can we use this inside our controller? We *already* know the answer. If we need a service from the container, we need to add an argument with the right type-hint. But... is our service already... somehow in Symfony's container? Let's find out! At your terminal, run:

```
php bin/console debug:autowiring Markdown
```

Hmm, it only shows the two results from the bundle. But wait! At the bottom it says:

> 1 more concrete service would be displayed when adding the --all option.

Um... ok - let's add --all to this:

```
php bin/console debug:autowiring Markdown --all
```

And there it is! Why did we need this --all flag? Well, the "mostly-true" explanation is that, to keep this list short, Symfony hides *your* services from the list... because you already know they exist.

Anyways, yes! Our service is - *somehow* - already available in Symfony's container. We'll learn *how* that happened later, but the important thing now is that we can use the MarkdownHelper type-hint to get an instance of our class.

Let's do it! Back in the controller, add a 4th argument: MarkdownHelper $markdownHelper:

51 lines src/Controller/QuestionController.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 48
}
}
```

Down below, say $parsedQuestionText = $markdownHelper->parse($questionText):

51 lines src/Controller/QuestionController.php

```
... lines 1 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 41
$parsedQuestionText = $markdownHelper->parse($questionText);
... lines 43 - 48
}
}
```

Testing time! Refresh and... yea! Undefined variable coming from MarkdownHelper! Woo! I'm happy because this *proves* that the service was autowired into the controller. The method is blowing up... but our service is alive!

Inside of MarkdownHelper, we're trying to use the cache and markdown parser services... but we don't have access to those here:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
class MarkdownHelper
{
public function parse(string $source): string
{
return $cache->get('markdown_'.md5($source), function() use ($source, $markdownParser) {
return $markdownParser->transformMarkdown($source);
});
}
}
```

How can we get them? The answer to that is "dependency injection": a threatening-sounding word for a delightfully simple concept. It's also one of the most *fundamental* concepts in Symfony... or really any object-oriented coding. Let's tackle it next!

Chapter 10: Autowiring Dependencies into a Service

Our MarkdownHelper service is... sort of working:

14 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
class MarkdownHelper
{
public function parse(string $source): string
{
return $cache->get('markdown_'.md5($source), function() use ($source, $markdownParser) {
return $markdownParser->transformMarkdown($source);
});
}
}
```

We can call it from the controller... but inside, we're trying to use two services - cache and markdown parser - that we don't have access to. How can we get those objects?

Real quick: I've said many times that there are service objects "floating around" in Symfony. But even though that's true, you *can't* just grab them out of thin air. There's no, like, Cache::get() static call or something that will magically give us that object. And that's *good* - that's a recipe for writing bad code.

Passing Dependencies to the Method?

So how *can* we get access to services? Currently, we only know one way: by autowiring them into our controller methods:

51 lines src/Controller/QuestionController.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
... lines 7 - 9
use Symfony\Contracts\Cache\CacheInterface;
... lines 11 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 48
}
}
```

Which we can't do here, because that's a superpower that *only* controllers have.

Hmm, but one idea is that we could *pass* the markdown parser and cache *from* our controller *into* parse():

51 lines src/Controller/QuestionController.php

```
... lines 1 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 41
$parsedQuestionText = $markdownHelper->parse($questionText);
... lines 43 - 48
}
}
```

This won't be our *final* solution, but let's try it!

On parse(), add two more arguments: MarkdownParserInterface $markdownParser and CacheInterface - from Symfony\Contracts - $cache:

17 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
use Symfony\Contracts\Cache\CacheInterface;

class MarkdownHelper
{
public function parse(string $source, MarkdownParserInterface $markdownParser, CacheInterface $cache): string
{
... lines 12 - 14
}
}
```

Cool! This method is happy.

Back in QuestionController, pass the two extra arguments: $markdownParser and $cache:

51 lines src/Controller/QuestionController.php

```
... lines 1 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 41
$parsedQuestionText = $markdownHelper->parse($questionText, $markdownParser, $cache);
... lines 43 - 48
}
}
```

Ok team - let's see if it works! Find your browser and refresh. It does!

Method Arguments versus Class Dependencies

On a high level, this solution makes sense: since we can't grab service objects out of thin air in MarkdownHelper, we *pass* them in. But, if you think about it, the markdown parser and cache objects aren't really "input" to the parse() function. What I mean is, the $source argument to parse() makes *total* sense: when we call the method, we *of course* need to pass in the content we want parsed.

But these next two arguments don't really control how the function behaves... you would probably always pass these *same* values *every* time you called the method. No, instead of function arguments, these objects are more *dependencies* that the service needs in order to do its work. It's just stuff that *must* be available so that parse() can do its job.

For *dependencies* like this - for service objects or configuration that your service simply *needs*, instead of passing them through the individual methods, we instead pass them through the *constructor*.

Dependency Injection via the Constructor

At the top, create a new public function __construct(). Move the two arguments here... and delete them from parse():

26 lines src/Service/MarkdownHelper.php

```
... lines 1 - 4
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
use Symfony\Contracts\Cache\CacheInterface;
class MarkdownHelper
{
... lines 10 - 12
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
... lines 15 - 16
}
public function parse(string $source): string
{
... lines 21 - 23
}
}
```

Before we finish this, I need to tell you that autowiring in fact works in *two* places. We already know that you can autowire services into your controller methods. But you can *also* autowire services into the __construct() method of a service. In fact, that is the *main* place where autowiring is meant to work. The fact that autowiring *also* works for controller methods was... kind of an added feature to make life easier. And it *only* works for controllers - you can't add a MarkdownParserInterface argument to parse() and expect Symfony to autowire that because *we* are the ones that are calling that method and passing it arguments.

Anyways, when Symfony instantiates MarkdownHelper, it will pass us these two arguments thanks to autowiring. What do we... *do* with them? Create two private properties: $markdownParser and $cache. Then, in the constructor, set those: $this->markdownParser = $markdownParser and $this->cache = $cache:

26 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
private $markdownParser;
private $cache;
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache)
{
$this->markdownParser = $markdownParser;
$this->cache = $cache;
}
... lines 18 - 24
}
```

Basically, when the object is instantiated, we're taking those objects and storing them for later. Then, whenever we call parse(), the two properties will already hold those objects. Let's use them: $this->cache, and then we don't need to pass $markdownParser to the use because we can instead say $this->markdownParser:

26 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
... lines 10 - 18
public function parse(string $source): string
{
return $this->cache->get('markdown_'.md5($source), function() use ($source) {
return $this->markdownParser->transformMarkdown($source);
});
}
}
```

I love it! This class is now a *perfect* service: we add our dependencies to the constructor, set them on properties, then use them below.

Dependency Injection?

By the way, what we *just* did has a fancy name! Ooo. It's *dependency injection*. But don't be too impressed: it's a simple concept. Whenever you're inside a service - like MarkdownHelper - and you realize that you need something that you don't have access to, you'll follow the *same* solution: add another constructor argument, create a property, *set* that onto the property, then use it in your methods. *That* is dependency injection. A *big* word to *basically* mean: if you need something, don't expect to grab it out of thin air: force Symfony to pass it *to* you by adding it to the constructor.

Phew! Back in QuestionController, we can celebrate by removing the two extra arguments to parse():

49 lines src/Controller/QuestionController.php

```
... lines 1 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownHelper $markdownHelper)
{
... lines 33 - 39
$parsedQuestionText = $markdownHelper->parse($questionText);
... lines 41 - 46
}
}
```

And when we move over and refresh... it works!

If this didn't feel *totally* comfortable yet, don't worry. The process of creating services is something that we're gonna to do over and over again. The benefit is that we now have a beautiful service - a tool - that we can use from anywhere in our app. We pass it the markdown string and it takes care of the caching and markdown processing.

Heck, in QuestionController, we don't even need the $markdownParser and $cache arguments to the show() method!

51 lines src/Controller/QuestionController.php

```
... lines 1 - 5
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;
... lines 7 - 9
use Symfony\Contracts\Cache\CacheInterface;
... lines 11 - 12
class QuestionController extends AbstractController
{
... lines 15 - 32
public function show($slug, MarkdownParserInterface $markdownParser, CacheInterface $cache, MarkdownHelper
$markdownHelper)
{
... lines 35 - 48
}
}
```

Remove them and, on top of the class, even though it doesn't hurt anything, let's delete the two use statements.

Next: the service container holds services! That's true! But it *also* holds something else: scalar configuration.

Chapter 11: Parameters

We know there are lots of useful service objects floating around that, internally, Symfony keeps inside something called a container. But this container thing can *also* hold something *other* than services: it can hold scalar configuration called *parameters*. You can use these to do some cool stuff.

debug:container --parameters

Earlier, we learned that you can get a list of *every* service in the container by running debug:container.

php bin/console debug:container

Big giant list. To get a list of the "parameters" in the container, add a --parameters flag:

php bin/console debug:container --parameters

There are a *bunch* of them. But most of these aren't very important - they're values used internally by low-level services. For example, one parameter is called kernel.charset, which is set to UTF-8. That's probably used in various places internally.

Adding Parameters

The point is: the container can *also* hold scalar config values and it's sometimes useful to add your *own*. So, how could we do that?

Go into config/packages/ and open *any* config file. Let's open cache.yaml because we're going to use parameters for a caching trick. Add a key called parameters:

23 lines config/packages/cache.yaml

```
parameters:
... lines 2 - 23
```

We know that the framework key means that the config below it will be passed to FrameworkBundle. The parameters key is special: it means that we're adding parameters to the container. Invent a new one called, how about, cache_adapter set to cache.adapter.apcu:

23 lines config/packages/cache.yaml

```
parameters:
cache_adapter: cache.adapter.apcu
... lines 3 - 23
```

There *should* now be a new parameter in the container called cache_adapter. We're not *using* it anywhere... but it should exist.

Reading a Parameter in a Controller

How *do* we use it? There are two ways. First, you could read it in a controller. Open src/Controller/QuestionController.php and find the show() method. Inside, use a new shortcut method dump($this->getParameter('cache_adapter')):

51 lines src/Controller/QuestionController.php

```
... lines 1 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownHelper $markdownHelper)
{
dump($this->getParameter('cache_adapter'));
... lines 34 - 48
}
}
```

If we move over and refresh the show page... there it is! The string cache.adapter.apcu.

Reading Parameters in Config Files

But this is *not* the most common way to use parameters. The *most* common way is to *reference* them in *config* files. Once a parameter exists, you can *use* that parameter in *any* config file via a special syntax. Down below in cache.yaml, remove the cache.adapter.apcu string and replace it with quotes, then %cache_adapter%:

23 lines config/packages/cache.yaml

```
parameters:
cache_adapter: cache.adapter.apcu
framework:
cache:
... lines 6 - 16
# APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
app: '%cache_adapter%'
... lines 19 - 23
```

Let's try it! Move over and refresh. Yes! Things are still working.

The *key* is that when you surround something by % signs, Symfony realizes that you are referencing a *parameter*. These parameters sort of work like variables inside of config files. Oh, and quotes are *normally* optional in YAML, but they *are* needed when a string starts with %. If you're ever not sure if quotes are needed around something, just play it safe and add them.

Overriding a Parameter in Dev

But so far... this isn't *that* interesting: we're basically setting a variable up here and using it below... which is fine, but not that useful yet.

However, we can use parameters to do our "cache adapter override" in a smarter way. Remember, in dev/cache.yaml, we're overriding the framework.cache.app key to be cache.adapter.filesystem:

4 lines config/packages/dev/cache.yaml

```
framework:
cache:
app: cache.adapter.filesystem
```

Now, instead of overriding that config, we can override the *parameter*.

Add parameters: and then use the same name as the other file: cache_adapter: set to cache.adapter.filesystem:

3 lines config/packages/dev/cache.yaml

```
parameters:
cache_adapter: cache.adapter.filesystem
```

Ok, in the main cache.yaml, we're setting the app key to the cache_adapter parameter. This is initially set to apcu, but we override it in the dev environment to be filesystem. This works because the *last* value wins: Symfony doesn't resolve the cache_adapter parameter until *all* the config files have been loaded.

We can see this in the terminal. Run:

```
php bin/console debug:container --parameters
```

And... yes! The value is cache.adapter.filesystem. How would this parameter look in the prod environment? We could change the environment in the .env file and re-run this command. *Or*, we can use a trick: run the command with --env=prod. That flag works for *any* command:

```
php bin/console debug:container --parameters --env=prod
```

This time, it's cache.adapter.apcu. Oh, and I didn't clear my cache before running this, but you really *should* do that before doing *anything* in the prod environment.

Parameters Usually Live in services.yaml

So... those are parameters! Simple config variables to help you kick butt.

But I *do* want to change one thing. By convention, files in the config/packages directory hold bundle configuration - like for FrameworkBundle or TwigBundle. For services and parameters that *we* want to add to the container directly, there's a different file: config/services.yaml.

Copy the parameter from cache.yaml:

23 lines config/packages/cache.yaml

```
parameters:
cache_adapter: cache.adapter.apcu
... lines 3 - 23
```

Remove it, and paste it here:

29 lines config/services.yaml

```
# This file is the entry point to configure your own services.
# Files in the packages/ subdirectory configure your dependencies.
# Put parameters here that don't need to change on each machine where the app is deployed
# https://symfony.com/doc/current/best_practices/configuration.html#application-related-configuration
parameters:
cache_adapter: cache.adapter.apcu
... lines 8 - 29
```

Now, *technically*, this makes no difference: Symfony loads the files in config/packages and services.yaml at the same time: any config can go in any file. But defining all of your parameters in one spot is nice.

Creating services_dev.yaml

Of course, you might *now* be wondering: what about the parameter in config/packages/dev/cache.yaml? Remember: the class that loads these files is src/Kernel.php:

55 lines src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel
{
... lines 14 - 32
protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader): void
{
... lines 35 - 39
$loader->load($confDir.'/{packages}/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{packages}/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}'.self::CONFIG_EXTS, 'glob');
$loader->load($confDir.'/{services}_'.$this->environment.self::CONFIG_EXTS, 'glob');
}
... lines 45 - 55
```

It loads all the files in packages/, packages/{environment} and then services.yaml. Oh, but there is one *more* line: it *also* tries to load a services_{environment}.yaml file. If you need to override a parameter or service - more on that soon - in the dev environment, this is the key.

Create that file: services_dev.yaml. Then copy the config from dev/cache.yaml and paste it here:

3 lines config/services_dev.yaml

```
parameters:
cache_adapter: cache.adapter.filesystem
```

We can now completely *delete* the old cache.yaml file.

That's it! We set the cache_adapter parameter in services.yaml, override it in services_dev.yaml and reference the final value in cache.yaml. We *rule*.

Next, let's leverage a *core* parameter to disable our markdown caching in the dev environment. The trick is: how can we access configuration from inside our MarkdownHelper service?

Chapter 12: Service Config & Non-Autowireable Arguments

At your terminal, run

php bin/console debug:container --parameters

Most parameters are low-level values that probably aren't useful to us. But there are several that start with kernel. that *are* useful. These are added by Symfony itself. Need to know the current environment? You can use kernel.environment. Oh, and I use kernel.project_dir pretty frequently: if you ever need to point to a file path from a config file, *this* is super handy.

But what the one I want to use right now is kernel.debug, which is currently set to true. Basically when you're in the dev or test environments, this is true. When you're in prod, it's false.

Here's the challenge: let's pretend that we're trying to customize the markdown-parsing logic itself: we're making changes to the HTML it outputs somehow. But because we're caching the Markdown, each time we make a change to the parser, we need to clear the cache before we can see it. To make life nicer, let's use this flag to *disable* markdown caching when kernel.debug is set to true.

Dependency Injection with Scalar Values

Open up MarkdownHelper. In the same way that this class needs the MarkdownParserInterface and CacheInterface services to do its job, it now *also* needs to know whether or not we're in debug mode. What do we do when we're inside a service and need access to a service or some config that we don't have?

The answer is always the same: create a __construct() method if you don't have one already, add an argument, set that argument on a new property, then use it. *Usually* the "thing" we need is another service. But occasionally you'll need some configuration - like a debug boolean or maybe an API key. Even in those cases, we use this dependency injection flow.

Add a new argument called, how about, bool $isDebug. Create a property for this - private $isDebug - and set that in the constructor: $this->isDebug = $isDebug:

32 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
... lines 10 - 11
private $isDebug;
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug)
{
... lines 16 - 17
$this->isDebug = $isDebug;
}
... lines 20 - 30
}
```

Down in parse, use it: if $this->isDebug, then copy the return statement from below and paste it here:

32 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
... lines 10 - 20
public function parse(string $source): string
{
if ($this->isDebug) {
return $this->markdownParser->transformMarkdown($source);
}
... lines 26 - 29
}
}
```

Go team!

Non-Autowireable Arguments

So far, each time we've added an argument to a constructor, Symfony has known what to pass to it thanks to autowiring. But what about now? Do you think that, when Symfony tries to instantiate our service, it will know what value to pass to this $isDebug argument?

Let's find out! Move over and refresh. Doh! Syntax error! Come on Ryan! I'll add my missing semicolon and... drum roll... refresh!

The answer is no: Symfony does *not* know what to pass to $isDebug. But we get an awesome error: "cannot resolve argument $markdownHelper of QuestionController::show()" - that's telling us which controller this all starts with - and then:

> Cannot autowire service MarkdownHelper: argument $isDebug of method __construct is type-hinted bool. You should configure its value explicitly.

Yep, autowiring *only* works with class or interface type-hints. And that makes sense: how could Symfony *possibly* guess what we want for this argument? It's not, fortunately, *that* magic.

Adding Extra Service Config to services.yaml

This is our first example of a constructor argument that can't be autowired. When this happens, it's no problem: we just need to give Symfony a little "hint" about what we want.

How? Open up config/services.yaml. At a high level, this is where we configure our *own* services and parameters. For now, skip passed all the stuff on top - we're going to explore what that does soon. At the bottom of the file, indent four spaces so that you're under the services key, then type the full class name to our service: App\Service\MarkdownHelper:. Below this, we can pass configuration to help Symfony instantiate the object. Do that by saying arguments: and, beneath that, $isDebug set to, for now, just true:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 29
App\Service\MarkdownHelper:
arguments:
$isDebug: true
```

Yep, we're *literally* saying:

> Hey Symfony! If you see an argument named $isDebug in the constructor, pass true. But please keep autowiring the *other* arguments, because that rocks.

So... that *should* be enough to get it working! Try it! When we refresh... it's back! Let's *really* make sure it's doing what we want: inside MarkdownHelper, add dump($isDebug):

33 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
... lines 10 - 13
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug)
{
... lines 16 - 18
dump($isDebug);
}
... lines 21 - 31
}
```

This time when we reload... there it is: true.

Referencing %kernel.debug%

Of course, we don't *really* want to hardcode true: we want to reference the kernel.debug parameter. No problem: in services.yaml, add quotes then %kernel.debug%:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 29
App\Service\MarkdownHelper:
arguments:
$isDebug: '%kernel.debug%'
```

When we try the page, it should still be true... and it is! Let's double-check the prod environment. Find the .env file, change APP_ENV to prod:

22 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=prod
... lines 18 - 22
```

Then go clear the cache:

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

When that's done, find your browser and take it for a spin. Yep! Up on top, it prints false. The power! Change the environment back to dev:

22 lines .env

```
... lines 1 - 16
APP_ENV=dev
... lines 18 - 22
```

The Amazing (but not yet impressive) bind

Before we keep going, head back to services.yaml. There *are* other config keys we can use below a service to control how it's instantiated, but most of them aren't too important or common. However, there *is* one I want to show you. Rename arguments to bind:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 29
App\Service\MarkdownHelper:
bind:
$isDebug: '%kernel.debug%'
```

If you move over and refresh... that makes no difference at all. In fact, arguments and bind are *almost* identical. Really, they're *so* similar, that I'm not even going to explain the *subtle* difference. Just know that bind is *slightly* more powerful and it's what I typically use.

Next: I want to demystify what this file is doing on top so that we can *really* understand how services are being added to the container and how we can control them.

Chapter 13: All about services.yaml

When Symfony creates its container, it needs to get a big list of *every* service that should be in the container: each service's id, class name and the arguments that should be passed to its constructor. It gets this big list from exactly *two* places. The first - and the biggest - is from *bundles*.

If we run:

php bin/console debug:container

The *vast* majority of these services come from bundles. Each bundle has a list of the services it provides, which includes the id, class name and arguments for each one.

The *second* place the container goes to complete its list of services is our src/ directory. We already know that MarkdownHelper is in the service container because we've been able to autowire it to our controller.

services.yaml Registers *our* Services

So when the container is being created, it asks each bundle for its service list and then - to learn about *our* services - it reads services.yaml.

When Symfony starts parsing this file, *nothing* in the src/ directory has been registered as a service in the container. *Adding* our classes to the container is, in fact, the *job* of this file. And the way it does it is pretty amazing.

The _defaults Key

The first thing under services is a special key called _defaults:

33 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
autowire: true # Automatically injects dependencies in your services.
autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 14 - 33
```

This defines *default* options that should be applied to *each* service that's added to the container in this file. Every service registered here will have an option called autowire set to true and another called autoconfigure set to true.

Let me... say that a different way. When you configure a single service - like we're doing for MarkdownHelper - it's totally legal to say autowire: true or autowire: false. That's an option that you can configure on *any* service. The _defaults sections says:

> Hey! Don't make me manually add autowire: true to every service - make that the default value.

The autowire Option

What *does* the autowire option mean? Simply, it tells Symfony's container:

> Please try to guess the arguments to my constructor by reading their type-hints.

We *like* that feature, so it will be "on" automatically for all of our services. The other option - autoconfigure - is more subtle and we'll talk about it later.

So these 3 lines don't *do* anything: they just set up default config.

Service Auto-Registration

The next section - these 3 lines starting with App\ - are the key to *everything*:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 14
# makes classes in src/ available to be used as services
# this creates a service per class whose id is the fully-qualified class name
App\:
resource: '../src/*'
exclude: '../src/{DependencyInjection,Entity,Migrations,Tests,Kernel.php}'
... lines 20 - 33
```

This says:

> Hey container! Please look at my src/ directory and register *every* class you find as a service in the container.

And when it does this, thanks to _defaults, every service will have autowire and autoconfigure enabled. *This* is why MarkdownHelper was *instantly* available as a service in the container and why its arguments are being autowired. This is called "service auto-registration".

But remember, every service in the container needs to have a unique id. When you auto-register services like this, the id matches the class name. We can see this! The *vast* majority of the services in debug:container have a snake-case id. But if you go *all* the way to the top, *our* services are *also* in this list each service ID is *identical* to its class name.

This is done to keep life simple... but *also* because it powers autowiring. If we try to autowire App\Service\MarkdownHelper into our controller or another service, in order to figure out what to pass to that argument, autowiring looks in the container for a service whose id *exactly* matches the type-hint: App\Service\MarkdownHelper.

Anyways, back in services.yaml, after the _defaults section and this App\ block, we have now registered *every* class in the src/ directory as a service and told Symfony to autowire each one.

But do we really want *every* class in src/ to be a service? Actually, no. Not *all* classes are services and that's what the exclude: key helps with. For example, the Entity/ directory will eventually store database model classes, which are not services: they're just classes that hold some data.

So we register *everything* in src/ as a service, except for things in these directories. And actually, the exclude key is not *that* important. Heck, you could delete it! If you accidentally registered something as a service that is *not* a service, Symfony will *realize* that when you never use it, and remove it automatically from the container. No big deal.

The point is: everything in src/ is automatically *available* as a service in the container without you needing to think about it.

And... that's really it for the important stuff! The next section registers everything in src/Controller as a service:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 20
# controllers are imported separately to make sure services can be injected
# as action arguments even if you don't extend any base controller class
App\Controller\:
resource: '../src/Controller'
tags: ['controller.service_arguments']
... lines 26 - 33
```

But wait... didn't the section above already do that? Totally! This overrides those in order to add this "tag" thing. This is here to cover an "edge case" that doesn't apply to us. If we deleted this, everything would keep working. So... ignore it.

Now that we understand *how* our services are being added to the container, the config that we added to the bottom of this file will make more sense. Let's talk about it next and then leverage our new knowledge to learn a *way* cooler way to pass the $isDebug flag to MarkdownHelper.

Chapter 14: Binding Global Arguments

In this file, we've registered everything in src/ as a service and activated autowiring on all of them. That's *all* you need... *most* of time. But sometimes, a service needs a bit *more* configuration.

That's what we're doing at the bottom for MarkdownHelper:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 26
# add more service definitions when explicit configuration is needed
# please note that last definitions always *replace* previous ones
App\Service\MarkdownHelper:
bind:
$isDebug: '%kernel.debug%'
```

This service *is* registered above thanks to auto-registration. But down here, we're *overriding* that service: we're *replacing* the auto-registered one with our own so that we can add the extra bind config. This *still* has autowire and autoconfigure enabled on it, thanks to the _defaults section:

33 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
autowire: true # Automatically injects dependencies in your services.
autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 14 - 33
```

But we can now add any *extra* config that we need on this *one* service.

Moving bind to _defaults

When you have an argument that can't be autowired, there's actually another, *easier* way to fix it. Our custom config says that we want the $isDebug argument to MarkdownHelper - that's the third argument - to be set to the kernel.debug parameter. What if we moved this up to the _defaults section?

Do that: copy the bind lines, delete that service *entirely*:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 29
App\Service\MarkdownHelper:
bind:
$isDebug: '%kernel.debug%'
```

And then, up under _defaults, paste:

31 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
autowire: true # Automatically injects dependencies in your services.
autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
bind:
$isDebug: '%kernel.debug%'
... lines 16 - 31
```

Let's see if it works! When we refresh, no errors... and the $isDebug flag that we're dumping is *still* true! This is awesome!

When you add a bind to _defaults, you're setting up a global *convention*: we can now have an $isDebug argument in *any* of our services and Symfony will automatically know to pass the kernel.debug parameter. Thanks to this, we no longer need to override the MarkdownHelper service to add the bind: it's already there! *This* is how I typically handle non-autowireable arguments.

Adding a Type to the Bind

Oh, and if you want, you can add the *type* to the bind - like bool $isDebug:

31 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
... lines 12 - 13
bind:
bool $isDebug: '%kernel.debug%'
... lines 16 - 31
```

This will *still* work because we have the bool type-hint in MarkdownHelper. When we refresh, yep! No errors.

But now, remove the bool type-hint and refresh again. Error!

> Cannot autowire MarkdownHelper: argument $isDebug has no type-hint, you should configure its value explicitly.

Pretty cool. Let's put our bool type-hint back so it matches our bind *exactly*. And... now that it's working, I'll remove the dump() from MarkdownHelper:

33 lines src/Service/MarkdownHelper.php

```
... lines 1 - 7
class MarkdownHelper
{
... lines 10 - 13
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug)
{
... lines 16 - 18
dump($isDebug);
}
... lines 21 - 31
}
```

Here's the big picture: most arguments can be autowired. And when you have one that *can't*, you can set a bind on the specific service *or* set a *global* bind, which is the quickest option.

Next, let's talk about what happens when there are *multiple* services in the container that implement the same interface. How can we choose which one we want?

Chapter 15: Named Autowiring

Let's start with a challenge: let's pretend that, to help us debug something, we want to log some messages from inside MarkdownHelper. Okay: logging is work and work is done by services.... so let's go see if our project has a logger!

Find your terminal and run our favorite command:

```
php bin/console debug:autowiring log
```

There it is! We can use LoggerInterface to get a service whose id is monolog.logger. Notice that there are a *bunch* of loggers listed here. Ignore the others for now: we'll talk about them in a few minutes.

In MarkdownHelper, how do we get access to a service that we need? It's the *same* process every time: add another constructor argument: LoggerInterface $logger. To create a new property and set it, I'm going to use a PhpStorm shortcut. With my cursor on the argument, I'll hit Alt+Enter and select "Initialize properties":

39 lines src/Service/MarkdownHelper.php

```
... lines 1 - 5
use Psr\Log\LoggerInterface;
... lines 7 - 8
class MarkdownHelper
{
... lines 11 - 13
private $logger;
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug,
LoggerInterface $logger)
{
... lines 18 - 20
$this->logger = $logger;
}
... lines 23 - 37
}
```

Nice! But it's not magic: that just created the property and set it down here.

In parse(), let's add some very important code: if stripos($source, 'cat') !== false, then say $this->logger and... let's use, ->info('Meow!'):

39 lines src/Service/MarkdownHelper.php

```
... lines 1 - 8
class MarkdownHelper
{
... lines 11 - 23
public function parse(string $source): string
{
if (stripos($source, 'cat') !== false) {
$this->logger->info('Meow!');
}
... lines 29 - 36
}
}
```

Let's take it for a spin! Move over, refresh... then click any link on the web debug toolbar to jump into the profiler. In the "Logs" section... there's our message!

Multiple Logger Services

The *true* reason we're doing this - other than to practice dependency injection - is to talk about how we can work with the *many* logger services in the container. Symfony's logging library - called Monolog - has this concept of logger channels. They're... kind of like logger categories and their useful because you can send logs from different channels to different files.

This is what you're seeing inside of debug:autowiring: each logger "channel" is actually its own, unique logger *service*. The question is: we know how to get the "main" logger service, but how could we autowire one of these *other* loggers if we needed to?

Creating a Logger Channel

Logging channels are not a *super* important concept - but it's a *great* example of this problem. To see how to handle it, let's add our *own* logger channel. The logger services comes from a bundle called MonologBundle. By adding a little config to that bundle, we can get a shiny new logger channel.

In config/packages/, create a new file called monolog.yaml. Inside say monolog: and below, set channels: to an array. Let's create one new channel called markdown:

3 lines config/packages/monolog.yaml

```
monolog:
channels: ['markdown']
```

By the way, if you're surprised that there was no monolog.yaml file by default, there actually *is*: there's one in the dev/ directory and another in prod/. Loggers behave *pretty* differently in dev versus prod. Thanks to this new file, the markdown channel will exist in *all* environments.

Anyways, *now* find your terminal and run debug:autowiring:

```
php bin/console debug:autowiring log
```

Yes! The bundle created a new service for us called monolog.logger.markdown.

Named Autowiring

So back to my original question: how can we get access to this logger? Well, this is already telling us! This says that if we type-hint an argument with LoggerInterface *and name* the argument $markdownLogger, it will pass us the monolog.logger.markdown service.

Ok, let's try it! Back in MarkdownHelper, rename the argument from $logger to $markdownLogger... and update the variable name below:

39 lines src/Service/MarkdownHelper.php

```
... lines 1 - 8
class MarkdownHelper
{
... lines 11 - 15
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug,
LoggerInterface $markdownLogger)
{
... lines 18 - 20
$this->logger = $markdownLogger;
}
... lines 23 - 37
}
```

Let's see what difference this makes. When we reload, it still works... but open up the profiler and go to the Logs section. Yes! There it is! It says "channel": markdown. For this tutorial, I'm not really concerned about how or *why* we would use a different logger channel. The point is: this *proves* that we just fetched one of the *other* logger services.

The *whole* reason this works is because MonologBundle is smart: it sets up "autowiring aliases" for each channel. Basically, it makes sure that we can autowire the *main* logger with the type-hint *or* any of the other loggers with a type-hint and argument name combination. It sets all of that up *for* us, so we can just take advantage of it.

But what if it *hadn't* done that? Or, what if we needed to access one of the *many* lower-level services in the container that *cannot* be autowired? This is the *last* missing piece of the autowiring puzzle. Let's talk about it next.

Chapter 16: Fetching Non-Autowireable Services

There are *many* services in the container and only a *small* number of them can be autowired. That's by design: most services are pretty low-level and you will rarely need to use them.

But what if you *do* need to use one? How can we do that?

To see how, we're going to use our markdown channel logger as an example. It actually *is* autowireable if you use the LoggerInterface type-hint *and* name your argument $markdownLogger.

But back in MarkdownHelper, to go deeper, let's be complicated and change the argument's name to something else - like $mdLogger:

39 lines src/Service/MarkdownHelper.php

```
... lines 1 - 8
class MarkdownHelper
{
... lines 11 - 15
public function __construct(MarkdownParserInterface $markdownParser, CacheInterface $cache, bool $isDebug,
LoggerInterface $mdLogger)
{
... lines 18 - 20
$this->logger = $mdLogger;
}
... lines 23 - 37
}
```

Excellent! If you refresh the page now, it doesn't break, but if you open the profiler and go to the Logs section, you'll notice that this is using the app channel. That's the "main" logger channel. Because our argument name doesn't match any of the "special" names, it passes us the *main* logger.

So here's the big picture: I want to tell Symfony that the $mdLogger argument to MarkdownHelper should be passed the monolog.logger.markdown service. I don't want any fancy autowiring: I want to tell Symfony *exactly* which service to pass to this argument.

And, there are *two* ways to do this.

Passing Services via Bind

You might guess the first: it's with bind, which works *just* as well to pass services as it does to pass scalar config like parameters:

31 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
... lines 12 - 13
bind:
bool $isDebug: '%kernel.debug%'
... lines 16 - 31
```

First, go copy the *full* class name for LoggerInterface, paste that under bind and add $mdLogger to match our name. But, what value do we *set* this to?

If you look back at debug:autowiring, the id of the service we want to use is monolog.logger.markdown. Copy that and paste it onto our bind.

But... wait. If we stopped now, Symfony would *literally* pass us the string monolog.logger.markdown. That's... not helpful: we want it to pass us the *service* that has this id. To communicate that, prefix the service id with @:

32 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
... lines 12 - 13
bind:
... line 15
Psr\Log\LoggerInterface $mdLogger: '@monolog.logger.markdown'
... lines 17 - 32
```

That's a super-special syntax to *tell* Symfony that we're referring to a *service*.

Let's try this thing! Refresh, then open the Logs section of the profiler. Yes! We're back to logging through the markdown channel!

The bind key is your Swiss Army knife for configuring any argument that can't be autowired.

Adding Autowiring Aliases

But there's *one* other way - besides bind - that we can accomplish this. I'm mentioning it... almost more because it will help you understand how the system works: it's no better or worse than bind.

Copy the LoggerInterface bind line, delete it, move to the bottom of the file, go in four spaces so that we're directly under services and paste:

33 lines config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 31
Psr\Log\LoggerInterface $mdLogger: '@monolog.logger.markdown'
```

That will work too. But... this probably deserves some explanation.

This syntax creates a service "alias": it adds a service to the container whose id is Psr\Log\LoggerInterface $mdLogger. I know, that's a strange id, but it's totally legal. If anyone ever asks for this service, they will *actually* receive the monolog.logger.markdown service.

Why does that help us? I told you earlier that when autowiring sees an argument type-hinted with Psr\Log\LoggerInterface, it looks in the container for a service with that exact id. And, well... that's not *entirely* true. It *does* do that, but only after it *first* looks for a service whose id is the type-hint + the argument name. So yes, it looks for a service whose id is Psr\Log\LoggerInterface $mdLogger. And guess what? We just created a service with that id.

To prove I'm not shouting random information, move over, refresh, and open up the profiler. Yes! It's *still* using the markdown channel. The *super* cool thing is that, back at your terminal, run debug:autowiring log again:

```
php bin/console debug:autowiring log
```

Check it out! Our $mdLogger shows up in the list! By creating that alias, we are doing the *exact* same thing that MonologBundle does internally to set up the *other* named autowiring entries. These are *all* service *aliases*: there is a service with the id of Psr\Log\LoggerInterface $markdownLogger and it's an *alias* to the monolog.logger.markdown service.

Phew! I promise team, that's as deep & dark as you'll probably ever need to get with all this service autowiring business. But as a bonus, the autowiring alias stuff will be *great* small talk for your next Zoom party. Your virtual friends are going to *love* it. I know I would.

Now that we are service *experts*, let's look back at our controller. Because, it's a service too!

Chapter 17: Controllers: Boring, Beautiful Services

Head back to our trusty controller: src/Controller/QuestionController.php. It may be obvious, but it's worth mentioning that controllers are *also* services that live in the container. Yep, they're good, old, normal boring services that behave *just* like anything else. Well except that they have that one *extra* superpower that no other service has: the ability to autowire arguments into its methods. That normally *only* works for the constructor.

Using bind in Controller Arguments

Open up config/services.yaml. A few minutes ago, we added this global "bind" called bool $isDebug:

33 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
... lines 12 - 13
bind:
bool $isDebug: '%kernel.debug%'
... lines 16 - 33
```

Thanks to that, we can add a bool $isDebug argument to the constructor of any service and Symfony will pass us this value. But can we *also* add this argument to a controller method? Absolutely! In the controller, add another with this name: bool $isDebug. I'll dump that down here:

50 lines src/Controller/QuestionController.php

```
... lines 1 - 10
class QuestionController extends AbstractController
{
... lines 13 - 30
public function show($slug, MarkdownHelper $markdownHelper, bool $isDebug)
{
dump($isDebug);
... lines 34 - 47
}
}
```

Now, find you browser, go back to our show page, refresh and... that works wonderfully.

The point is: this ability to autowire arguments into a method is unique to controllers, but it works *exactly* the same as normal, constructor autowiring.

Constructor Injection

And because a controller is a normal, boring service, we can also use *normal* dependency injection. Remove the $isDebug argument. Let's pretend that we want to log something. This time, create a public function __construct() and give it two arguments LoggerInterface $logger and bool $isDebug. Like last time, I'll put my cursor on one of the arguments, hit Alt+Enter, and go to "Initialize properties" to create both of those properties and set them below:

64 lines src/Controller/QuestionController.php

```
... lines 1 - 5
use Psr\Log\LoggerInterface;
... lines 7 - 11
class QuestionController extends AbstractController
{
private $logger;
private $isDebug;
public function __construct(LoggerInterface $logger, bool $isDebug)
{
$this->logger = $logger;
$this->isDebug = $isDebug;
}
... lines 22 - 62
}
```

Down in the show() method, we can say something like if $this->isDebug, then $this->logger->info():

> We are in debug mode!

If you refresh now, open the Profiler, and go to logs... there it is!

So... yeah! Controllers are normal services and, if you want to, you can *entirely* use "normal" dependency injection through the constructor. Heck the biggest reason that autowiring was added to the method was convenience. I *usually* autowire into my methods, but if you need a service in *every* method, using the constructor can help clean things up.

Next, let's talk about the *final* missing piece to configuration: environment variables!

Chapter 18: Environment Variables

One big part of Symfony's configuration system that we have *not* talked about yet is: environment variables. To show them off, let's implement a *real* feature in our app.

Go to sentry.io. If you've never used Sentry before, it's a cloud-based error monitoring tool: it's a really great way to track and debug errors on production. Not that *those* ever happen. Ahem. They also have excellent integration with Symfony.

If you don't already have an account, sign up - it's free. Once you do, you'll end up on a "Getting Started" page that looks something like this. I'll select Symfony from the list. Ok: it wants us to install some sentry/sentry-symfony package.

Installing sentry/sentry-symfony & Contrib Recipes

Before you do, make sure you've committed all your changes to Git.

```
git add .
git commit -m "your commit message here..."
```

I committed before hitting record, so I'm good to go. I like to do this before installing a new package so I can see what its recipe does.

Back on the docs, copy the composer require line, move over, and paste:

```
composer require sentry/sentry-symfony
```

It's downloading some packages and... interesting! It says:

> The package for this recipe comes from the contrib repository, which is open to community contributions. Do you want to execute this recipe?

There are actually *two* places that recipes come from. The first is the main, official recipe repository, which is heavily-guarded for quality. Every recipe we've installed so far has been from that. The second is a "contrib" repository. That repository is *less* guarded for quality to make it easier for recipes from the community to be added, though the recipe still requires approval from a core Symfony member *or* an author of the package itself. The point is: if you're cautious, you can check a contrib recipe before you install it.

I'm going to say yes permanently by saying p. Ok, what did the recipe do? Run:

```
git status
```

It modified the normal stuff - like composer.json, composer.lock, symfony.lock and config/bundles.php because this package contains a bundle: SentryBundle:

15 lines config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 12
Sentry\SentryBundle\SentryBundle::class => ['all' => true],
];
```

Hello Environment Variables & .env

The recipe also *updated* .env and added a new config file. Let's go see what's going on.

First, open up .env and scroll to the bottom. Woh! This has a new section that sets an environment variable called SENTRY_DSN:

26 lines .env

```
... lines 1 - 22
###> sentry/sentry-symfony ###
SENTRY_DSN=
###
```

Environment variables are not a Symfony or PHP concept: they're values that you can pass to *any* process on your computer to configure that process's behavior. Symfony supports *reading* environment variables, which we'll see in a minute. But *setting* them can be a pain: it's different for every operating system. For that reason, when Symfony loads, it reads this file and sets anything here as an environment variable *for* you.

Reading Environment Variables with %env()%

So... if we're *setting* a SENTRY_DSN environment variable... what's *using* that? Go into config/packages/ and open the shiny new sentry.yaml file, which, not surprisingly, configures the new SentryBundle. Check this out: it has a dsn key set to a *very* strange value: %env(SENTRY_DSN)%:

3 lines config/packages/sentry.yaml

```
sentry:
dsn: '%env(SENTRY_DSN)%'
```

This... kind of looks like a parameter, right? It has percent signs on both sides, just like how, in cache.yaml, we referenced the cache_adapter parameter with %cache_adapter%:

20 lines config/packages/cache.yaml

```
framework:
cache:
... lines 3 - 14
app: '%cache_adapter%'
... lines 16 - 20
```

And... it is *sort* of a parameter, but with a special super-power: when you surround something by %env()%, it tells Symfony to read the SENTRY_DSN environment value.

Why Environment Variables?

So... *why* are we setting an environment variable in .env and then reading it here? Well, the SENTRY_DSN string will be a *sensitive* value: if someone got *access* to it, they would be able to send information to our Sentry account.

Look back at the setup guide and skip down to the DSN part. Technically, we *could* copy this value and paste it right into sentry.yaml. The problem is that this file will be committed to git... and it's generally a *bad* idea to commit sensitive values to your repository.

To avoid that, this bundle correctly recommended that we use an environment variable: we'll store the environment variable somewhere *else*, then read it here.

.env & .env.local

And, as we talked about, it *is* possible to set a *real* SENTRY_DSN environment variable on your system. But... since that's a pain, Symfony allows us to *instead* define any environment variables we need in .env if we want to... which we will.

But... this .env file is *also* committed to the repository: you can see that in the terminal if you run:

```
git status
```

So if we pasted the SENTRY_DSN value here, we would have the same problem: the sensitive value would be committed to the repository.

Here's the deal: the .env file is meant to store non-sensitive *default* values for your environment variables - usually values that are good for local development. This works because *after* Symfony loads .env, it looks for *another* file called .env.local.

We don't have that yet, so let's create it: .env.local.

Anything you put in this file will *override* the values in .env. Let's add our real value here: SENTRY_DSN= then paste:

2 lines .env.local

```
SENTRY_DSN=https://7f45741877f3498eab0ae2bee6463d57@o372370.ingest.sentry.io/5186941
```

Perfect! In .env, we set SENTRY_DSN to a non-sensitive default:

26 lines .env

```
... lines 1 - 22
###> sentry/sentry-symfony ###
SENTRY_DSN=
###
```

in this case empty quotes means "don't send data to Sentry": and in .env.local we *override* that to the real value:

2 lines .env.local

```
SENTRY_DSN=https://7f45741877f3498eab0ae2bee6463d57@o372370.ingest.sentry.io/5186941
```

If you're confused *why* this is better, there's *one* more thing I need to tell you. Open up .gitignore: the .env.local file is *ignored* from Git:

18 lines .gitignore

```
... line 1
###> symfony/framework-bundle ###
/.env.local
... lines 4 - 18
```

Check it out: at your terminal, run:

git status

It does *not* see .env.local: our sensitive value will *not* be committed. To see the *final* environment variable values, we can run:

php bin/console about

This gives us a *bunch* of info about our app including, at the bottom, a list of the environment variables being loaded from the .env files. It's working perfectly.

Seeing it Work!

So let's... see if Sentry works! In the show() controller, throw a very realistic new \Exception():

> bad stuff happened!

65 lines src/Controller/QuestionController.php

```
... lines 1 - 11
class QuestionController extends AbstractController
{
... lines 14 - 41
public function show($slug, MarkdownHelper $markdownHelper)
{
if ($this->isDebug) {
$this->logger->info('We are in debug mode!');
}
throw new \Exception('bad stuff happened!');
... lines 48 - 62
}
}
```

When we installed SentryBundle, it *did* add some services to the container. But the main purpose of those services isn't for us to interact with them directly: it's for *them* to hook *into* Symfony. The bundle's services are set up to listen for errors and send them to Sentry.

So all *we* need to do is... refresh! There's our error. Back on Sentry, I should be able to go to sentry.io and... yep! It takes me over to the SymfonyCasts issues and we have a new entry: Exception: bad stuff happened!

Tip

If you don't see your logs showing up in Sentry, there could be some connection error that's being hidden from you. If you want to debug, check out Ryan's comment about this: https://bit.ly/sentry-debug

Next, how do you handle setting environment variables when you deploy? It's time to check out a *cool* new system called the secrets vault.

Chapter 19: The Secrets Vault

How do does *deployment* work with environment variables? Because, in a real app, we're going to have a *bunch* of sensitive environment variables - like database username & password, API keys and more.

Deployment 101

I don't want to get *too* far into the topic of deployment right now, but here's the general idea. Step 1: get your code onto your production machine and run composer install to populate the vendor/ directory. Step 2, *somehow* create a .env.local file with all your production values. And step 3, run:

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

to clear the production cache. The Symfony documentation has more details, but... it's *basically* that simple.

The *trickiest* part is step 2: creating the .env.local file. *Somehow*, your deployment system needs to have access to the sensitive environment variable values so it can populate this file. But, since we're not committing those to our repository... where *should* we store them?

Hello Secrets Vaults

To solve this problem, a concept called a "Secrets Vault" has been invented. And there are actually several *cloud-based* vaults where you can store your secrets in *their* system, then securely read them while you're deploying. Those are *excellent* options.

Symfony *also* comes with its *own* secrets vault, which is *super* cool because it allows us to commit our sensitive values - called secrets - into Git!

Dumping the Sentry Connection Details

To help us see this, in QuestionController::show(), add a third argument: HubInterface $sentryHub. Below, dump($sentryHub):

67 lines src/Controller/QuestionController.php

```
... lines 1 - 6
use Sentry\State\HubInterface;
... lines 8 - 12
class QuestionController extends AbstractController
{
... lines 15 - 42
public function show($slug, MarkdownHelper $markdownHelper, HubInterface $sentryHub)
{
dump($sentryHub);
... lines 46 - 64
}
}
```

The *main* purpose of SentryBundle's services is *not* for us to interact with them directly like this. But this will be a handy way to *quickly* see our SENTRY_DSN value. By the way, I *found* this interface, of course, by using debug:autowiring.

Check it out: back on your browser, I'll close the sentry.io tab and refresh. Down on the web debug toolbar, it dumps a Hub object. If you expand the stack property... and expand again, again, and *again*, there it is! By digging, we can see that our production SENTRY_DSN value is being used.

Creating the Vault

Here's the goal: we're going to *move* the SENTRY_DSN environment value *into* our vault. A vault is basically a collection of *encrypted* values. And in Symfony, you'll have *two* vaults: one for the dev environment - which will contain non-sensitive default values - and a *separate* one for the prod environment with the *real* values.

So, each "secret" will need to be set in *both* vaults. Let's start by putting SENTRY_DSN into the dev vault. How do we do that? Find your terminal and run a shiny new command:

```
php bin/console secrets:set SENTRY_DSN
```

Because we're *in* the dev environment, this will populate the dev vault. And, a good value in dev is actually an empty string. If SENTRY_DSN is empty, Sentry is disabled.

So, just hit "Enter". Ah!

> Warning: No value provided, aborting.

This is a bug in Symfony where it doesn't allow an empty secret. It's already been fixed and an empty value *is* allowed in Symfony 5.0.8. So hopefully, you won't get this.

If you *do*, we can work around it: re-run the command with a - on the end, which tells Symfony to read from STDIN.

```
php bin/console secrets:set SENTRY_DSN -
```

Then, as *crazy* as it sounds, hit Control+D - as in "dog". That was *just* a fancy way to set SENTRY_DSN to an empty string.

The config/secrets/ Public And Private Key Files

Because this was the *first* secret that we set into the dev vault, it automatically *initialized* the vault. It says:

> Sodium keys have been generated at config/secrets/dev/

Let's go check that out! Open up the *new* config/secrets/ directory. *Excellent*: this has a dev/ sub-directory because we just created the dev vault.

The dev.encrypt.public.php file returns the key that's used to add or update secrets:

4 lines config/secrets/dev/dev.encrypt.public.php

```
... lines 1 - 2
return
"\xE2\xDBrR\x9Fz\xC7\xED\x2B\x23\x0F\xB0\x14\x00\xD2d\xC0N\x3AU\xD3M\xC5\xA8\x012\x80\xA2\xA2\xEBFq";
```

It's used to *encrypt* secrets. The dev.decrypt.private.php file does the opposite: its value is used to *decrypt* the secrets so that our app can read them:

4 lines config/secrets/dev/dev.decrypt.private.php

```
... lines 1 - 2
return
"\x86\xB75m\xEFM\x11\x04\x15\xFB\x03\xC8\xF5\xA2b9\xF0eU\xFF\xEA\xD5\x0F\x06\xAC\x05\x89\xC8\x08\x7F\x8A\x9E
```

The decrypt key is *usually* a sensitive value that we would *not* commit to the repository. However, we usually *do* commit the decrypt key for the *dev* vault for two reasons. First, the values in the dev vault are hopefully not very sensitive. And second, we *do* want other developers on our team to be able to decrypt the dev secrets locally. Otherwise... their code won't work.

This directory will also contain one file per secret. We *will* commit this because it's encrypted.

Creating the prod Vault

Let's repeat the same process to put SENTRY_DSN into the prod vault. Run the command again but *also* pass --env=prod:

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

For the value, open .env.local, copy the long DSN string, then paste here. You won't *see* the value because the command is hiding it for security purposes.

And... boom! This generated the prod vault and encrypted the secret. Check out config/secrets/prod. It has the same files, but the output had one extra, *angry* looking note:

> DO NOT COMMIT THE DECRYPTION KEY FOR THE PROD ENVIRONMENT

It's talking about prod.decrypt.private.php. This file *does* need to be here in order for our app to decrypt & read the prod secrets. But we are *not* going to commit it. This is the *one* sensitive value that your deploy script will need to know about.

Tip

Instead of creating the prod.decrypt.private.php file when deploying, you can *also* set the key on a SYMFONY_DECRYPTION_SECRET environment variable. See Production Secrets for more info.

And notice how this is a different color in my editor? That's because... in our .gitignore file, we are *already* ignoring the prod.decrypt.private.php file:

18 lines .gitignore

```
... line 1
###> symfony/framework-bundle ###
... lines 3 - 5
/config/secrets/prod/prod.decrypt.private.php
... lines 7 - 9
###
... lines 11 - 18
```

Committing the Secrets Vaults

Cool! Let's commit them! At your terminal... run:

```
git add config/secrets
```

And then:

```
git status
```

Yes! This added the encrypted secret values themselves, both the encrypt *and* decrypt keys for the dev environment, but *only* the encrypt key for prod. Other developers will be able to add *new* keys for prod, but not *read* them. Isn't encryption cool?

Now that our vaults are set up, let's *use* these secret values in our app! Doing that will be easier than you think. Let's tackle it next.

Chapter 20: Using & Overriding Secrets

We have successfully added the SENTRY_DSN secret value to both the dev and prod vaults.

Listing the Secrets

How can I *prove* that? By running:

php bin/console secrets:list

Because we're in the dev environment, this reads the dev vault. There's our one secret. To see its value, add --reveal:

php bin/console secrets:list --reveal

Behind-the-scenes, that used the dev "decrypt" key to decrypt the value: it's an empty string. Ignore this "local value" thing for a minute.

We can do the same thing for the prod vault by passing --env=prod:

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

Including adding --reveal to *see* the value.

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

Reading Secrets in your App

Ok: because we're in the dev environment, the dev secret - the empty string - is the one that should be used. Refresh the page, check the dump, and expand it a few times. It's *still* using the production value.

Go back into config/packages/sentry.yaml:

3 lines config/packages/sentry.yaml

```
sentry:
dsn: '%env(SENTRY_DSN)%'
```

We're *still* using the syntax for reading *environment* variables. How can we tell it to read the SENTRY_DSN secret instead? Surprise! To tell Symfony to read a SENTRY_DSN *secret*, we use the *exact* same syntax.

Environment Variables vs Secrets

Let me explain: when Symfony sees the %env()% syntax, it *first* looks to see if an environment variable called SENTRY_DSN exists. If it does, it uses it. If there is *not*, it *then* looks for a secret in the vault called SENTRY_DSN. So reading environment variables and secrets uses the same syntax, but environment variables take priority.

This means one important thing: when you identify an environment variable that you want to convert into a secret, you need to *remove* it entirely as an environment variable. Set a value as an environment variable *or* a secret, but not both. Delete the SENTRY_DSN entry from .env and .env.local:

26 lines .env

```
... lines 1 - 22
###> sentry/sentry-symfony ###
SENTRY_DSN=
###
```

2 lines .env.local

```
SENTRY_DSN=https://7f45741877f3498eab0ae2bee6463d57@o372370.ingest.sentry.io/5186941
```

Now Symfony should be read from our dev vault. Refresh... expand the object and... yes! All the values are null! It works!

Let's try out production. Until now, to switch to the prod environment, I've been updating the .env file:

26 lines .env

```
... lines 1 - 15
###> symfony/framework-bundle ###
APP_ENV=dev
... lines 18 - 20
###
... lines 22 - 26
```

But now that we understand .env.local, let's add APP_ENV=prod there instead:

2 lines .env.local

```
APP_ENV=prod
```

Next, clear your cache:

php bin/console cache:clear

Then spin back to your browser and refresh. This time the dump is on top. If I expand it... yes! It's using the *production* values. Booya! That works because my project has the prod decrypt key. If that was *not* there, we would get an error.

Go ahead and take out the APP_ENV= line in .env.local to get back to the dev environment:

2 lines .env.local

```
APP_ENV=prod
```

And in QuestionController, let's cleanup: remove the dump(), the new Exception and the HubInterface argument:

65 lines src/Controller/QuestionController.php

```
... lines 1 - 12
class QuestionController extends AbstractController
{
... lines 15 - 42
public function show($slug, MarkdownHelper $markdownHelper)
{
if ($this->isDebug) {
$this->logger->info('We are in debug mode!');
}
$answers = [
'Make sure your cat is sitting `purrrfectly` still  ',
'Honestly, I like furry shoes better than MY cat',
'Maybe... try saying the spell backwards?',
];
... lines 54 - 62
}
}
```

After this... things are working again.

Overriding Secrets Locally

You are now *ready* to use Symfony's secrets system. But! The fact that environment variables take *precedent* over secrets is something that we can use to our advantage.

Find your terminal and run:

```
php bin/console secrets:list --reveal
```

In the dev environment, the SENTRY_DSN value is set to an empty string. Let's pretend that, while developing, I want to temporarily set SENTRY_DSN to a *real* value so I can test that integration.

We *could* use secrets:set to override the value... but that would update the secrets file... and then we would have to be super careful to avoid committing that change.

There's a better way. In .env.local, set SENTRY_DSN to the real value. Well, I'll put "FOO" here so it's obvious when this value is being used.

Now run that command again:

```
php bin/console secrets:list --reveal
```

The "Value" is still empty quotes, but now it has a "Local Value" set to the string we just used! The "Local Value" is the one that will be used. Why? Because our new environment variable *overrides* the secret: environment variables *always* win over secrets. This "Local Value" is a fancy way of saying that.

I'll take that value out of .env.local so that my secret is once again used.

Next: let's have some fun! We're going to install MakerBundle and start generating some code!

Chapter 21: MakerBundle & Autoconfigure

Congrats on making it *so* far! *Seriously*: your work on this tutorial is going to make *everything* else you do make a *lot* more sense. Now, it's time to celebrate!

One of the *best* parts of Symfony is that it has a *killer* code generator. It's called "MakerBundle"... because shouting "make me a controller!" is more fun than saying "generate me a controller".

Hello MakerBundle

Let's get it installed. Find your terminal and run:

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

We're adding --dev because we won't need the MakerBundle in production, but that's a minor detail.

As I've mentioned *so* many times - sorry - bundles give you services. In this case, MakerBundle doesn't give you services that you will use directly, like in your controller. Nope, it gives you services that power a *huge* list of new *console* commands.

When the install finishes, run:

```
php bin/console make:
```

Woh! Our app suddenly has a *bunch* of commands that start with make:, like make:command, make:controller, make:crud, make:entity, which will be a database entity and more.

Let's try one of these! Let's make our own custom console command!

Making a Custom Console Command

Because, in Symfony, you can hook into *anything*. The bin/console executable

```
php bin/console
```

is no exception: we can add our *own* command here. I do this all the time for CRON jobs or data importing. To get started, run:

```
php bin/console make:command
```

Ok: it asks us for a command name. I like to prefix mine with app: how about app:random-spell. Our new command will output a random magical spell - very useful!

And... we're done! You can see that this created a new src/Command/RandomSpellCommand.php file. Let's go check it out!

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
protected static $defaultName = 'app:random-spell';
protected function configure()
{
$this
->setDescription('Add a short description for your command')
->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
;
}
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$arg1 = $input->getArgument('arg1');
if ($arg1) {
$io->note(sprintf('You passed an argument: %s', $arg1));
}
if ($input->getOption('option1')) {
// ...
}
$io->success('You have a new command! Now make it your own! Pass --help to see your options.');
return 0;
}
}
```

Cool! We can see the name on top, it has a description, some options... and, at the bottom, it ultimately prints a message. We'll start customizing this in a minute.

But before we do that... guess what? The new command already works! Run:

```
php bin/console app:random-spell
```

It's alive! This message is coming from the bottom of the new class.

Service Auto Configuration

But wait: how did Symfony *instantly* see our new command class and start using it? Is it because... it lives in a Command/ directory and Symfony scans for classes that live there? Nope! We could rename this to directory to DefinitelyNoCommandsInHere/ and it would *still* see the command.

The way this works is way cooler. Open up config/services.yaml and look at the _defaults section. We talked about what autowire: true means, but I did *not* explain the purpose of autoconfigure: true:

33 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
autowire: true # Automatically injects dependencies in your services.
autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 14 - 33
```

Because this is below _defaults, autoconfiguration *is* active on *all* of our services, *including* our new command.

When autoconfiguration is enabled for a service, it basically tells Symfony:

> Yo! Please look at the base class or interface of this service and if it *looks* like it should be a command, or an event listener or something *else* that hooks *into* Symfony, please automatically integrate it into that system. Thanks!

In other words, Symfony sees our service, *notices* that it extends Command:

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 4
use Symfony\Component\Console\Command\Command;
... lines 6 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 41
}
```

And thinks:

> I bet this is meant to be a console command. I'll just... hook it into that system automatically.

I *love* this because it means there is *zero* configuration needed to get things working. And you'll see this in a bunch of places in Symfony: you create a class, make it implement an interface and... it'll just start working. We'll see another example in a few minutes with a Twig extension.

Ok! Now that our command is working, let's *customize* it! Playing with console commands is one of my *favorite* things to do in Symfony. Let's go!

Chapter 22: Playing with a Custom Console Command

Let's make our new console command *sing*! Start by giving it a better description: "Cast a random spell!":

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 15
protected function configure()
{
$this
->setDescription('Cast a random spell!')
... lines 20 - 21
;
}
... lines 24 - 41
}
```

For the arguments and options, these describe what you can *pass* to the command. Like, if we configured two arguments, then we could pass two things, like foo and bar after the command. The *order* of arguments is important. Options are things that start with --. Some have values and some don't.

Configuring Arguments & Options

Anyways, let's add one argument called your-name. The argument name is an internal key: you won't see that when using the command. Give it some docs: this is "your name". Let's also add an option called yell so that users can type --yell if they want us to *scream* the spell at them in uppercase:

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 15
protected function configure()
{
$this
->setDescription('Cast a random spell!')
->addArgument('your-name', InputArgument::OPTIONAL, 'Your name')
->addOption('yell', null, InputOption::VALUE_NONE, 'Yell?')
;
}
... lines 24 - 41
}
```

There are more ways to configure this stuff - like you can make an argument optional or required or allow the --yell flag to have a value... but you get the idea.

Executing the Command

When someone *calls* our command, Symfony will run execute(). Let's rename the variable to $yourName and read out the your-name argument:

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 24
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$yourName = $input->getArgument('your-name');
... lines 29 - 40
}
}
```

So *if* the user passes a first argument, we're going to get it here and then, if we have a name, let's say Hi! and then $yourName:

43 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 24
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$yourName = $input->getArgument('your-name');
if ($yourName) {
$io->note(sprintf('Hi %s!', $yourName));
}
... lines 33 - 40
}
}
```

Cool! For the random spell part, I'll paste some code to get it:

55 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 24
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$yourName = $input->getArgument('your-name');
if ($yourName) {
$io->note(sprintf('Hi %s!', $yourName));
}
$spells = [
'alohomora',
'confundo',
'engorgio',
'expecto patronum',
'expelliarmus',
'impedimenta',
'reparo',
];
$spell = $spells[array_rand($spells)];
... lines 45 - 52
}
}
```

Let's check to see if the user passed a --yell flag: if we have a yell option, then $spell = strtoupper($spell):

55 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 24
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$yourName = $input->getArgument('your-name');
if ($yourName) {
$io->note(sprintf('Hi %s!', $yourName));
}
$spells = [
... lines 35 - 41
];
$spell = $spells[array_rand($spells)];
if ($input->getOption('yell')) {
$spell = strtoupper($spell);
}
... lines 49 - 52
}
}
```

Finally, we can use the $io variable to output the spell. This is an instance of SymfonyStyle: it's basically a set or shortcuts for rendering things in a nice way, asking the user questions, printing tables and a lot more. Let's say $io->success($spell):

55 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 11
class RandomSpellCommand extends Command
{
... lines 14 - 24
protected function execute(InputInterface $input, OutputInterface $output): int
{
$io = new SymfonyStyle($input, $output);
$yourName = $input->getArgument('your-name');
if ($yourName) {
$io->note(sprintf('Hi %s!', $yourName));
}
$spells = [
... lines 35 - 41
];
$spell = $spells[array_rand($spells)];
if ($input->getOption('yell')) {
$spell = strtoupper($spell);
}
$io->success($spell);
return 0;
}
}
```

Done! Let's try it! Back at your terminal, start by running the command, but with a --help option:

```
php bin/console app:random-spell --help
```

This tells us everything about our command: the argument, --yell option and a bunch of other options that are built into every command. Try the command with no flags:

```
php bin/console app:random-spell
```

Hello random spell! Try with a name argument:

```
php bin/console app:random-spell Ryan
```

Hi command! And of course we can pass --yell

```
php bin/console app:random-spell Ryan --yell
```

to tell it to scream at us.

There are *many* more fun things you can do with a command, like printing lists, progress bars, asking users questions with auto-complete and more. You'll have no problems figuring that stuff out.

Commands are Services

Oh, but I do want to mention one more thing: a command is a good normal, boring service. I love that! So what if we needed to *access* another service from within a command? Like a database connection? Well... it's the same process as always. Let's log something.

Add a public function __construct with one argument LoggerInterface $logger:

66 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 12
class RandomSpellCommand extends Command
{
... lines 15 - 17
public function __construct(LoggerInterface $logger)
{
... lines 20 - 22
}
... lines 24 - 64
}
```

I'll use my new PhpStorm shortcut - actually I need to hit Escape first - then press Alt+Enter and go to "Initialize properties" to create that property and set it:

66 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 12
class RandomSpellCommand extends Command
{
... line 15
private $logger;
public function __construct(LoggerInterface $logger)
{
$this->logger = $logger;
... lines 21 - 22
}
... lines 24 - 64
}
```

But there is *one* unique thing with commands. The parent Command class has its *own* constructor, which we need to call. Call parent::__construct(): we don't need to pass any arguments to it:

66 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 12
class RandomSpellCommand extends Command
{
... lines 15 - 17
public function __construct(LoggerInterface $logger)
{
$this->logger = $logger;
parent::__construct();
}
... lines 24 - 64
}
```

I can't think of *any* other part of Symfony where this is required - it's a quirk of the command system. Anyways, right before we print the success message, say $this->logger->info() with: "Casting spell" and then $spell:

66 lines src/Command/RandomSpellCommand.php

```
... lines 1 - 12
class RandomSpellCommand extends Command
{
... lines 15 - 33
protected function execute(InputInterface $input, OutputInterface $output): int
{
... lines 36 - 52
$spell = $spells[array_rand($spells)];
if ($input->getOption('yell')) {
$spell = strtoupper($spell);
}
$this->logger->info('Casting spell: '.$spell);
... lines 60 - 63
}
}
```

Let's make sure this works! Run the command again.

```
php bin/console app:random-spell Ryan --yell
```

Cool, that still works. To see if it logged, we can check the log file directly:

```
tail var/log/dev.log
```

I'll move this up a bit... there it is!

Next, let's "make" *one* more thing with MakerBundle. We're going to create our own Twig filter so we can parse markdown through our caching system.

Chapter 23: Making a Twig Extension (Filter)

When we installed KnpMarkdownBundle, it gave us a new service that we used to parse markdown into HTML. But it did more than that. Open up templates/question/show.html.twig and look down where we print out the answers. Because, that bundle *also* gave us a service that provided a custom Twig filter. We could suddenly say {{ answer|markdown }} and that would process the answer through the markdown parser:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 36
<ul class="list-unstyled">
{% for answer in answers %}
<li class="mb-4">
<div class="d-flex justify-content-center">
... lines 41 - 43
<div class="mr-3 pt-2">
{{ answer|markdown }}
... line 46
</div>
... lines 48 - 52
</div>
</li>
{% endfor %}
</ul>
</div>
{% endblock %}
```

The only problem is that this doesn't use our caching system. We created our *own* MarkdownHelper service to handle that:

39 lines src/Service/MarkdownHelper.php

```
... lines 1 - 8
class MarkdownHelper
{
... lines 11 - 23
public function parse(string $source): string
{
if (stripos($source, 'cat') !== false) {
$this->logger->info('Meow!');
}
if ($this->isDebug) {
return $this->markdownParser->transformMarkdown($source);
}
return $this->cache->get('markdown_'.md5($source), function() use ($source) {
return $this->markdownParser->transformMarkdown($source);
});
}
}
```

It uses the markdown parser service but *also* caches the result. Unfortunately, the markdown filter uses the markdown parser from the bundle directly and skips our cool cache layer.

So. What we really want is to have a filter like this that, when used, calls *our* MarkdownHelper service to do its work.

make:twig-extension

Let's take this one piece at a time. First: how can we add custom functions or filters to Twig? Adding features to Twig is work... so it should be *no* surprise that we do this by creating a service. But in order for Twig to understand our service, it needs to look a certain way.

MakerBundle can help us get started. Find your terminal and run:

php bin/console make:

to see our list.

Making the Twig Extension

Let's use make:twig-extension:

php bin/console make:twig-extension

For the name: how about MarkdownExtension. Ding! This created a new src/Twig/MarkdownExtension.php file. Sweet! Let's go open it up:

33 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 8
class MarkdownExtension extends AbstractExtension
{
public function getFilters(): array
{
return [
// If your filter generates SAFE HTML, you should add a third
// parameter: ['is_safe' => ['html']]
// Reference: https://twig.symfony.com/doc/2.x/advanced.html#automatic-escaping
new TwigFilter('filter_name', [$this, 'doSomething']),
];
}
public function getFunctions(): array
{
return [
new TwigFunction('function_name', [$this, 'doSomething']),
];
}
public function doSomething($value)
{
// ...
}
}
```

Just like with our command, in order to hook into Twig, our class needs to implement a specific interface or extend a specific base class. That helps tell us what *methods* our class needs to have.

Right now, this adds a new filter called filter_name and a new function called function_name:

33 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 8
class MarkdownExtension extends AbstractExtension
{
public function getFilters(): array
{
return [
... lines 14 - 16
new TwigFilter('filter_name', [$this, 'doSomething']),
];
}
public function getFunctions(): array
{
return [
new TwigFunction('function_name', [$this, 'doSomething']),
];
}
... lines 27 - 31
}
```

Creative! If someone used the filter in their template, Twig would *actually* call the doSomething() method down here and we would return the final value after applying our filter logic:

33 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 8
class MarkdownExtension extends AbstractExtension
{
... lines 11 - 27
public function doSomething($value)
{
// ...
}
}
```

Autoconfigure!

And guess what? Just like with our command, Twig is *already* aware of our class! To prove that, at your terminal, run:

```
php bin/console debug:twig
```

And if we look up... there it is: filter_name. And the reason that Twig instantly sees our new service is *not* because it lives in a Twig/ directory. It's once again thanks to the autoconfigure feature:

33 lines config/services.yaml

```
... lines 1 - 8
services:
# default configuration for services in *this* file
_defaults:
... line 12
autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 14 - 33
```

Symfony notices that it extends AbstractExtension from Twig:

33 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 4
use Twig\Extension\AbstractExtension;
... lines 6 - 8
class MarkdownExtension extends AbstractExtension
{
... lines 11 - 31
}
```

A class that *all* Twig extensions extend - and thinks:

> Oh! This must be a Twig extension! I'll tell Twig about it

Tip

Technically, all Twig extensions must implement an ExtensionInterface and Symfony checks for this interface for autoconfigure. The AbstractExtension class implements this interface.

Adding the parse_markdown Filter

This means that we're ready to work! Let's call the filter parse_markdown... so it doesn't collide with the other filter. When someone uses this filter, I want Twig to call a new parseMarkdown() method that we're going to add to this class:

26 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 8
class MarkdownExtension extends AbstractExtension
{
public function getFilters(): array
{
return [
... lines 14 - 16
new TwigFilter('parse_markdown', [$this, 'parseMarkdown']),
];
}
... lines 20 - 24
}
```

Remove getFunctions(): we don't need that.

Below, rename doSomething() to parseMarkdown(). And for now, just return TEST:

26 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 8
class MarkdownExtension extends AbstractExtension
{
... lines 11 - 20
public function parseMarkdown($value)
{
return 'TEST';
}
}
```

Let's do this! In show.html.twig, change to use the new parse_markdown filter:

59 lines templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 36
<ul class="list-unstyled">
{% for answer in answers %}
<li class="mb-4">
<div class="d-flex justify-content-center">
... lines 41 - 43
<div class="mr-3 pt-2">
{{ answer|parse_markdown }}
... line 46
</div>
... lines 48 - 52
</div>
</li>
{% endfor %}
</ul>
</div>
{% endblock %}
```

Moment of truth! Spin over to your browser and refresh. Our new filter *works*!

Of course, TEST isn't a *great* answer to a question, so let's make the Twig extension use MarkdownHelper. Once again, we find ourselves in a familiar spot: we're inside of a service and we need access to another service. Yep, it's dependency injection to the rescue! Create the public function __construct() with one argument: MarkdownHelper $markdownHelper. I'll hit Alt+Enter and go to "Initialize properties" to create that property and set it below:

34 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 9
class MarkdownExtension extends AbstractExtension
{
private $markdownHelper;
public function __construct(MarkdownHelper $markdownHelper)
{
$this->markdownHelper = $markdownHelper;
}
... lines 18 - 32
}
```

Inside the method, thanks to our hard work of centralizing our logic into MarkdownHelper, this couldn't be easier: return $this->markdownHelper->parse($value):

34 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 9
class MarkdownExtension extends AbstractExtension
{
... lines 12 - 28
public function parseMarkdown($value)
{
return $this->markdownHelper->parse($value);
}
}
```

$value will be whatever "thing" is being piped into the filter: the answer text in this case.

Ok, it should work! When we refresh... hmm. It's parsing through Markdown but Twig is output escaping it. Twig output escapes everything you print and we fixed this earlier by using the raw filter to tell Twig to *not* do that.

But there's another solution: we can tell Twig that the parse_markdown filter is "safe" and doesn't *need* escaping. To do that, add a 3rd argument to TwigFilter: an array with 'is_safe' => ['html']:

34 lines src/Twig/MarkdownExtension.php

```
... lines 1 - 9
class MarkdownExtension extends AbstractExtension
{
... lines 12 - 18
public function getFilters(): array
{
return [
// If your filter generates SAFE HTML, you should add a third
// parameter: ['is_safe' => ['html']]
// Reference: https://twig.symfony.com/doc/2.x/advanced.html#automatic-escaping
new TwigFilter('parse_markdown', [$this, 'parseMarkdown'], ['is_safe' => ['html']]),
];
}
... lines 28 - 32
}
```

That says: it is safe to print this value into HTML without escaping.

Oh, but in a real app, in parseMarkdown(), I would probably *first* call strip_tags on the $value argument to remove any HTML tags that a bad user may have entered into their answer there. *Then* we can safely use the final HTML.

Anyways, when we move over and refresh, it's *perfect*: a custom Twig filter that parses markdown *and* uses our cache system.

Friends! You rock! Congrats on finishing the Symfony Fundamental course! This was a lot of work and your reward is that *everything* else you do will make more sense and take less time to implement. Nice job.

In the next course, we're going to *really* take things up to the next level by adding a database layer so we can dynamically load *real* questions and real answers. And if *you* have a real question and want a real answer, we're always here for you down in the comments.

Alright friends - seeya next time!