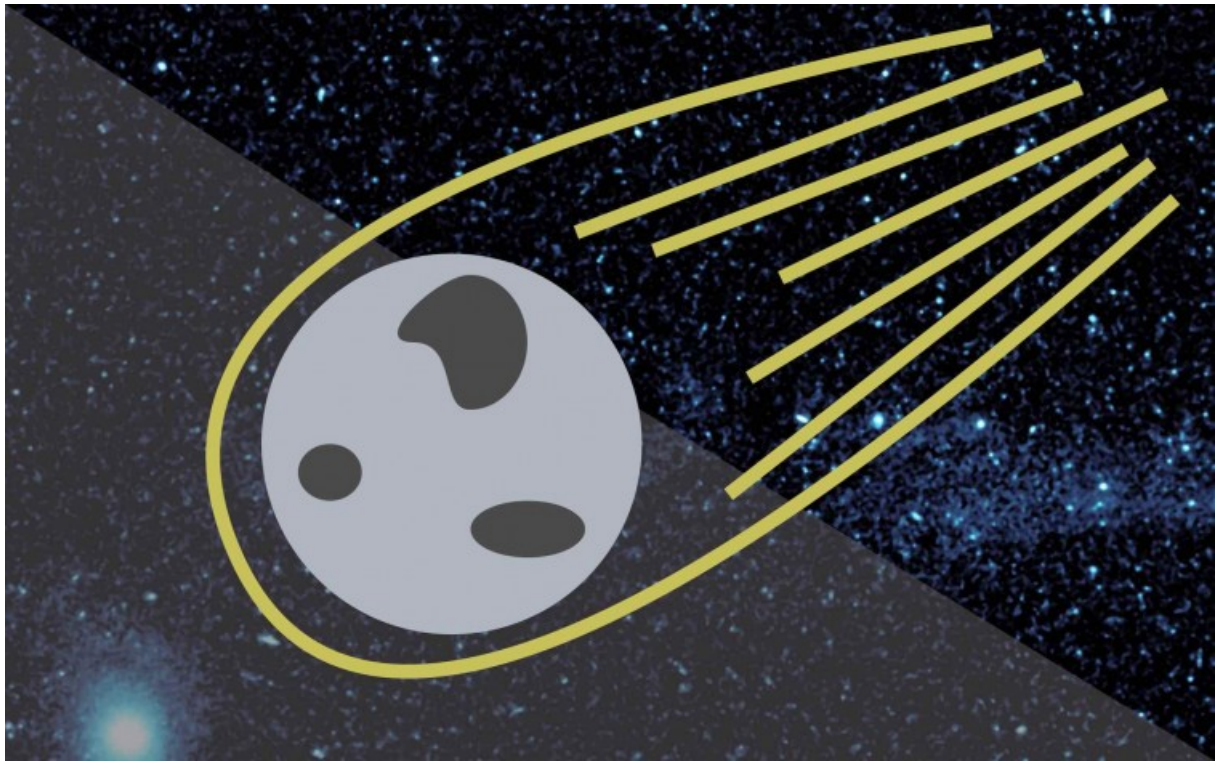


Symfony 4 Fundamentals: Services, Config & Environments



With <3 from SymfonyCasts

Chapter 1: Bundles give you Services


Yo guys! Welcome to episode 2 of our Symfony 4 series! This is a *very* important episode because we're going to take just a *little* bit of time to *really* understand how our app works. I'm talking about how configuration works, the significance of different files and a *whole* lot more.

And yea, there's a reason we're doing this work *now* and not in a distant, future episode: by understanding a few FUNdamentals, *everything* else in Symfony will make a *lot* more sense. So let's dig in and get to work!

[Code Download](#)

As always, if you code along with me, we instantly become best friends. Pow! Download the course code from this page and unzip it. Inside, you'll find a `start/` directory with the same code you see here.

Open the `README.md` file for a whimsical space poem... *and* instructions on how to get the app setup. The last step will be to find a terminal, move into the project and run:



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

to start the built-in PHP web server. Ok! Let's load up our app! Find your browser and go to `http://localhost:8000`. Welcome back to... "The Space Bar": the latest and greatest intergalactic news and sharing site for astronauts and non-human-eating aliens across the universe. Or, it *will* be when we're finished.

[Services: Objects that do Work](#)

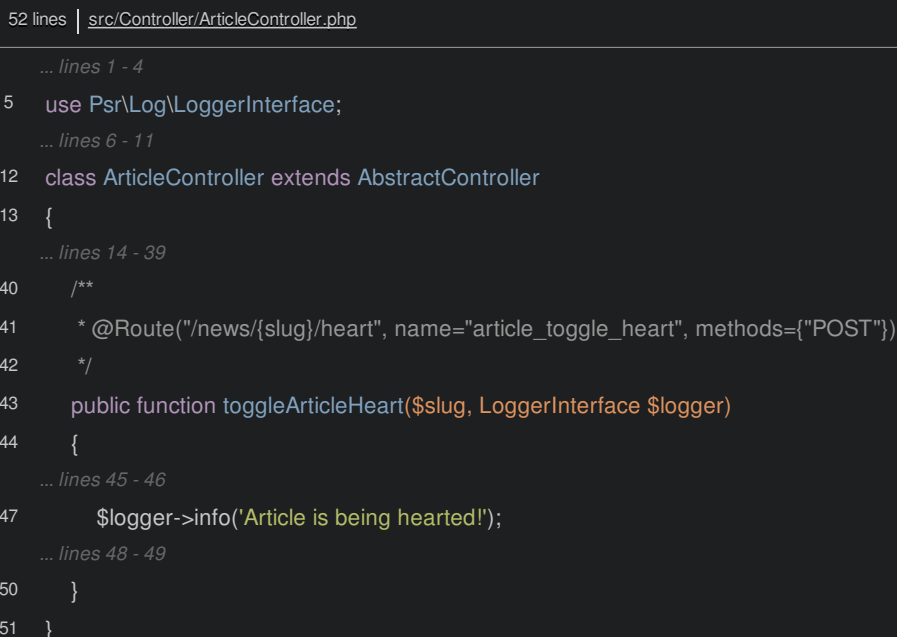
Let's start off stage 2 of our journey with a pop quiz: in episode 1, what did I say was the *most* important part of Symfony? If you answered Fabien... you're technically right, but the *real* answer is: services. Remember: a service is an object that does work: there's a logger service and a Twig service.

To get a list of the services that we can access, you can go to your terminal, open a new tab, and run:



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

For example, to get the logger service, we can use the `LoggerInterface` type-hint:



```
52 lines | src/Controller/ArticleController.php
... lines 1 - 4
5 use Psr\Log\LoggerInterface;
... lines 6 - 11
12 class ArticleController extends AbstractController
13 {
... lines 14 - 39
40 /**
41  * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
42  */
43 public function toggleArticleHeart($slug, LoggerInterface $logger)
44 {
... lines 45 - 46
47     $logger->info('Article is being hearted!');
... lines 48 - 49
50 }
51 }
```

You can see this in our controller: yep, as soon as we add an argument with the `LoggerInterface` type-hint, Symfony knows to pass us the logger service.

Where do Services Come From? Bundles

But... where do these service objects come from? I mean, *somebody* must be creating them in the background for us, right? *Totally!* It's not very important yet, but every service is stored inside *another* object called the *container*. And each service has an internal name, just like routes.

And what exactly puts these services *into* the container? The answer: *bundles*. Bundles are Symfony's *plugin* system. Look inside `config/` and open a `bundles.php` file there:

```
12 lines | config/bundles.php
... lines 1 - 2
3  return [
4      Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
5      Symfony\Bundle\WebServerBundle\WebServerBundle::class => ['dev' => true],
6      Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
7      Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
8      Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
9      Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
10     Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true, 'test' => true],
11 ];
```

Yep, our app has *seven* bundles so far - basically seven plugins. We installed 6 of these in episode 1: the recipe system automatically updates this file when you require a bundle. Sweet!

So let's put this all together: Symfony is *really* nothing more than a collection of services. And bundles are what *actually* prepare those service objects and put them into the container. For example, `MonologBundle` is responsible for giving us the logger service.

Bundles can also do *other* things - like add routes. But they really have one main job: bundles give you services. If you add a bundle, you get more services. And remember, services are *tools*.

So let's install a new bundle and play with some new tools!

Chapter 2: KnpMarkdownBundle & its Services

Scroll down to the "Why do Asteroids Taste like Bacon" article and click to see it. Here's the goal: I want the article body to be processed through *markdown*. Of course, Symfony doesn't come with a markdown-processing service... but there's probably a bundle that *does*! Google for KnpMarkdownBundle and find its GitHub page.

Installing a Bundle

Let's get this installed: copy the composer require line. Then, move over to your terminal, paste and... go!

```
$ composer require knplabs/knp-markdown-bundle
```

Notice that this is a *bundle*: you can see it right in the name. That means it likely contains two things: First, of course, some PHP classes. And second, some configuration that will add one or more new *services* to our app!

And.... installed! It executed one recipe, which made just *one* change:

```
$ git status
```

Yep! It updated bundles.php, which *activates* the bundle:

```
13 lines | config/bundles.php
... lines 1 - 2
3   return [
... lines 4 - 10
11   Knp\Bundle\MarkdownBundle\KnpMarkdownBundle::class => ['all' => true],
12   ];
```

Finding the new Service

So... what's different now? Run:

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

and scroll to the top. Surprise! *We* have a new tool! Actually, there are *two* interfaces you can use to get the *same* markdown service. How do I know these will give us the *same* object? And which should we use? We'll talk about those two questions in the next chapter.

But since it doesn't matter, let's use MarkdownInterface. Open ArticleController. In show(), create a new variable - \$articleContent - and set it to the multiline HEREDOC syntax. I'm going to paste in some fake content. This is the same beefy content that's in the template. In the controller, let's markdownify some stuff! Add some emphasis to jalapeno bacon and let's turn beef ribs into a link to <https://baconipsum.com/>:

72 lines | [src/Controller/ArticleController.php](#)

... lines 1 - 11

```
12 class ArticleController extends AbstractController
13 {
```

... lines 14 - 24

```
25     public function show($slug)
26     {
```

```
27         $comments = [
```

... lines 28 - 30

```
31     ];
```

```
32
```

```
33     $articleContent = <<<<EOF
```

```
34     Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
35     lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
36     labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
37     turkey shank eu pork belly meatball non cupim.
```

```
38
```

```
39     Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
40     laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
41     capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
42     picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
43     occaecat lorem meatball prosciutto quis strip steak.
```

```
44
```

```
45     Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
46     mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
47     strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
48     cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
49     fugiat.
```

```
50     EOF;
```

... lines 51 - 57

```
58     }
```

... lines 59 - 70

```
71 }
```

Pass this into the template as a new articleContent variable:

72 lines | [src/Controller/ArticleController.php](#)

... lines 1 - 11

```
12 class ArticleController extends AbstractController
```

```
13 {
```

... lines 14 - 24

```
25     public function show($slug)
```

```
26     {
```

... lines 27 - 32

```
33         $articleContent = <<<<EOF
```

```
34     Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
```

```
35     lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
```

```
36     labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
```

```
37     turkey shank eu pork belly meatball non cupim.
```

```
38
```

```
39     Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
```

```
40     laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
```

```
41     capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
```

```
42     picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
```

```
43     occaecat lorem meatball prosciutto quis strip steak.
```

```
44
```

```
45     Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
```

```
46     mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
```

```
47     strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
```

```
48     cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
```

```
49     fugiat.
```

```
50     EOF;
```

```
51
```

```
52     return $this->render('article/show.html.twig', [
```

... lines 53 - 55

```
56         'articleContent' => $articleContent,
```

```
57     ]);
```

```
58 }
```

... lines 59 - 70

```
71 }
```

And *now*, in the template, remove *all* the old stuff and just print {{ articleContent }}:

83 lines | [templates/article/show.html.twig](#)

```
... lines 1 - 4
5  {% block body %}
6
7  <div class="container">
8      <div class="row">
9          <div class="col-sm-12">
10             <div class="show-article-container p-3 mt-4">
... lines 11 - 25
26                 <div class="row">
27                     <div class="col-sm-12">
28                         <div class="article-text">
29                             {{ articleContent }}
30                         </div>
31                     </div>
32                 </div>
... lines 33 - 71
72             </div>
73         </div>
74     </div>
75 </div>
76
77 {% endblock %}
... lines 78 - 83
```

Let's try it! Go back to our site and refresh! No surprise: it's the *raw* content. *Now* it's time to process this through Markdown!

[Using the Markdown Service](#)

In ArticleController, tell Symfony to pass us the markdown service by adding a type-hinted argument. Let's use MarkdownInterface: MarkdownInterface \$markdown:

75 lines | [src/Controller/ArticleController.php](#)

```
... lines 1 - 4
5  use Michelf\MarkdownInterface;
... lines 6 - 12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 25
26     public function show($slug, MarkdownInterface $markdown)
27     {
... lines 28 - 60
61     }
... lines 62 - 73
74 }
```

Now, below, \$articleContent = \$markdown-> - we never looked at the documentation to see *how* to use the markdown service... but thanks to PhpStorm, it's pretty self-explanatory - \$markdown->transform(\$articleContent):

75 lines | [src/Controller/ArticleController.php](#)

```
... lines 1 - 4
5 use Michelf\MarkdownInterface;
... lines 6 - 12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 25
26 public function show($slug, MarkdownInterface $markdown)
27 {
... lines 28 - 33
34 $articleContent = <<<EOF
... lines 35 - 50
51 EOF;
52
53 $articleContent = $markdown->transform($articleContent);
... lines 54 - 60
61 }
... lines 62 - 73
74 }
```

Un-escaping Raw HTML

And that's it! Refresh! It works! Um... *kind of*. It *is* transforming our markdown into HTML... but if you look at the HTML source, it's all being *escaped*! Bah!

Actually, this is *awesome*! One of Twig's super-powers - in addition to having very stylish hair - is to automatically escape any variable you render. That means you're protected from XSS attacks without doing *anything*.

If you *do* know that it's safe to print raw HTML, just add `|raw`:

83 lines | [templates/article/show.html.twig](#)

```
... lines 1 - 4
5 {% block body %}
6
7 <div class="container">
8   <div class="row">
9     <div class="col-sm-12">
10       <div class="show-article-container p-3 mt-4">
... lines 11 - 25
26         <div class="row">
27           <div class="col-sm-12">
28             <div class="article-text">
29               {{ articleContent|raw }}
30             </div>
31           </div>
32         </div>
... lines 33 - 71
72       </div>
73     </div>
74   </div>
75 </div>
76
77 {% endblock %}
... lines 78 - 83
```

Try it again! Beautiful!

So here is our first big lesson:

1. Everything in Symfony is done by a service
2. Bundles give us these services... and installing new bundles gives us *more* services.

And 3, Twig clearly gets its hair done by a professional.

Next, let's use a service that's *already* being added to our app by an existing bundle: the *cache* service.

Chapter 3: The Cache Service

Thanks to the 7 bundles installed in our app, we *already* have a bunch of useful services. In fact, Symfony ships with a *killer* cache system out of the box! Run:

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

Scroll to the top. Ah! Check out CacheltemPoolInterface. Notice it's an alias to cache.app. And, further below, there's another called AdapterInterface that's an alias to that same key.

Understanding Autowiring Types & Aliases

Honestly, this can be confusing at first. Internally, each service has a unique name, or "id", just like routes. The internal id for Symfony's cache service is cache.app. That's not very important yet... except that, if you see two entries that are both aliases to the same service, it means that you can use *either* type hint to get the *exact* same object. Yep, both CacheltemPoolInterface and AdapterInterface will cause the *exact* same object to be passed to you.

So... which one should we use? The docs will recommend one, but it technically does *not* matter. The only difference is that PhpStorm may auto-complete different methods for you based on the interface or class you choose. So if it doesn't auto-complete the method you're looking for, try the other interface.

Using Symfony's Cache

Let's use the AdapterInterface. Go back to our controller. Here's our next mission: to cache the markdown transformation: there's *no* reason to do that on every request! At the top of the method, add AdapterInterface \$cache:

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9  use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 66
67     }
... lines 68 - 79
80 }
```

Cool! Let's go use it! Symfony's cache service implements the PHP-standard cache interface, called PSR-6... in case you want Google it and geek-out over the details. But, you probably shouldn't care about this... it just means better interoperability between libraries. So... I guess... yay!

But... there's a downside.... a *dark* side. The standard is very powerful... but kinda weird to use at first. So, watch closely.

Start with \$item = \$cache->getItem(). We need to pass this a cache *key*. Use markdown_ and then md5(\$articleContent):

```

81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9 use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28 {
... lines 29 - 34
35     $articleContent = <<<EOF
... lines 36 - 51
52 EOF;
53
54     $item = $cache->getItem('markdown_'.md5($articleContent));
... lines 55 - 66
67 }
... lines 68 - 79
80 }

```

Excellent! Different markdown content will have a different key. Now, when we call `getItem()` this does *not* actually go and fetch that from the cache. Nope, it just creates a `CacheItem` object in memory that can *help* us fetch and save to the cache.

For example, to check if this key is *not* already cached, use `if (!$item->isHit())`:

```

81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9 use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28 {
... lines 29 - 53
54     $item = $cache->getItem('markdown_'.md5($articleContent));
55     if (!$item->isHit()) {
... lines 56 - 57
58     }
... lines 59 - 66
67 }
... lines 68 - 79
80 }

```

Inside we need to *put* the item into cache. That's a two-step process. Step 1: `$item->set()` and then the value, which is `$markdown->transform($articleContent)`. Step 2: `$cache->save($item)`:

```

81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9  use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 53
54         $item = $cache->getItem('markdown_'.md5($articleContent));
55         if (!$item->isHit()) {
56             $item->set($markdown->transform($articleContent));
57             $cache->save($item);
58         }
... lines 59 - 66
67     }
... lines 68 - 79
80 }

```

I know, I know - it smells a bit over-engineered... but it's *crazy* powerful and *insanely* quick.

Tip

In Symfony 4.1, you will be able to use the `Psr\SimpleCache\CacheInterface` type-hint to get a "simpler" (but less powerful) cache object.

After all of this, add `$articleContent = $item->get()` to fetch the value from cache:

```

81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9  use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 53
54         $item = $cache->getItem('markdown_'.md5($articleContent));
55         if (!$item->isHit()) {
56             $item->set($markdown->transform($articleContent));
57             $cache->save($item);
58         }
59         $articleContent = $item->get();
... lines 60 - 66
67     }
... lines 68 - 79
80 }

```

Debugging the Cache

Ok, let's do this! Find your browser and refresh! Check this out: remember that we have a web debug toolbar icon for the cache! I'll click and open that in a new tab.

Hmm. There are a number of things called "pools". Pools are different cache systems and most are used internally by Symfony. The one *we're* using is called `cache.app`. And, cool! We had a cache "miss" and two calls: we wrote to the cache

and then read from it.

Refresh the page again... and re-open the cache profiler. This time we *hit* the cache. Yes!

And just to make sure we did our job correctly, go back to the markdown content. Let's emphasize "turkey" with two asterisks:

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 8
9 use Symfony\Component\Cache\Adapter\AdapterInterface;
... lines 10 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28 {
... lines 29 - 34
35     $articleContent = <<<EOF
... lines 36 - 38
39 **turkey** shank eu pork belly meatball non cupim.
... lines 40 - 51
52 EOF;
53
54     $item = $cache->getItem('markdown_'.md5($articleContent));
55     if (!$item->isHit()) {
56         $item->set($markdown->transform($articleContent));
57         $cache->save($item);
58     }
59     $articleContent = $item->get();
... lines 60 - 66
67 }
... lines 68 - 79
80 }
```

Refresh again! Yes! The change *does* show up thanks to the new cache key. And this time, in the profiler, we had another miss and write on cache.app.

Check you out! You just learned Symfony's cache service! Add that to your toolkit!

But this leaves some questions: it's great that Symfony *gives* us a cache service... but where is it saving the cache files? And more importantly, what if I need to *change* the cache service to save the cache somewhere else, like Redis? That's next!

Chapter 4: Configuring a Bundle

Let's dump this markdown object: I want to know exactly what this object *actually* is: `dump($markdown);die;`

```
83 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 60
61         dump($markdown);die;
... lines 62 - 68
69     }
... lines 70 - 81
82 }
```

Refresh the article page! Ok, it's an instance of some Max object - probably from the bundle or some library it's using. And it looks like it has a features array where you can turn some features on and off.

So here's the *burning* question: because the bundle *gives* us this service automatically, how can we *configure* it? I mean, what if I want to turn some of these features on or off, or I want to swap the class from Max to a different class from the bundle?

The answer is... science! I mean... configuration! Imagine you're the bundle author: you can probably think of the types of things a user might want to change. So, you setup a simple configuration array that can be used to control those things.

And yea... that's basically how it works, except the config is in YAML.

Dumping Bundle Configuration

And there's an *awesome* way to find out *all* of the configuration options for a bundle without reading the documentation... because /know, we all like to skip reading the docs!

The name of this bundle is KnpMarkdownBundle. At your terminal, use that to dump its config:

```
$ ./bin/console config:dump KnpMarkdownBundle
```

Boom! Say hello to a big YAML example of all of the config options for this bundle. Sometimes, the keys are self-explanatory. But other times - you'll want to cross-reference this with the bundle's docs to find out more.

In this case, down below on the docs, it tells us that the bundle ships with a number of different parsers: it looks like it defaults to this "max" parser: fully-featured, but a bit slow.

Configuring the Parser

To prove we can do it, let's try to change to the "light" parser. According to the docs, we can do that by using the `knp_markdown`, `parser`, `service` config and setting its value to `markdown.parser.light`.

Ok! But... where should this config live? Move over to your project and look in the `config/` directory and then `packages/`. Create a new file called `knp_markdown.yaml`. Then, copy the configuration, paste it here and change the service to the one from the docs: `markdown.parser.light`:

```
4 lines | config/packages/knp_markdown.yaml
1 knp_markdown:
2   parser:
3     service: markdown.parser.light
```

Before we see what that did, find your terminal and run:

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

This... is a bummer. Normally, Symfony is smart-enough to rebuild its cache whenever we change a config file. But... there's currently a bug in Symfony where it does *not* notice *new* config files. So, for now, we need to do this on the rare occasion when we add a new file to config. It should be fixed soon.

So... what did this config change... actually... do? Well, because the purpose of a bundle is to give us services, the purpose of *configuring* a bundle is to *change* how those services behave. That might mean that a service will suddenly use a different class, or that different arguments are passed to a service object. As a user, it doesn't really matter to us: the bundle takes care of the ugly details.

Ok, refresh!

Ah, in this case the change is obvious! Our markdown parser is now an instance of a Light class. Cool!

[More about Bundle Configuration](#)

Now: why did I put this in a file named knp_markdown.yaml? Is that important? Actually, no! As we'll learn soon, Symfony automatically loads *all* files in packages/, and their names are meaningless, technically!

The *super*important part is the root - meaning, non-indented - key: knp_markdown. *Each* file in packages/ configures a different bundle. Any configuration under knp_markdown is passed to the KnpMarkdownBundle. Any config under framework configures FrameworkBundle, which is Symfony's one, "core" bundle:

31 lines | [config/packages/framework.yaml](#)

```
1 framework:
2     secret: '%env(APP_SECRET)%'
3     #default_locale: en
4     #csrf_protection: ~
5     #http_method_override: true
6
7     # Enables session support. Note that the session will ONLY be started if you read or write from it.
8     # Remove or comment this section to explicitly disable session support.
9     session:
10         handler_id: ~
11
12     #esi: ~
13     #fragments: ~
14     php_errors:
15         log: true
16
17     cache:
18         # Put the unique name of your app here: the prefix seed
19         # is used to compute stable namespaces for cache keys.
20         #prefix_seed: your_vendor_name/app_name
21
22         # The app cache caches to the filesystem by default.
23         # Other options include:
24
25         # Redis
26         #app: cache.adapter.redis
27         #default_redis_provider: redis://localhost
28
29         # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30         #app: cache.adapter.apcu
```

And yea, twig configures TwigBundle:

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

```
1 twig:
2     paths: ['%kernel.project_dir%/templates']
3     debug: '%kernel.debug%'
4     strict_variables: '%kernel.debug%'
```

Every bundle has its *own* set of valid config. Heck, let's go check out Twig's config:

```
$ ./bin/console config:dump TwigBundle
```

or we can just the config key instead:

```
$ ./bin/console config:dump twig
```

Say hello to *all* of the valid options for TwigBundle. Of course, these keys are explained more on the TwigBundle docs... but isn't this awesome?

Next: the service container has been hiding something *huge* from us... like "dark matter" huge. Let's find out what it is.

Chapter 5: debug:container & Cache Config

I want to talk more about this key: `markdown.parser.light`:

```
4 lines | config/packages/knp_markdown.yaml
1 knp_markdown:
2   parser:
3     service: markdown.parser.light
```

We got this from the documentation: it told us that there are five different valid values that we can put for the service key.

But, this is more than just a random config key that the bundle author dreamt up. Remember: all services live inside an object called the *container*. And each has an internal name, or id.

It's not *really* important, but it turns out that `markdown.parser.light` is the id of a service in the container! Yep, with this config, we're telling the bundle that when we ask for the Markdown parser - like we are in the controller - it should now pass us the service that has this id.

Go to your terminal and run:

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

And scroll to the top. Check this out! The `MarkdownInterface` is now an alias to `markdown.parser.light`! *Before* the config change, this was `markdown.parser.max`. Yep, this literally means that when we use `MarkdownInterface`, Symfony will pass us a service whose id is `markdown.parser.light`.

Normally, you do *not* need to worry about all of this. I mean, if you just want to use this bundle and configure a few things, follow its docs, make some config tweaks, go on a space walk, and then keep going!

[The Many other Services in the Container](#)

But we're on a quest to *really* understand how things work! Here's the truth, this is *not* a full list of all of the services in the container. Nope, not even close. This time, run:

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

This is actually the *full* list of the *many* services in the container. The service id is on the left, and the class for that object is on the right. Don't worry about the `--show-private` flag: that just makes sure this lists *everything*.

But, in reality, *most* of these services are internal, boring objects that you'll never use. The most important services show up in `debug:autowiring` and are really easy to access.

But yea... you *can* also fetch and use *any* of these services, and sometimes you'll need to. I'll show you how a bit later.

But here are the two big takeaways:

1. There are *many* services in the container and each has an id.
2. The services you'll use 99% of the time show up in `debug:autowiring` and are easy to access.

[Configuring the Cache Object](#)

Let's play with one more object. Instead of dumping `$markdown`, dump the `$cache` object:

```

82 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 53
54         dump($cache);die;
... lines 55 - 67
68     }
... lines 69 - 80
81 }

```

Find your page and refresh! Interesting: it's something called a TraceableAdapter, and, inside, a FilesystemAdapter!

So I guess our cache is being saved to the filesystem... and we can even see *where* in var/cache/dev/pools.

So... how can we configure the cache service? Of course, the *easiest* answer is just to Google its docs. But, we don't even need to do that! The cache service is provided by the FrameworkBundle, which is the one bundle that came automatically with our app.

Debugging your Current Config

Tip

In a recent change to the recipe, the cache config now lives in its own file config/packages/cache.yaml

Open framework.yaml and scroll down:

```

31 lines | config/packages/framework.yaml
1  framework:
... lines 2 - 16
17  cache:
18      # Put the unique name of your app here: the prefix seed
19      # is used to compute stable namespaces for cache keys.
20      #prefix_seed: your_vendor_name/app_name
21
22      # The app cache caches to the filesystem by default.
23      # Other options include:
24
25      # Redis
26      #app: cache.adapter.redis
27      #default_redis_provider: redis://localhost
28
29      # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30      #app: cache.adapter.apcu

```

Hey! This file even comes with documentation about how to configure the cache! Of course, to get an even *bigger* example, we can run:

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

Here's the cache section, with some docs about the different keys. Now, try a slightly *different* command:

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

Instead of dumping *example* config, this is our *current* config! Under cache, there are 6 configured keys. But, you won't see all of these in framework.yaml: these are the bundle's default values. And yea! You can see that this app key is set to cache.adapter.filesystem.

Changing to an APCu Cache

The docs in framework.yaml tell us that, yep, if we want to change the cache system, app is the key we want. Let's uncomment the last one to set app to use APCu: an in-memory cache that's not as awesome as Redis, but easier to install:

```
31 lines | config/packages/framework.yaml
1  framework:
  ... lines 2 - 16
17  cache:
  ... lines 18 - 28
29    # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30    app: cache.adapter.apcu
```

And just like with markdown, cache.adapter.apcu is a service that already exists in the container.

Ok, go back and refresh! Yes! The cache is now using an APCuAdapter internally!

Tip

Fun fact! Running `./bin/console cache:clear` clears Symfony's internal cache that helps your app run. But, it purposely does *not* clear anything that *you* store in cache. If you want to clear that, run `./bin/console cache:pool:clear cache.app`.

Bundle Config: the Good & Bad

So the *great* thing about configuring bundles is that you can make powerful changes with very simple config tweaks. You can also dump your config and Symfony will give you a *great* error if you have any typos.

The *downside* about configuring bundles is that... you *really* need to rely on the debug tools and documentation. I mean, there's no way we could sit here long enough and eventually figure out that the cache system is configured under framework, cache, app: the config structure is totally invented by the bundle.

Let's go back to our controller and remove that dump:

```
82 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
  ... lines 16 - 26
27 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28 {
  ... lines 29 - 53
54     dump($cache);die;
  ... lines 55 - 67
68 }
  ... lines 69 - 80
81 }
```

Make sure everything still works. Perfect! If you get an error, make sure to install the APCu PHP extension.

Next, let's explore Symfony *environments* and *totally* demystify the purpose of each file in config/.

Chapter 6: Explore! Environments & Config Files

Not unlike our space-traveling users, we are *also* pretty adventurous. Sure, *they* might be discovering intelligent life on other planets or exploring binary planets inside the habitable zone. But *we*! We are going to explore the config/ directory and learn *all* of its secrets. Seriously, this is *cool* stuff!

Environment?

We know that Symfony is really just a set of routes and a set of services. And we *also* know that the files in config/packages *configure* those services. But, who *loads* these files? And what is the importance - if *any* - of these sub-directories?

Well, put on your exploring pants, because we're going on a journey!

The code that runs our app is like a machine: it shows articles and will eventually allow people to login, comment and more. The machine always does the same work, but... it needs some *configuration* in order to do its job. Like, where to write log files or what the database name and password are.

And there's other config too, like whether to log *all* messages or just errors, or whether to show a big beautiful exception page - which is great for development - or something aimed at your end-users. Yep, the *behavior* of your app can change based on its config.

Symfony has an *awesome* way of handling this called *environments*. It has two environments out-of-the-box: dev and prod. In the dev environment, Symfony uses a set of config that's... well... great for development: big errors, log everything and show me the web debug toolbar. The prod environment uses a set of config that's optimized for speed, only logs errors, and hides technical info on error pages.

How Environments Work

Ok, I know what you're thinking: this makes sense from a high level... but how does it *work*? Show me the code!

Open the public/ directory and then index.php:

```

40 lines | public/index.php
... lines 1 - 2
3  use App\Kernel;
4  use Symfony\Component\Debug\Debug;
5  use Symfony\Component\Dotenv\Dotenv;
6  use Symfony\Component\HttpFoundation\Request;
7
8  require __DIR__.'/../vendor/autoload.php';
9
10 // The check is to ensure we don't use .env in production
11 if (!isset($_SERVER['APP_ENV'])) {
12     if (!class_exists(Dotenv::class)) {
13         throw new RuntimeException('APP_ENV environment variable is not defined. You need to define environment variables for confi
14     }
15     (new Dotenv())->load(__DIR__.'/../.env');
16 }
17
18 $env = $_SERVER['APP_ENV'] ?? 'dev';
19 $debug = $_SERVER['APP_DEBUG'] ?? ('prod' !== $env);
20
21 if ($debug) {
22     umask(0000);
23
24     Debug::enable();
25 }
26
27 if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ?? false) {
28     Request::setTrustedProxies(explode(',', $trustedProxies), Request::HEADER_X_FORWARDED_ALL ^ Request::HEADER_X_FORV
29 }
30
31 if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? false) {
32     Request::setTrustedHosts(explode(',', $trustedHosts));
33 }
34
35 $kernel = new Kernel($env, $debug);
36 $request = Request::createFromGlobals();
37 $response = $kernel->handle($request);
38 $response->send();
39 $kernel->terminate($request, $response);

```

This is the front controller: a fancy word to mean that it is the *first* file that's executed for *every* page. You don't normally worry about it, but... it's kind of interesting.

It's looking for an environment variable called APP_ENV:

```

40 lines | public/index.php
... lines 1 - 9
10 // The check is to ensure we don't use .env in production
11 if (!isset($_SERVER['APP_ENV'])) {
... lines 12 - 15
16 }
... lines 17 - 40

```

Tip

If you start a new project today, you won't see this APP_ENV logic. It's been moved to a config/bootstrap.php file.

We're going to talk more about environment variables later, but they're just a way to store config values. One confusing thing is that environment *variables* are a *totally* different thing than what we're talking about right now: Symfony environments.

Forget *how* the `$env` variable is set for a moment, and go down to see how it's used:

```
40 lines | public/index.php
... lines 1 - 17
18  $env = $_SERVER['APP_ENV'] ?? 'dev';
... lines 19 - 34
35  $kernel = new Kernel($env, $debug);
... lines 36 - 40
```

Ah! It's passed into some Kernel class! The `APP_ENV` variable is set in a `.env` file, and right now it's set to `dev`. Again, more on environment *variables* later.

Anyways, the string `dev` - is being passed into a Kernel class. The *question* is... what does that *do*?

[Debugging the Kernel Class](#)

Well... good news! That Kernel class is *not* some core part of Symfony. Nope, it lives right inside our app! Open `src/Kernel.php`:

62 lines | [src/Kernel.php](#)

... lines 1 - 2

```
3 namespace App;
4
5 use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
6 use Symfony\Component\Config\Loader\LoaderInterface;
7 use Symfony\Component\DependencyInjection\ContainerBuilder;
8 use Symfony\Component\HttpKernel\Kernel as BaseKernel;
9 use Symfony\Component\Routing\RouteCollectionBuilder;
10
11 class Kernel extends BaseKernel
12 {
13     use MicroKernelTrait;
14
15     const CONFIG_EXTS = '{php,xml,yaml,yml}';
16
17     public function getCacheDir()
18     {
19         ... line 19
20     }
21
22     public function getLogDir()
23     {
24         ... line 24
25     }
26
27     public function registerBundles()
28     {
29         ... lines 29 - 34
30     }
31
32     protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
33     {
34         ... lines 39 - 47
35     }
36
37     protected function configureRoutes(RouteCollectionBuilder $routes)
38     {
39         ... lines 52 - 59
40     }
41 }
```

After some configuration, there are three methods I want to look at. The first is `registerBundles()`:

```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 26
27 public function registerBundles()
28 {
29     $contents = require $this->getProjectDir().'./config/bundles.php';
30     foreach ($contents as $class => $envs) {
31         if (isset($envs['all']) || isset($envs[$this->environment])) {
32             yield new $class();
33         }
34     }
35 }
... lines 36 - 60
61 }

```

This is what loads the config/bundles.php file:

```

13 lines | config/bundles.php
... lines 1 - 2
3 return [
4     Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
5     Symfony\Bundle\WebServerBundle\WebServerBundle::class => ['dev' => true],
6     Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
7     Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
8     Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
9     Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
10    Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true, 'test' => true],
11    Knp\Bundle\MarkdownBundle\KnpMarkdownBundle::class => ['all' => true],
12 ];

```

And check this out: some of the bundles are only loaded in *specific* environments. Like, the WebServerBundle is only loaded in the dev environment:

```

13 lines | config/bundles.php
... lines 1 - 2
3 return [
... line 4
5     Symfony\Bundle\WebServerBundle\WebServerBundle::class => ['dev' => true],
... lines 6 - 11
12 ];

```

And the DebugBundle is similar. Most are loaded in all environments.

The code in Kernel handles this: you can pretty easily guess that `$this->environment` is set to the environment, so, dev!


```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 26
27 public function registerBundles()
28 {
29     $contents = require $this->getProjectDir().'config/bundles.php';
30     foreach ($contents as $class => $envs) {
31         if (isset($envs['all']) || isset($envs[$this->environment])) {
... line 32
33         }
34     }
35 }
... lines 36 - 60
61 }

```

The other two important methods are `configureContainer()`... which basically means "configure services"... and `configureRoutes()`:

```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37 protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38 {
39     $container->setParameter('container.autowiring.strict_mode', true);
40     $container->setParameter('container.dumper.inline_class_loader', true);
41     $confDir = $this->getProjectDir().'config';
42     $loader->load($confDir.'/packages/**/*.self::CONFIG_EXTS', 'glob');
43     if (is_dir($confDir.'/packages/'.$this->environment)) {
44         $loader->load($confDir.'/packages/'.$this->environment.'/**/*.self::CONFIG_EXTS', 'glob');
45     }
46     $loader->load($confDir.'/services'.self::CONFIG_EXTS, 'glob');
47     $loader->load($confDir.'/services_'.$this->environment.self::CONFIG_EXTS, 'glob');
48 }
49
50 protected function configureRoutes(RouteCollectionBuilder $routes)
51 {
52     $confDir = $this->getProjectDir().'config';
53     if (is_dir($confDir.'/routes/')) {
54         $routes->import($confDir.'/routes/**/*.self::CONFIG_EXTS', '/', 'glob');
55     }
56     if (is_dir($confDir.'/routes/'.$this->environment)) {
57         $routes->import($confDir.'/routes/'.$this->environment.'/**/*.self::CONFIG_EXTS', '/', 'glob');
58     }
59     $routes->import($confDir.'/routes'.self::CONFIG_EXTS, '/', 'glob');
60 }
61 }

```

Of course! Because - say it with me now:

Symfony is just a set of services and routes.

Ok, I'll stop jamming that point down your throat.

Package File Loading

Look at configureContainer() first:

```
62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37 protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38 {
39     $container->setParameter('container.autowiring.strict_mode', true);
40     $container->setParameter('container.dumper.inline_class_loader', true);
41     $confDir = $this->getProjectDir().'/config';
42     $loader->load($confDir.'/packages/'.self::CONFIG_EXTS, 'glob');
43     if (is_dir($confDir.'/packages/'.$this->environment)) {
44         $loader->load($confDir.'/packages/'.$this->environment.'/**/*'.self::CONFIG_EXTS, 'glob');
45     }
46     $loader->load($confDir.'/services'.self::CONFIG_EXTS, 'glob');
47     $loader->load($confDir.'/services_'.$this->environment.self::CONFIG_EXTS, 'glob');
48 }
... lines 49 - 60
61 }
```

When Symfony boots, it needs config: it needs to know where to log or how to connect to the database. To get *all* of the config, it calls this *one* method. You can ignore these first two lines: they're internal optimizations.

After, it's uses some sort of \$loader to load configuration files:

```
62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37 protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38 {
... lines 39 - 41
42     $loader->load($confDir.'/packages/'.self::CONFIG_EXTS, 'glob');
43     if (is_dir($confDir.'/packages/'.$this->environment)) {
44         $loader->load($confDir.'/packages/'.$this->environment.'/**/*'.self::CONFIG_EXTS, 'glob');
45     }
46     $loader->load($confDir.'/services'.self::CONFIG_EXTS, 'glob');
47     $loader->load($confDir.'/services_'.$this->environment.self::CONFIG_EXTS, 'glob');
48 }
... lines 49 - 60
61 }
```

This CONFIG_EXTS constant is just a fancy way to load any PHP, XML or YAML files:

```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 14
15     const CONFIG_EXTS = '{php,xml,yaml,yml}';
... lines 16 - 60
61 }

```

First, it loads any files that live *directly* in packages/:

```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37     protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38     {
... lines 39 - 41
42         $loader->load($confDir.'/packages/*'.self::CONFIG_EXTS, 'glob');
... lines 43 - 47
48     }
... lines 49 - 60
61 }

```

But then, it looks to see if there is an environment-specific sub-directory, like packages/dev. And if there *is*, it loads all of *those* files:

```

62 lines | src/Kernel.php
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37     protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38     {
... lines 39 - 42
43         if (is_dir($confDir.'/packages/'.$this->environment)) {
44             $loader->load($confDir.'/packages/'.$this->environment.'/*'.self::CONFIG_EXTS, 'glob');
45         }
... lines 46 - 47
48     }
... lines 49 - 60
61 }

```

Right now, in the dev environment, it will load 5 additional files. The *order* of how this happens is the *key*: any overlapping config in the environment-specific files *override* those from the main files in packages/.

For example, open the main routing.yaml. This is not very important, but it sets some strict_requirements flag to ~... which is null in YAML:

```

4 lines | config/packages/routing.yaml
1 framework:
2     router:
3         strict_requirements: ~

```

But then in the dev environment, that's *overridden*: strict_requirements is set to true:

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

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

To prove it, find your terminal and run:

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

Since we're in the dev environment right now... yep! The `strict_requirements` value is `true`!

This *also* highlights something we talked about earlier: the *names* of the files are *not* important... at *all*. This could be called `hal9000.yaml` and not change a thing. The *important* part is the root key, which tells Symfony which bundle is being configured.

Usually, the filename matches the root key... ya know for sanity. But, it doesn't have to. The organization of these files is subjective: it's meant to make as much sense as possible. The `routing.yaml` file *actually* configures something under the `framework` key.

My *big* point is this: *all* of these files are really part of the same configuration system and, technically, their contents could be copied into one giant file called `my_big_old_config_file.yaml`.

Oh and I said earlier that Symfony comes with only two environments: dev and prod. Well... I lied: there is also a test environment used for automated testing. And... you can create more!

Go *back* to `Kernel.php`. The *last* file that's loaded is `services.yaml`:

```
62 lines | src/Kernel.php
```

```
... lines 1 - 10
11 class Kernel extends BaseKernel
12 {
... lines 13 - 36
37   protected function configureContainer(ContainerBuilder $container, LoaderInterface $loader)
38   {
... lines 39 - 45
46     $loader->load($confDir.'/services'.self::CONFIG_EXTS, 'glob');
47     $loader->load($confDir.'/services_'. $this->environment.self::CONFIG_EXTS, 'glob');
48   }
... lines 49 - 60
61 }
```

More on that file later. It can also have an environment-specific version, like `services_test.yaml`.

Route Loading

And the `configureRoutes()` method is pretty much the same: it automatically loads everything from the `config/routes` directory and then looks for an environment-specific subdirectory:

62 lines | [src/Kernel.php](#)

... lines 1 - 10

```
11 class Kernel extends BaseKernel
```

```
12 {
```

... lines 13 - 49

```
50     protected function configureRoutes(RouteCollectionBuilder $routes)
```

```
51     {
```

```
52         $confDir = $this->getProjectDir().'config';
```

```
53         if (is_dir($confDir.'/routes/')) {
```

```
54             $routes->import($confDir.'/routes/*.self::CONFIG_EXTS, '/', 'glob');
```

```
55         }
```

```
56         if (is_dir($confDir.'/routes/'.$this->environment)) {
```

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

```
58         }
```

```
59         $routes->import($confDir.'/routes'.self::CONFIG_EXTS, '/', 'glob');
```

```
60     }
```

```
61 }
```

So.. yea! All of the files inside config/ either configure services or configure routes. No biggie.

But *now*, with our new-found knowledge, let's tweak the cache service to behave differently in the dev environment. And, let's learn how to *change* to the prod environment.

Chapter 7: Leveraging the prod Environment

Right now, our app is in the dev environment. How can we change it to prod? Just open `.env` and set `APP_ENV` to prod!

```
# .env

# ...
APP_ENV=prod
# ...
```

Then... refresh!

This page may or may *not* work for you. One *big* difference between the dev and prod environments is that in the prod environment, the internal Symfony cache is *not* automatically rebuilt. That's because the prod environment is *wired* for speed.

In practice, this means that whenever you want to switch to the prod environment... like when deploying... you need to run a command:

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

The `bin/console` file *also* reads the `.env` file, so it knows we're in the prod environment.

And *now* when we refresh, it should *definitely* work. And check it out! There's no web debug toolbar. And if you go to a fake page, you get a very boring error page. And yea, you can *totally* customize this: just Google for "Symfony error pages": it's really easy. The point is, this is *not* a big development exception page anymore.

Click back into our article, and then go find its template: `show.html.twig`. Let's change the 3 hours ago to 4 hours ago:

83 lines | [templates/article/show.html.twig](#)

```
... lines 1 - 4
5  {% block body %}
6
7  <div class="container">
8    <div class="row">
9      <div class="col-sm-12">
10       <div class="show-article-container p-3 mt-4">
11         <div class="row">
12           <div class="col-sm-12">
13             ... line 13
14               <div class="show-article-title-container d-inline-block pl-3 align-middle">
15                 ... lines 15 - 17
18                   <span class="pl-2 article-details"> 4 hours ago</span>
19                 ... lines 19 - 22
20               </div>
21             </div>
22           </div>
23         </div>
24       </div>
25     </div>
26     ... lines 26 - 71
72   </div>
73 </div>
74 </div>
75 </div>
76
77 {% endblock %}
... lines 78 - 83
```

Move back to your browser and refresh! Yep! The page did *not* update! That's the behavior I was talking about.

To make it update, find your terminal and run:

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

[The dev and prod Cache Directories](#)

Oh, and check out the var/cache directory. Each environment has its own cache directory: dev and prod. When you run cache:clear, it *basically* just clears the directory and recreates a few files. But there is *another* command:

```
$ ./bin/console cache:warmup
```

This goes a step further and creates *all* of the cache files that Symfony will *ever* need. By running this command when you deploy, the first requests will be much faster. Heck, you can even deploy to a read-only filesystem!

And *now* when you refresh... it works: 4 hours ago.

[Changing the Cache in the dev Environment](#)

Change the environment back to dev:

```
# .env
# ...
APP_ENV=dev
# ...
```

Here's our next challenge. In config/packages/framework.yaml, we configured the cache to use APCu:

```
31 lines | config/packages/framework.yaml
1  framework:
  ... lines 2 - 16
17  cache:
  ... lines 18 - 28
29  # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30  app: cache.adapter.apcu
```

What if we *did* want to use this for production, but in the dev environment, we wanted to use the *filesystem* cache instead for simplicity. How could we do that?

We *already* know the answer! We just need to override this key inside the dev environment. Create a new file in config/packages/dev called framework.yaml... though technically, this could be called anything. We just need the same keys: framework, cache, app. Add those, but now set app to cache.adapter.filesystem, which was the *original* value:

```
4 lines | config/packages/dev/framework.yaml
1  framework:
2  cache:
3  app: cache.adapter.filesystem
```

Let's see if it worked! Open ArticleController and dump the \$cache object so we can see what it looks like:

```

82 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 53
54         dump($cache);die;
... lines 55 - 67
68     }
... lines 69 - 80
81 }

```

And, refresh! Yes! It's using the FilesystemAdapter! What about the prod environment? In .env, change APP_ENV back to prod:

```

# .env
# ...
APP_ENV=prod
# ...

```

But don't forget to clear the cache:

```

$ ./bin/console cache:clear

```

The warmup part is optional. Refresh! Yea! In the prod environment, the cache *still* uses APCu.

Change the environment back to dev:

```

# .env
# ...
APP_ENV=dev
# ...

```

In reality, you won't spend much time in the prod environment... it mostly exists for when you deploy to production.

Let's also remove the dump():

```

82 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28     {
... lines 29 - 53
54         dump($cache);die;
... lines 55 - 67
68     }
... lines 69 - 80
81 }

```

Oh man, with environments behind us, we can jump into the *heart* of our tutorial.. the thing I have been *waiting* to do: create our *own* services.

Chapter 8: Creating Services!

Open ArticleController and find the show() action:

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 26
27 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache)
28 {
... lines 29 - 53
54     $item = $cache->getItem('markdown_'.md5($articleContent));
55     if (!$item->isHit()) {
56         $item->set($markdown->transform($articleContent));
57         $cache->save($item);
58     }
59     $articleContent = $item->get();
... lines 60 - 66
67 }
... lines 68 - 79
80 }
```

I think it's time to move our markdown & caching logic to a different file. Why? Two reasons. First, this method is getting a bit long and hard to read. And second, we can't *re-use* any of this code when it's stuck in our controller. And... bonus reason! If you're into unit testing, this code cannot be tested.

On the surface, this is the *oldest* trick in the programming book: if you want to re-use some code, move it into its own function. But, what we're about to do will form the *cornerstone* of almost *everything* else in Symfony.

Create the Service Class

Instead of moving this code to a *function*, we're going to create a new class and move into a new *method*. Inside *src/*, create a new directory called *Service*. And then a new PHP class called *MarkdownHelper*:

```
18 lines | src/Service/MarkdownHelper.php
... lines 1 - 2
3 namespace App\Service;
4
5 class MarkdownHelper
6 {
... lines 7 - 16
17 }
```

The name of the directory - *Service* - and the name of the class are not important at all: you can put your code *wherever* you want. The power!

Inside, let's add a public function called, how about, *parse()*: with a string *\$source* argument that will *return* a string:

```

18 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 class MarkdownHelper
6 {
7     public function parse(string $source): string
8     {
... lines 9 - 15
16     }
17 }

```

And... yea! Let's just copy our markdown code from the controller and paste it here!

```

18 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 class MarkdownHelper
6 {
7     public function parse(string $source): string
8     {
9         $item = $cache->getItem('markdown_'.md5($source));
10        if (!$item->isHit()) {
11            $item->set($markdown->transform($source));
12            $cache->save($item);
13        }
14
15        return $item->get();
16    }
17 }

```

I know, it's not going to work yet - we've got undefined variables. But, worry about that later. Return the string at the bottom:

```

18 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 class MarkdownHelper
6 {
7     public function parse(string $source): string
8     {
... lines 9 - 14
15        return $item->get();
16    }
17 }

```

And... congrats! We just created our first service! What? Remember, a service is just a class that does work! And yea, this class does work! The *really* cool part is that we can *automatically* autowire our new service.

Find your terminal and run:

Tip

Since *Symfony 4.2* this command only shows service aliases. If you want to see all the services you can pass a `--all` option.

```

$ ./bin/console debug:autowiring

```

Scroll up. Boom! There is MarkdownHelper. It already lives in the container, just like all the core services. That means, in ArticleController, instead of needing to say `new MarkdownHelper()`, we can autowire it: add another argument:

MarkdownHelper \$markdownHelper:

```
77 lines | src/Controller/ArticleController.php
... lines 1 - 4
5 use App\Service\MarkdownHelper;
... lines 6 - 14
15 class ArticleController extends AbstractController
16 {
... lines 17 - 24
25 /**
26  * @Route("/news/{slug}", name="article_show")
27  */
28 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache, MarkdownHelper $markdownHelper)
29 {
... lines 30 - 62
63 }
... lines 64 - 75
76 }
```

Below, *simplify*: \$articleContent = \$markdownHelper->parse(\$articleContent):

```
77 lines | src/Controller/ArticleController.php
... lines 1 - 4
5 use App\Service\MarkdownHelper;
... lines 6 - 14
15 class ArticleController extends AbstractController
16 {
... lines 17 - 24
25 /**
26  * @Route("/news/{slug}", name="article_show")
27  */
28 public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache, MarkdownHelper $markdownHelper)
29 {
... lines 30 - 35
36     $articleContent = <<<EOF
... lines 37 - 52
53 EOF;
54
55     $articleContent = $markdownHelper->parse($articleContent);
... lines 56 - 62
63 }
... lines 64 - 75
76 }
```

Ok, let's try it! Refresh! We expected this:

Undefined variable \$cache

Inside MarkdownHelper. But hold on! This *proves* that Symfony's container is *instantiating* the MarkdownHelper and then passing it to us. So cool!

[Dependency Injection: The Wrong Way First](#)

In MarkdownHelper, oh, update the code to use the \$source variable:

```

18 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 class MarkdownHelper
6 {
7     public function parse(string $source): string
8     {
9         $item = $cache->getItem('markdown_'.md5($source));
10        if (!$item->isHit()) {
11            $item->set($markdown->transform($source));
12            $cache->save($item);
13        }
14
15        return $item->get();
16    }
17 }

```

Here's the problem: MarkdownHelper *needs* the cache and markdown services. To say it differently, they're *dependencies*. So how can we *get* them from here?

Symfony follows object-orientated best practices... which means that there's no way to *magically* fetch them out of thin air. But that's no problem! If you *ever* need a service or some config, just *pass them in*.

The easiest way to do this is to add them as arguments to parse(). I'll show you a different solution in a minute - but let's get it working. Add AdapterInterface \$cache and MarkdownInterface \$markdown:

```

21 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 use Michelf\MarkdownInterface;
6 use Symfony\Component\Cache\Adapter\AdapterInterface;
7
8 class MarkdownHelper
9 {
10    public function parse(string $source, AdapterInterface $cache, MarkdownInterface $markdown): string
11    {
12        ... lines 12 - 18
19    }
20 }

```

If you try it now... it fails:

Too few arguments passed to parse(): 1 passed, 3 expected.

This makes sense! In ArticleController, we are calling parse():

```

77 lines | src/Controller/ArticleController.php
... lines 1 - 14
15 class ArticleController extends AbstractController
16 {
17    ... lines 17 - 27
28    public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache, MarkdownHelper $markdownHelper)
29    {
30        ... lines 30 - 54
55        $articleContent = $markdownHelper->parse($articleContent);
56        ... lines 56 - 62
63    }
64    ... lines 64 - 75
76 }

```

This is important: that whole autowiring thing works for controller actions, because that is a unique time when *Symfony* is calling our method. But everywhere else, it's good old-fashioned object-oriented coding: if we call a method, we need to pass all the arguments.

No problem! Add `$cache` and `$markdown`:

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 14
15 class ArticleController extends AbstractController
16 {
... lines 17 - 27
28     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache, MarkdownHelper $markdownHelper)
29     {
... lines 30 - 54
55         $articleContent = $markdownHelper->parse(
56             $articleContent,
57             $cache,
58             $markdown
59         );
... lines 60 - 66
67     }
... lines 68 - 79
80 }
```

And... refresh! It works! We *just* isolated our code into a re-usable service. We *rule*. Go high-five some strangers!

Proper Dependency Injection

Then come back! Because there's a *much* better way to do all of this. Whenever you have a service that depends on *other* services, like `$cache` or `$markdown`, instead of passing those in as arguments to the individual *method*, you should pass them via a *constructor*.

Let me show you: create a public function `__construct()`. Next, move the two arguments into the constructor, and create properties for each: `private $cache`; and `private $markdown`:

```
30 lines | src/Service/MarkdownHelper.php
... lines 1 - 7
8 class MarkdownHelper
9 {
10     private $cache;
11     private $markdown;
12
13     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown)
14     {
... lines 15 - 16
17     }
18
19     public function parse(string $source): string
20     {
... lines 21 - 27
28     }
29 }
```

Inside the constructor, set these: `$this->cache = $cache` and `$this->markdown = $markdown`:

```

30 lines | src/Service/MarkdownHelper.php
... lines 1 - 7
8  class MarkdownHelper
9  {
10     private $cache;
11     private $markdown;
12
13     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown)
14     {
15         $this->cache = $cache;
16         $this->markdown = $markdown;
17     }
18
19     ... lines 18 - 28
29 }

```

By putting this in the constructor, we're basically saying that *whoever* uses the MarkdownHelper is *required* to pass us a cache object and a markdown object. From the perspective of this class, we don't care *who* uses us, but we *know* that they will be *forced* to pass us our dependencies.

Thanks to that, in parse() we can safely use \$this->cache and \$this->markdown:

```

30 lines | src/Service/MarkdownHelper.php
... lines 1 - 7
8  class MarkdownHelper
9  {
10     private $cache;
11     private $markdown;
12
13     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown)
14     {
15         $this->cache = $cache;
16         $this->markdown = $markdown;
17     }
18
19     public function parse(string $source): string
20     {
21         $item = $this->cache->getItem('markdown_'.md5($source));
22         if (!$item->isHit()) {
23             $item->set($this->markdown->transform($source));
24             $this->cache->save($item);
25         }
26
27     ... lines 26 - 27
28 }
29 }

```

One of the advantages of passing dependencies through the constructor is that it's easier to call our methods: we *only* need to pass arguments that are specific to that method - like the article content:

```

77 lines | src/Controller/ArticleController.php
... lines 1 - 14
15 class ArticleController extends AbstractController
16 {
... lines 17 - 27
28     public function show($slug, MarkdownInterface $markdown, AdapterInterface $cache, MarkdownHelper $markdownHelper)
29     {
... lines 30 - 54
55         $articleContent = $markdownHelper->parse($articleContent);
... lines 56 - 62
63     }
... lines 64 - 75
76 }

```

And, hey! We can also remove the extra controller arguments. And, on top, we don't *need* to, but let's remove the old use statements:

```

75 lines | src/Controller/ArticleController.php
... lines 1 - 4
5 use App\Service\MarkdownHelper;
6 use Psr\Log\LoggerInterface;
7 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
8 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
9 use Symfony\Component\HttpFoundation\JsonResponse;
10 use Symfony\Component\HttpFoundation\Response;
11 use Twig\Environment;
12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 25
26     public function show($slug, MarkdownHelper $markdownHelper)
27     {
... lines 28 - 60
61     }
... lines 62 - 73
74 }

```

[Configuring the Constructor Args?](#)

But there's still one *big* question! How did nobody notice that there was a thermal exhaust pipe that would cause the whole Deathstar to explode? And also, because the container is responsible for *instantiating* MarkdownHelper, how will it know what values to pass? Don't we need to somehow *tell* it that it needs to pass the cache and markdown services as arguments?

Actually, no! Move over to your browser and refresh. It just *works*.

Black magic! Well, not really. When you create a service class, the arguments to its constructor are *autowired*. That means that we can use any of the classes or interfaces from debug:autowiring as type-hints. When Symfony creates our MarkdownHelper:

75 lines | [src/Controller/ArticleController.php](#)

... lines 1 - 4

5 use App\Service\MarkdownHelper;

... lines 6 - 12

13 class ArticleController extends AbstractController

14 {

... lines 15 - 25

26 public function show(\$slug, MarkdownHelper \$markdownHelper)

27 {

... lines 28 - 60

61 }

... lines 62 - 73

74 }

It knows what to do!

Yep, we just organized our code into a brand new service and touched *zero* config files. This is huge!

Next, let's get smarter, and find out how we can access core services that *cannot* be autowired.

Chapter 9: Using Non-Standard Services: Logger Channels

Let's add some logging to MarkdownHelper. As always, we just need to find *which* type-hint to use. Run:

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

And look for log. We've seen this before: `LoggerInterface`. To get this in `MarkdownHelper`, just add a *third* argument: `LoggerInterface $logger`:

```
37 lines | src/Service/MarkdownHelper.php
... lines 1 - 5
6  use Psr\Log\LoggerInterface;
... lines 7 - 8
9  class MarkdownHelper
10 {
... lines 11 - 14
15     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $logger)
16     {
... lines 17 - 19
20     }
... lines 21 - 35
36 }
```

Like before, we need to create a new property and set it below. Great news! PhpStorm has a *shortcut* for this! With your cursor on `$logger`, press `Alt+Enter`, select "Initialize fields" and hit OK:

```
37 lines | src/Service/MarkdownHelper.php
... lines 1 - 5
6  use Psr\Log\LoggerInterface;
... lines 7 - 8
9  class MarkdownHelper
10 {
... lines 11 - 12
13     private $logger;
14
15     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $logger)
16     {
... lines 17 - 18
19         $this->logger = $logger;
20     }
... lines 21 - 35
36 }
```

Awesome! Down in `parse()`, if the source contains the word `bacon`... then of course, we need to know about that! Use `$this->logger->info('They are talking about bacon again!')`:

```

37 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 21
22     public function parse(string $source): string
23     {
24         if (stripos($source, 'bacon') !== false) {
25             $this->logger->info('They are talking about bacon again!');
26         }
... lines 27 - 34
35     }
36 }

```

Ok, try it! This article *does* talk about bacon. Refresh! To see if it logged, open the profiler and go to "Logs". Yes! Here is our message. I *love* autowiring.

[The Other Loggers](#)

Go back to your terminal. The debug:autowiring output say that LoggerInterface is an alias to monolog.logger. That is the *id* of the service that's being passed to us. Fun fact: you can get a bit more info about a service by running:

```
$ ./bin/console debug:container monolog.logger
```

This is cool - but you could also learn a lot by dumping it. Anyways, we *normally* use debug:container to list *all* of the services in the container. But we can also get a filtered list. Let's find *all* services that contain the word "log":

```
$ ./bin/console debug:container --show-private log
```

There are about 6 services that I'm *really* interested in: these monolog.logger. something services.

[Logging Channels](#)

Here's what's going on. Symfony uses a library called Monolog for logging. And Monolog has a feature called *channels*, which are kind of like *categories*. Instead of having just *one* logger, you can have *many* loggers. Each has a unique name - called a channel - and each can do *totally* different things with their logs - like write them to different log files.

In the profiler, it even shows the channel. Apparently, the *main* logger uses a channel called app. But other parts of Symfony are using other channels, like request or event. If you look in config/packages/dev/monolog.yaml, you can see different *behavior* based on the channel:

```

20 lines | config/packages/dev/monolog.yaml
1  monolog:
2    handlers:
3      main:
4        type: stream
5        path: "%kernel.logs_dir%/%kernel.environment%.log"
6        level: debug
7        channels: ["!event"]
8      # uncomment to get logging in your browser
9      # you may have to allow bigger header sizes in your Web server configuration
10     #firephp:
11     #   type: firephp
12     #   level: info
13     #chrome.php:
14     #   type: chrome.php
15     #   level: info
16     console:
17       type: console
18       process_psr_3_messages: false
19       channels: ["!event", "!doctrine", "!console"]

```

For example, most logs are saved to a dev.log file. But, thanks to this channels: ["!event"] config, which means "not event", anything logged to the "event" logger is *not* saved to this file.

This is a *really* cool feature. But mostly... I'm telling you about this because it's a *great* example of a new problem: how could we access one of these *other* Logger objects? I mean, when we use the LoggerInterface type-hint, it gives us the *main* logger. But what if we need a *different* Logger, like the "event" channel logger?

[Creating a new Logger Channel](#)

Actually, let's create our own *new* channel called markdown. I want anything in this channel to log to a different file.

To do this, inside config/packages, create a file: monolog.yaml. Monolog is interesting: it doesn't normally have a main configuration file: it only has environment-specific config files for dev and prod. That makes sense: we log things in completely different ways based on the environment.

But *we're* going to add some config that will create a *new* channel, and we want that to exist in *all* environments. Add monolog, then channels set to [markdown]:

```

3 lines | config/packages/monolog.yaml
1  monolog:
2    channels: [markdown]

```

That's it!

Because of a Symfony bug - which, is *now* fixed (woo!) - but won't be available until the next version - Symfony 4.0.5 - we need to clear the cache manually when adding a new config file:

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

As *soon* as that finishes, run debug:container again:

```
$ ./bin/console debug:container log
```

Yea! Suddenly we have a new logger service - monolog.logger.markdown! So cool.

Go back to the "dev" monolog.yaml file. Copy the first log handler, paste, and give it a key called `markdown_logging` - that's just a meaningless internal name. Change the path to `markdown.log` and *only* log the markdown channel:

```
25 lines | config/packages/dev/monolog.yaml
1  monolog:
2    handlers:
... lines 3 - 7
8    markdown_logging:
9      type: stream
10     path: "%kernel.logs_dir%/markdown.log"
11     level: debug
12     channels: ["markdown"]
... lines 13 - 25
```

Ok! If you go to your browser *now* and refresh... it *does* work. But if you check the logs, we are - of course - *still* logging to the app channel Logger. Yep, there's no `markdown.log` file yet.

Fetching a Non-Standard Service

So how can we tell Symfony to *not* pass us the "main" logger, but instead to pass us the `monolog.logger.markdown` service? This is our *first* case where autowiring doesn't work.

That's *no* problem: when autowiring doesn't do what you want, just... correct it! Open `config/services.yaml`. Ignore all of the configuration on top for now. But notice that we're under a key called `services`. Yep, *this* is where we configure how *our* services work. At the bottom, add `App\Service\MarkdownHelper`, then below it, arguments:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 25
26  App\Service\MarkdownHelper:
27    arguments:
... lines 28 - 32
```

The argument we want to configure is called `$logger`. Use that here: `$logger`. We are telling the container what *value* to pass to that argument. Use the service id: `monolog.logger.markdown`. Paste!

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 25
26  App\Service\MarkdownHelper:
27    arguments:
28      $logger: 'monolog.logger.markdown'
... lines 29 - 32
```

Find your browser and... try it! Bah! A big error:

Argument 3 passed to `MarkdownHelper::__construct()` must implement `LoggerInterface`, string given.

Ah! It's *totally* legal to set an argument to a string value. But we don't want to pass the *string* `monolog.logger.markdown`! We want to pass the *service* with this id!

To do that, use a special Symfony syntax: add an `@` symbol:

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

... lines 1 - 4

5 services:

... lines 6 - 25

26 App\Service\MarkdownHelper:

27 arguments:

28 \$logger: '@monolog.logger.markdown'

... lines 29 - 32

This tells Symfony not to pass us that *string*, but to pass us the *service* with that id.

Try it again! It works! Check out the var/log directory... boom! We have a markdown.log file!

Next, I'll show you an even *cooler* way to configure this. And we'll learn more about what all this config in services.yaml does.

Chapter 10: services.yaml & the Amazing bind

When Symfony loads, it needs to figure out *all* of the services that should be in the container. Most of the services come from external bundles. But we *now* know that we can add our *own* services, like MarkdownHelper. We're unstoppable!

All of that happens in services.yaml under the services key:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 32
```

This is *our* spot to add *our* services. And I want to demystify what the config in this file *actually* does:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
6      # default configuration for services in *this* file
7      _defaults:
8          autowire: true    # Automatically injects dependencies in your services.
9          autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
10         public: false     # Allows optimizing the container by removing unused services; this also means
11                             # fetching services directly from the container via $container->get() won't work.
12                             # The best practice is to be explicit about your dependencies anyway.
13
14     # makes classes in src/ available to be used as services
15     # this creates a service per class whose id is the fully-qualified class name
16     App\:
17         resource: '../src/*'
18         exclude: '../src/{Entity,Migrations,Tests}'
19
20     # controllers are imported separately to make sure services can be injected
21     # as action arguments even if you don't extend any base controller class
22     App\Controller\:
23         resource: '../src/Controller'
24         tags: ['controller.service_arguments']
... lines 25 - 32
```

All of this - except for the MarkdownHelper stuff we *just* added - comes standard with every new Symfony project.

Understanding _defaults

Let's start with _defaults:

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

... lines 1 - 4

```
5 services:
6     # default configuration for services in *this* file
7     _defaults:
8         autorewire: true     # Automatically injects dependencies in your services.
9         autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
10        public: false        # Allows optimizing the container by removing unused services; this also means
11                             # fetching services directly from the container via $container->get() won't work.
12                             # The best practice is to be explicit about your dependencies anyway.
... lines 13 - 32
```

This is a special key that sets *default* config values that should be applied to *all* services that are registered in this *file*.

For example, `autorewire: true` means that any services registered in this file should have the autowiring behavior turned on:

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

... lines 1 - 4

```
5 services:
6     # default configuration for services in *this* file
7     _defaults:
8         autorewire: true     # Automatically injects dependencies in your services.
... lines 9 - 32
```

Because yea, you can *actually* set autowiring to false if you want. In fact, you could set autowiring to false on just *one* service to override these defaults:

```
services:
  _defaults:
    autorewire: true
  # ...
  App\Service\MarkdownHelper:
    autorewire: false
  # ...
```

The `autoconfigure` option is something we'll talk about during the last chapter of this course - but it's not too important:

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

... lines 1 - 4

```
5 services:
6     # default configuration for services in *this* file
7     _defaults:
... line 8
9         autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 10 - 32
```

We'll also talk about `public: false` even sooner:

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

... lines 1 - 4

```
5 services:
6     # default configuration for services in *this* file
7     _defaults:
... lines 8 - 9
10        public: false        # Allows optimizing the container by removing unused services; this also means
11                             # fetching services directly from the container via $container->get() won't work.
12                             # The best practice is to be explicit about your dependencies anyway.
... lines 13 - 32
```

The point is: we've established a few *default* values for any services that this file registers. No big deal.

[Service Auto-Registration](#)

The *real* magic comes down here with this App\ entry:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 13
14  # makes classes in src/ available to be used as services
15  # this creates a service per class whose id is the fully-qualified class name
16  App\:
17      resource: '../src/*'
18      exclude: '../src/{Entity,Migrations,Tests}'
... lines 19 - 32
```

This says:

Make *all* classes inside src/ available as services in the container.

You can see this in real life! Run:

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

At the top, yep! Our controller and MarkdownHelper appear in this list. And any *future* classes will also show up here, automatically.

But wait! Does that mean that all of our classes are *instantiated* on every single request? Because, that would be *super* wasteful!

Sadly... yes! Bah, I'm kidding! Come on - Symfony kicks way more but than that! No: this line simply tells the container to be *aware* of these classes. But services are *never* instantiated until - and *unless* - someone asks for them. So, if we didn't ask for our MarkdownHelper, it would never be instantiated on that request. Winning!

[Services are only Instantiated Once](#)

Oh, and one important thing: each service in the container is instantiated a maximum of *once* per request. If *multiple* parts of our code ask for the MarkdownHelper, it will be created just *once*, and the *same* instance will be passed each time. That's *awesome* for performance: we don't need *multiple* markdown helpers... even if we need to call `parse()` multiple times.

[The Services exclude Key](#)

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 13
14  # makes classes in src/ available to be used as services
15  # this creates a service per class whose id is the fully-qualified class name
16  App\:
... line 17
18      exclude: '../src/{Entity,Migrations,Tests}'
... lines 19 - 32
```

The exclude key is not too important: if you *know* that some classes don't need to be in the container, you can exclude them for a small performance boost in the dev environment only.

So between `_defaults` and this App\ line - which we have given the fancy name - "service auto-registration" - everything just... works! New classes are added to the container and autowiring handles most of the heavy-lifting!

Oh, and this last `App\Controller\` part is not important:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 19
20  # controllers are imported separately to make sure services can be injected
21  # as action arguments even if you don't extend any base controller class
22  App\Controller\:
23      resource: '../src/Controller'
24      tags: ['controller.service_arguments']
... lines 25 - 32
```

The classes in `Controller\` are *already* registered as services thanks to the `App\` section. This adds a special tag to controllers... which you just *shouldn't* worry about. Honestly.

Finally, at the bottom, if you need to configure *one* service, this is where you do it: put the class name, then the config below:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 25
26  App\Service\MarkdownHelper:
27      arguments:
28          $logger: '@monolog.logger.markdown'
... lines 29 - 32
```

Services Ids = Class Name

And actually, this is *not* the *class name* of the service. It's *really* the service *id*... which happens to be equal to the class name. Run:

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

Most services in the container have a "snake case" service id. That's the best-practice for re-usable bundles. But thanks to service auto-registration, *our* service id's are equal to their class name. I just wanted to point that out.

The Amazing bind

Thanks to *all* of this config... well... we don't need to spend much time in this config file! We *only* need to configure the "special cases" - like we did for `MarkdownHelper`.

And actually.. there's a *much* cooler way to do that! Copy the service id and delete the config:

```
32 lines | config/services.yaml
... lines 1 - 4
5  services:
... lines 6 - 25
26  App\Service\MarkdownHelper:
27      arguments:
28          $logger: '@monolog.logger.markdown'
... lines 29 - 32
```

If we didn't do *anything* else, Symfony would once-again pass us the "main" `Logger` object.

Now, add a new key beneath `_defaults` called `bind`. Then add `$markdownLogger` set to `@monolog.logger.markdown`:

```

32 lines | config/services.yaml
... lines 1 - 4
5  services:
6      # default configuration for services in *this* file
7      _defaults:
... lines 8 - 13
14     # setup special, global autowiring rules
15     bind:
16         $markdownLogger: '@monolog.logger.markdown'
... lines 17 - 32

```

Copy that argument name, open MarkdownHelper, and rename the argument from \$logger to \$markdownLogger. Update it below too:

```

37 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 14
15     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger)
16     {
... lines 17 - 18
19         $this->logger = $markdownLogger;
20     }
... lines 21 - 35
36 }

```

Ok: markdown.log still only has one line. And... refresh! Check the file... hey! It worked!

I /love bind: it says:

If you find *any* argument named \$markdownLogger, pass this service to it.

And because we added it to _defaults, it applies to *all* our services. Instead of configuring our services one-by-one, we're creating project-wide *conventions*. Next time you need this logger? Yep, just name it \$markdownLogger and keep coding.

Next! In addition to services, the container can *also* hold flat configuration: called *parameters*.

Chapter 11: Config Parameters

The *container* is a fancy, scary word for a simple concept: the object that holds all of the *services* in our app. But actually, the container can *also* hold a *second* type of thing: normal boring config values! These are called *parameters* and, it turns out they're pretty handy!

Open `config/packages/framework.yaml`. We configured the cache system to use this `cache.adapter.apcu` service:

```
31 lines | config/packages/framework.yaml
1  framework:
    ... lines 2 - 16
17  cache:
    ... lines 18 - 28
29      # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30      app: cache.adapter.apcu
```

And then, in the dev environment only, we're *overriding* that to use `cache.adapter.filesystem`:

```
4 lines | config/packages/dev/framework.yaml
1  framework:
2  cache:
3  app: cache.adapter.filesystem
```

Creating a Parameter

Simple enough! But parameters can make this *even* easier. Check this out: inside *any* configuration file - because, remember, *all* of these files are loaded by the same system - you can add a `parameters` key. And below that, you can invent whatever keys you want.

Let's invent one called `cache_adapter`. Set its value to `cache.adapter.apcu`:

```
34 lines | config/packages/framework.yaml
1  parameters:
2      cache_adapter: cache.adapter.apcu
    ... lines 3 - 34
```

Using a Parameter

This basically creates a variable. And now we can *reference* this variable in *any* of these configuration files. How? Remove `cache.adapter.apcu` and, inside quotes, replace it with `%cache_adapter%`:

```
34 lines | config/packages/framework.yaml
1  parameters:
2      cache_adapter: cache.adapter.apcu
3
4  framework:
    ... lines 5 - 19
20  cache:
    ... lines 21 - 31
32      # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
33      app: '%cache_adapter%'
```

Yep, whenever you surround a string with percent signs, Symfony will *replace* this with that parameter's value.

Overriding a Parameter

So yea... parameters are basically config *variables*. And, what programmer doesn't like variables!?

The cool thing is, now that we have a parameter called `cache_adapter`, inside of the dev config, we can shorten things. Change the key to parameters and *override* `cache_adapter`: `cache.adapter.filesystem`:

```
3 lines | config/packages/dev/framework.yaml
1  parameters:
2  cache_adapter: 'cache.adapter.filesystem'
```

Oh, and you may have noticed that *sometimes* I use quotes in YAML and sometimes I don't. Yay consistency! YAML is *super* friendly... and so most of the time, quotes aren't needed. But sometimes, like when a value starts with % or contains @, you *do* need them. Sheesh! Don't worry too much: if you're not sure, use quotes. You'll get a clear error anyways when you *do* need them.

Ok, let's see if this works! Open `MarkdownHelper` and `dump($this->cache)`:

```
38 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 21
22 public function parse(string $source): string
23 {
... lines 24 - 27
28     dump($this->cache);die;
... lines 29 - 35
36 }
37 }
```

In your browser, wave hello to this astronaut. Then, refresh! Yes! It is *still* using the filesystem adapter, since we're in the dev environment.

Moving Parameters to services.yaml

Now that we know that *any* config file can define parameters... let's *stop* putting them everywhere! I mean, usually, for organization, we like to *only* define parameters in one-ish files: `services.yaml`. Let's remove the parameter from the main `framework.yaml` and add it there:

```
33 lines | config/services.yaml
1  # Put parameters here that don't need to change on each machine where the app is deployed
2  # https://symfony.com/doc/current/best_practices/configuration.html#application-related-configuration
3  parameters:
4  cache_adapter: cache.adapter.apcu
... lines 5 - 33
```

But... we have a problem. When you refresh *now*, woh! We're suddenly using the APCU adapter, even though we're in the dev environment! Whaaaaat?

Remember the *order* that these files are loaded: files in `config/packages` are loaded first, then anything in `config/packages/dev`, and *last*, `services.yaml`. That means that the config in `services.yaml` is *overriding* our dev config file!

Boo! How can we fix that? Create a new config file called `services_dev.yaml`. This is the built-in way to create an environment-specific *services* file. And you can see that we actually started with one for the test environment. Inside, copy the code from the dev `framework.yaml` and paste it here:

```
3 lines | config/services_dev.yaml
```

```
1 parameters:
2   cache_adapter: 'cache.adapter.filesystem'
```

Oh, and delete the old framework.yaml file. Now, refresh!

Woo! It *works*!

And that's really it! In framework.yaml, we just reference the parameter...

```
31 lines | config/packages/framework.yaml
```

```
1 framework:
  ... lines 2 - 16
17 cache:
  ... lines 18 - 28
29 # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
30 app: '%cache_adapter%'
```

Which can be set in *any* other file. Like in this case: we set it in services.yaml and override it in services_dev.yaml:

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

```
... lines 1 - 2
3 parameters:
4   cache_adapter: cache.adapter.apcu
... lines 5 - 33
```

```
3 lines | config/services_dev.yaml
```

```
1 parameters:
2   cache_adapter: 'cache.adapter.filesystem'
```

Actually, if you think about it, since framework.yaml is loaded *first*, the parameter isn't even *defined* at this point. But that's ok: you can reference a parameter, even if it's not set until later. Nice!

Using a Parameter in a Service

But wait, there's more! We can *also* use parameters inside our *code* - like in MarkdownParser. Suppose that we want to *completely* disable caching when we're in the dev environment.

How can we do that? Add a new argument called `$isDebug`:

```
44 lines | src/Service/MarkdownHelper.php
```

```
... lines 1 - 8
9 class MarkdownHelper
10 {
  ... lines 11 - 15
16   public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
17   {
  ... lines 18 - 21
22   }
  ... lines 23 - 42
43 }
```

Yep, in addition to other *services*, if your service has any config - like `isDebug` or an API key - those should *also* be passed as constructor arguments.

The idea is that we will configure Symfony to pass true or false based on our environment. I'll press Alt+Enter and select "Initialize fields" so that PhpStorm creates and sets that property for me:

```

44 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 13
14     private $isDebug;
15
16     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
17     {
... lines 18 - 20
21         $this->isDebug = $isDebug;
22     }
... lines 23 - 42
43 }

```

Below, we can say: if `$this->isDebug`, then just return the uncached value:

```

44 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 13
14     private $isDebug;
15
16     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
17     {
... lines 18 - 20
21         $this->isDebug = $isDebug;
22     }
23
24     public function parse(string $source): string
25     {
... lines 26 - 29
30         // skip caching entirely in debug
31         if ($this->isDebug) {
32             return $this->markdown->transform($source);
33         }
... lines 34 - 41
42     }
43 }

```

Notice: this is the *first* time that we've had a constructor argument that is *not* a service. This is important: Symfony will *not* be able to autowire this value. Sure, we gave it a bool type-hint, but that's not enough for Symfony to guess what we want. Oh, and reverse my logic - I had it backwards!

```

44 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
9  class MarkdownHelper
10 {
... lines 11 - 23
24     public function parse(string $source): string
25     {
... lines 26 - 30
31         if ($this->isDebug) {
... line 32
33         }
... lines 34 - 41
42     }
43 }

```

To see that the argument cannot be autowired, refresh! Yep! A clear message:

Cannot autowire service MarkdownHelper: argument \$isDebug must have a type-hint or be given a value explicitly.

This is the *other* main situation when autowiring does *not* work. But... just like before, it's no problem! If Symfony can't figure out what value to pass to an argument, just tell it! In services.yaml, we *could* configure the *argument* for *just* this one service. But that's no fun! Add another global bind instead: \$isDebug and just hardcode it to true for now:

```

34 lines | config/services.yaml
... lines 1 - 5
6  services:
7      # default configuration for services in *this* file
8      _defaults:
... lines 9 - 14
15     # setup special, global autowiring rules
16     bind:
... line 17
18         $isDebug: true
... lines 19 - 34

```

Ok, move over and... refresh! Yea! It works! And if you check out the caching section of the profiler... yes! No calls!

Built-in kernel.* Parameters

To set the \$isDebug argument to the correct value, we *could* create a parameter, set it to false in services.yaml, override it in services_dev.yaml, and use it under bind.

But don't do it! Symfony *already* has a parameter we can use! In your terminal, the debug:container command *normally* lists services. But if you pass --parameters, well, you can guess what it prints:

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

Just like with services, most of these are internal values you don't care about. But, there are several that *are* useful: they start with kernel., like kernel.debug. That parameter is true most of the time, but is false in the prod environment.

Oh, and kernel.project_dir is *also* really handy. Copy kernel.debug, move back to services.yaml, and use %kernel.debug%:

```
34 lines | config/services.yaml
... lines 1 - 5
6  services:
7    # default configuration for services in *this* file
8    _defaults:
... lines 9 - 14
15    # setup special, global autowiring rules
16    bind:
... line 17
18    $isDebug: '%kernel.debug%'
... lines 19 - 34
```

Try it! Refresh! It still works!

Ok, it's time to talk a *little* bit more about *controllers*. It turns out, *they're* services too!

Chapter 12: Constructors for your Controller

Autowiring works in exactly two places. First it works for controller actions. Arguments can either have the same name as a route wildcard - that's actually *not* autowiring - just a feature of controllers - *or* have a type-hint for a service:

```
75 lines | src/Controller/ArticleController.php
... lines 1 - 4
5 use App\Service\MarkdownHelper;
... lines 6 - 12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 22
23 /**
24  * @Route("/news/{slug}", name="article_show")
25  */
26 public function show($slug, MarkdownHelper $markdownHelper)
27 {
... lines 28 - 60
61 }
... lines 62 - 73
74 }
```

Tip

Actually, there are a *few* other types of arguments you can get in your controller. You'll learn about them as we continue!

The second place autowiring works is the `__construct()` method of services:

```
44 lines | src/Service/MarkdownHelper.php
... lines 1 - 4
5 use Michelf\MarkdownInterface;
6 use Psr\Log\LoggerInterface;
7 use Symfony\Component\Cache\Adapter\AdapterInterface;
8
9 class MarkdownHelper
10 {
... lines 11 - 15
16 public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
17 {
... lines 18 - 21
22 }
... lines 23 - 42
43 }
```

And actually, this is the *real* place where autowiring is meant to work: Symfony's container - and its autowiring logic - is really good at *instantiating* objects.

[Binding Non-Service Arguments to Controllers](#)

In `services.yaml`, we added an `$isDebug` *bind* and used it inside of `MarkdownHelper`:

```

34 lines | config/services.yaml
... lines 1 - 5
6  services:
7      # default configuration for services in *this* file
8      _defaults:
... lines 9 - 14
15     # setup special, global autowiring rules
16     bind:
... line 17
18     $isDebug: '%kernel.debug%'
... lines 19 - 34

```

So... could we also add an `$isDebug` argument to a controller function? It certainly makes sense, so let's try it!

Add bool `$isDebug` - the bool part is optional, it doesn't change any behavior. Below, just dump it:

```

77 lines | src/Controller/ArticleController.php
... lines 1 - 12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 25
26     public function show($slug, MarkdownHelper $markdownHelper, $isDebug)
27     {
28         dump($isDebug);die;
... lines 29 - 62
63     }
... lines 64 - 75
76 }

```

Try it. Refresh! And... woh! It does *not* work!

Controller `ArticleController::show()` requires that you provide a value for the `$isDebug` argument.

This... probably *should* work, and there's a good chance that it *will* work in the future. The bind functionality is relatively new. And it has *one* edge-case that does not currently work: you cannot use it to bind non-service arguments to controllers.

I know, kinda weird. But, we're working on it - and it might already work by the time you watch this. Yay open source!

[Adding a Constructor to your Controller](#)

Remove the `$isDebug` argument. So... how can we access non-service values - like parameters - from inside our controller? The answer is simple! Remember: our controller is a *service*, just like `MarkdownHelper`. And we're now *pretty* good at working with services, if I do say so myself.

Tip

In Symfony 4.1, the base `AbstractController` will have a `$this->getParameter()` shortcut method.

Add public function `__construct()` with a bool `$isDebug` argument. Then, dump that variable and die:

```

80 lines | src/Controller/ArticleController.php
... lines 1 - 12
13 class ArticleController extends AbstractController
14 {
15     public function __construct(bool $isDebug)
16     {
17         dump($isDebug);die;
18     }
... lines 19 - 30
31     public function show($slug, MarkdownHelper $markdownHelper)
32     {
... lines 33 - 65
66     }
... lines 67 - 78
79 }

```

Immediately when we refresh... it works! I'll press Alt+Enter and select "Initialize fields" to create an \$isDebug property and set it. I don't actually need to use this - but let's keep it as an example - I'll add a comment:

```

85 lines | src/Controller/ArticleController.php
... lines 1 - 12
13 class ArticleController extends AbstractController
14 {
15     /**
16      * Currently unused: just showing a controller with a constructor!
17      */
18     private $isDebug;
19
20     public function __construct(bool $isDebug)
21     {
22         $this->isDebug = $isDebug;
23     }
... lines 24 - 83
84 }

```

So, it's not as *convenient* as fetching a value via an argument to your controller action, but it works *just* the same. And actually, like I mentioned earlier, the container's job is really to *instantiate* services. And so autowiring should *really* only work for `__construct()` functions! In fact, the only reason that it *also* works for controller actions is for convenience! Yep, one core Symfony dev once said to another:

Hey! Autowiring is *great*! But since it's *so* common to need services in a controller function, maybe we should make it work there too!

And then some *other* core developer said:

Oh man, that sounds great! Virtual high-five!!

This is not really that important. But the point is this: Symfony's container is *great* and *instantiating* service objects and using autowiring to pass values to their constructor. But *every* other function call - um, except for controller actions - will not have this magic. So don't expect it.

Thanks to bind, we can define what values are passed to specific argument names. But, we can go further and control what value should be passed for a specific *type* hint. That's next.

Chapter 13: Installing Bundles with "Average" Docs

Let's do something fun! Google for SlackBundle - you'll find one called nexylan/slack-bundle. This is a fun library that gives us a *service* that can send messages to a Slack channel. To install it, find the composer require line, copy that, move over to your terminal and paste:

```
$ composer require nexylan/slack-bundle:2.2.0 php-http/guzzle6-adapter:1.1.1
```

Interesting: this installs the bundle *and* some other library called guzzle6-adapter. Wait for it to install and... it fails!

Don't panic. There are two important things happening. First, this installed *two* bundles! Cool! You can see both of them inside bundles.php:

```
15 lines | config/bundles.php
... lines 1 - 2
3  return [
... lines 4 - 11
12  Http\HttpplugBundle\HttpplugBundle::class => ['all' => true],
13  Nexy\SlackBundle\NexySlackBundle::class => ['all' => true],
14 ];
```

For the SlackBundle, it says "auto-generated recipe". That means that the bundle doesn't actually *have* a recipe... but Symfony Flex, at least added it to bundles.php for us. By the way, this is not necessarily a bad thing: sometimes a bundle doesn't really need a custom recipe!

The second bundle *did* have a recipe. Before I started recording, I committed my changes so far. To see what that recipe did, let's run:

```
$ git status
```

Interesting: it added a new configuration file called httpplug.yaml:

```
15 lines | config/packages/httpplug.yaml
1  httpplug:
2    plugins:
3      redirect:
4        preserve_header: true
5
6    discovery:
7      client: 'auto'
8
9    clients:
10     app:
11       http_methods_client: true
12       plugins:
13         - 'httpplug.plugin.content_length'
14         - 'httpplug.plugin.redirect'
```

We don't know what this does, but it probably configures some sensible defaults. Let's ignore it unless the docs tells us otherwise.

When composer require Fails

The *second* important thing I want to talk about is... well... this big error!

The child node "endpoint" at path "nexy_slack" must be configured.

Oof... this is not a great error message. It means that this bundle *requires* some configuration, which we don't have yet. And since it didn't add a configuration file via a recipe, we'll need to create it ourselves. But before we do that, the *most* important thing to understand is this: when you see an error like this after running `composer require`, Composer *did* finish successfully and the library *was* installed.

Configuring the Slack Endpoint

Ok, let's go read the docs so we can figure out how to configure this bundle. Ah, so *one* of the reasons that installing this bundle isn't smoother is that its documentation is out-of-date! Hopefully it will be updated soon, but actually, this is a *great* example of how to navigate less-than-up-to-date docs.

How do I know it's out-of-date? This AppKernel thing is a Symfony 3 concept. *We* don't need to worry about enabling bundles: this is done for us automatically.

If you scroll down... ah, *here* is the configuration. And it says that this is an example of *default* values... which probably means that we don't need to copy all of this. Yep, we just need to fill in the parts that are required, so, endpoint.

Let's copy part of the configuration file. But... it doesn't tell us *where* to put this! That's ok! We already know: it can live in *any* file in `config/packages`. Let's create a new one called `nexy_slack.yaml`. Paste the config, but the only key we need is `endpoint`:

```
3 lines | config/packages/nexy_slack.yaml
```

```
1 nexy_slack:
```

```
... lines 2 - 3
```

If you're coding along, here's how this will work. First, you'll need access to a Slack workspace where you're an admin. If you don't have one, you can create one: it's free and easy.

Once you've got it, go to your domain `/apps/manage`, then search the App Directory for "Incoming Webhooks". Click "Add Configuration" to setup a new webhook: I've already done this.

Thanks to this, you now have a new Webhook URL, which anyone can use to send messages to your Slack. There's no authentication - the URL is meant to be a secret. Um... yea, I know you can read mine - I'm super bad at secrets. I'll invalidate it after I record.

Copy the URL and paste it next to `endpoint`:

```
3 lines | config/packages/nexy_slack.yaml
```

```
1 nexy_slack:
```

```
2   endpoint: 'https://hooks.slack.com/services/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'
```

Now, move over and clear your cache:

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

Again, starting in Symfony 4.0.5, you will *not* need to clear your cache when adding a new config file.

What is `$this->get()`?

Sweet! The bundle is configured, so... how do we use it? Go back to the docs. Below, yea! Usage! And *this* is where things get *really* interesting. The code says `$this->get('nexy_slack.client')`. What the heck is that?

Actually, this is something from Symfony 3... which we do not recommend doing in Symfony 4 and may or may *not* work, depending on the situation. Basically, `$this->get()` is, or was, a shortcut to fetch a service by its id. Instead of doing this, we are - of course - going to fetch the service via autowiring.

You guys know the drill: find your terminal and run:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. The terminal prompt is a dollar sign followed by the command 'php bin/console debug:autowiring'.

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

And search for "Slack". Wait... nothing!

Yep... this bundle *technically* works with Symfony 4... but it hasn't been fully updated. And so, it doesn't expose *any* services for autowiring! Right now, there is *no* way to autowire that nexy_slack.client service.

We need to learn a *little* bit more about *public* versus *private* services. And then take control of things with an autowiring *alias*!

Chapter 14: Autowiring Aliases

The way we coded in Symfony 3 was a bit different than Symfony 4. And... well... we need to learn just a *little* bit about the Symfony 3 way. Why? Because, when you find bundles with outdated docs, or old StackOverflow answers, I want you to be able to *translate* that into Symfony 4.

Public Versus Private Services

In Symfony 3, services were defined as *public*. This means that you could use a `$this->get()` shortcut method in your controller to fetch a service by its id. Or, if you had the container object itself - yep, that's totally possible - you could say `$container->get()` to do the same thing.

But in Symfony 4, most services are *private*. What does that mean? Very simply, when a service is *private*, you *cannot* use the `$this->get()` shortcut to fetch it.

At first, it might seem like we're just making life more difficult! But actually, Symfony 4 simply has a new philosophy.

Tip

You may not see the `public: false` defined anymore because that's the default value starting in Symfony 4.

Open `services.yaml` and, below `_defaults`, check out the `public: false` config:

```
34 lines | config/services.yaml
... lines 1 - 5
6  services:
7    # default configuration for services in *this* file
8    _defaults:
9    ... lines 9 - 10
11   public: false    # Allows optimizing the container by removing unused services; this also means
12                   # fetching services directly from the container via $container->get() won't work.
13                   # The best practice is to be explicit about your dependencies anyway.
... lines 14 - 34
```

Thanks to this, any service that *we* create is *private*. And so, we *cannot* fetch our services with `$this->get()`. Increasingly, more and more third-party bundles are *also* making their services private.

And because so many services are now private, instead of using `$this->get()`, we need to fetch services via "dependency injection" - a fancy, scary-sounding term that describes what we've been doing... this entire tutorial: passing services and config as arguments. This is considered a better coding practice than `$this->get()`, which means that *we* get to write nice code. Woo!

And actually... it *also* makes your app faster! It's not huge, but private services are faster than public services.

If you DO Want to use `$this->get()`

Side note, if you *do* want to use the `$this->get()` shortcut to fetch a public service - which you should *not* - you'll need to change your base controller class to `Controller` instead of `AbstractController`:

```

85 lines | src/Controller/ArticleController.php
... lines 1 - 7
8  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... lines 9 - 12
13 class ArticleController extends AbstractController
14 {
... lines 15 - 83
84 }

```

It's not important why... and you shouldn't do it anyways :p.

Fetching a Service by id

Okay okay, so if we *can't* use the `$this->get()` shortcut, how the heck can we fetch this service? Let's experiment! Copy the service id, find your terminal, and run:

```
$ php bin/console debug:container nexy_slack.client
```

Apparently the class for this object is `Nexy\Slack\Client`. We didn't see *any* autowiring type-hints that would work in `debug:autowiring...` but... maybe it *will* work if we type-hint this class?

Let's try it! In `ArticleController::show()`, add another argument: `Client` - make sure you get the one from `Nexy\Slack` - then `$slack`:

```

94 lines | src/Controller/ArticleController.php
... lines 1 - 5
6  use Nexy\Slack\Client;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 36
37     public function show($slug, MarkdownHelper $markdownHelper, Client $slack)
38     {
... lines 39 - 79
80     }
... lines 81 - 92
93 }

```

Add an if statement: if `$slug === 'khaaaaaan'`:

```

94 lines | src/Controller/ArticleController.php
... lines 1 - 5
6  use Nexy\Slack\Client;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 36
37     public function show($slug, MarkdownHelper $markdownHelper, Client $slack)
38     {
39         if ($slug === 'khaaaaaan') {
... lines 40 - 44
45         }
... lines 46 - 79
80     }
... lines 81 - 92
93 }

```


Then we need to know about this! Go copy some code from the docs. Then, simplify a bit - we don't need the attachment stuff, this is coming from Khan and the text should be "Ah, Kirk, my old friend.":

```
94 lines | src/Controller/ArticleController.php
... lines 1 - 5
6 use Nexy\Slack\Client;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 36
37 public function show($slug, MarkdownHelper $markdownHelper, Client $slack)
38 {
39     if ($slug === 'khaaaaaan') {
40         $message = $slack->createMessage()
41             ->from('Khan')
42             ->withIcon(':ghost:')
43             ->setText('Ah, Kirk, my old friend...');
44         $slack->sendMessage($message);
45     }
... lines 46 - 79
80 }
... lines 81 - 92
93 }
```

Excellent! Copy the slug. Then go to that page in your browser. And... it *totally* did *not* work: the `$slack` argument is missing.

Well... I guess debug:autowiring doesn't lie. Copy the `$slack` argument to the constructor:

```
94 lines | src/Controller/ArticleController.php
... lines 1 - 5
6 use Nexy\Slack\Client;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 20
21 public function __construct(bool $isDebug, Client $slack)
22 {
... line 23
24 }
... lines 25 - 92
93 }
```

No, it won't work here either... but we *will* get a better error message. Actually, this is due to another shortcoming with controller autowiring: when it fails, the error isn't great. That will hopefully *also* be improved in the future. Again, a few of these features are still brand new!

Refresh! Ah, much better:

```
Cannot autowire service ArticleController: argument $slack of method __construct() references class
Nexy\Slack\Client, but no such service exists.
```

This basically means that we're missing configuration to tell the container *which* services to pass for this type-hint. So... are we dead? No way! We are in *total* control of autowiring.

Open `services.yaml`, then go copy the full class name for the client: `Nexy\Slack\Client`. Back under `bind`, instead of using the argument *name*, like `$slack`, put the *class* name: `Nexy\Slack\Client`. Set this to the target service id: `@nexy_slack.client`:

```

35 lines | config/services.yaml
... lines 1 - 5
6  services:
7      # default configuration for services in *this* file
8      _defaults:
... lines 9 - 14
15     # setup special, global autowiring rules
16     bind:
... lines 17 - 18
19     Nexy\Slack\Client: '@nexy_slack.client'
... lines 20 - 35

```

That's it! Bind has *two* super-powers: you can bind by the argument name *or* you can bind by a class or interface. We're defining our *own* rules for autowiring!

Tip

You should comment this line out with `nexylan/slack-bundle >=2.2.0` to avoid circular reference, as this alias is already added by the bundle

Let's make sure I'm not lying: refresh! Yes! There's our Slack notification.

Autowiring Aliases

But I want to make just *one* small tweak. In `services.yaml`, instead of putting this beneath `_defaults` and `bind`, let's un-indent it so that it's at the root of `services`:

```

37 lines | config/services.yaml
... lines 1 - 5
6  services:
... lines 7 - 19
20  # custom aliases for autowiring
21  Nexy\Slack\Client: '@nexy_slack.client'
... lines 22 - 37

```

Refresh again. It works *exactly* like before!

The difference is subtle. Config beneath `_defaults` *only* affects services that are added in this *file*. But when you put this same config directly under `services`, it will affect *all* services in the system. In practice... that makes *no* difference: only *our* code uses autowiring: third-party libraries do *not* using autowiring, to keep things predicatable.

The *biggest* reason I did this is that, when you run `debug:autowiring`:

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

and search for "Slack", there it is! When it's under `bind`, it won't show up here.

About Autowiring Logic

But... this trick *also* shows us a bit more about how autowiring *works*. By putting this config directly under `services`, we're creating a *new* service in the container with the id `Nexy\Slack\Client`. But this is not a real service, it's just an "alias" - a "shortcut" - to fetch the existing `nexy_slack.client` service.

Here's the important question: when an argument to a service hasn't been configured under `bind` or arguments, how does the autowiring figure out *which* service to pass? The answer is *super* simple: it just looks for a service whose id *exactly* matches the type-hint. Yep, *now* that there is a service whose id is `Nexy\Slack\Client`, we can use that class as a type-hint. That's *also* why *our* classes - like `MarkdownHelper` can be autowired: each class in `src/` is auto-registered as a service and given an id that matches the class name.

Ok, it's time to turn to something different, but *very* important in Symfony 4: environment variables. This will help us to *not* hardcode our secret Slack URL inside our code.

Chapter 15: Environment Variables

Our Slack feature is working... but this kind of sucks! Our "secret" URL is hardcoded in the middle of a config file!

```
3 lines | config/packages/nexy_slack.yaml
1 nexy_slack:
2   endpoint: 'https://hooks.slack.com/services/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'
```

This is a bummer because I don't want to commit this to version control! And what if I need to use a *different* value on production?

We're going to have this problem a *bunch* more times - for example - with our database password! We need *some* good way of *isolating* any sensitive or server-specific config so that they're not stuck in the middle of our code.

Intro to Environment Variables

One of the *best* ways to do this - and the way that Symfony recommends - is via *environment variables*. OooOOoo. But... environment variables are still kind of a mystery to a lot of PHP devs. A *mostly* accurate description is that they're variables that are set on your *operating* system, that can then be read by your code. How? Usually with the `getenv()` function or `$_SERVER`.

Actually, open `public/index.php`. Hey! Our code is *already* reading an environment variable: `APP_ENV`:

```
40 lines | public/index.php
... lines 1 - 9
10 // The check is to ensure we don't use .env in production
11 if (!isset($_SERVER['APP_ENV'])) {
12     if (!class_exists(Dotenv::class)) {
13         throw new \RuntimeException('APP_ENV environment variable is not defined. You need to define environment variables for confi
14     }
15     (new Dotenv())->load(__DIR__.'/../.env');
16 }
17
18 $env = $_SERVER['APP_ENV'] ?? 'dev';
... lines 19 - 40
```

Tip

If you start a new project today, you won't see this `APP_ENV` logic. It's been moved to a `config/bootstrap.php` file.

But here's the question: how can we *remove* this hardcoded URL, and *instead* tell the `NexySlackBundle` to read from some *environment* variable? I mean, it's not like we can just use the `getenv()` PHP function in the middle of YAML!

Copy the URL - we'll need it later, then empty the value. Symfony has a special syntax that can be used in config files to read from environment variables. It's a little weird at first, but stick with me: `%env()%`. Between the parentheses, put the name of the environment variable. We'll be setting a *new* environment variable, so how about, `SLACK_WEBHOOK_ENDPOINT`:

```
3 lines | config/packages/nexy_slack.yaml
1 nexy_slack:
2   endpoint: '%env(SLACK_WEBHOOK_ENDPOINT)%'
```

By convention, environment variables are uppercase. And huh... this *looks* like a *parameter*: it has the `%` at the beginning and at the end. And... internally, it is! It's just a *special* parameter that will eventually resolve to this environment variable.

If we refresh now... error! Perfect!

Environment variable not found: SLACK_WEBHOOK_ENDPOINT.

Setting Environment Variables in .env

I love clear errors. So... how the heck do we *set* environment variables? Well... unfortunately, it *totally* depends on your setup! The solution is different if you're using Apache, Nginx, Docker or some Platform-as-a-Service. I'll talk more about that later.

But since setting environment variables can be a pain, Symfony gives us a *much* easier way to set them while developing. How? Open the .env file at the root of our project:

```
# This file is a "template" of which env vars need to be defined for your application
# Copy this file to .env file for development, create environment variables when deploying to production
# https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration

###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=5ea3114a349591bd131296e00f21c20a
#TRUSTED_PROXIES=127.0.0.1,127.0.0.2
#TRUSTED_HOSTS=localhost,example.com
###< symfony/framework-bundle ###
```

This file is loaded inside index.php... as long as the APP_ENV environment variable isn't set some other way:

```
40 lines | public/index.php

... lines 1 - 9
10 // The check is to ensure we don't use .env in production
11 if (!isset($_SERVER['APP_ENV'])) {
12     if (!class_exists(Dotenv::class)) {
13         throw new \RuntimeException("APP_ENV environment variable is not defined. You need to define environment variables for confi
14     }
15     (new Dotenv())->load(__DIR__.'/../.env');
16 }

... lines 17 - 40
```

And... it's pretty simple: it reads all of these keys and *sets* each as a new environment variable. This file was originally added by a recipe and - this is *really* cool - other recipes will *update* this file: adding new environment variables for *their* libraries.

But, we're totally free to add our own stuff. Let's invent a new section on top:

```
# This file is a "template" of which env vars need to be defined for your application
# Copy this file to .env file for development, create environment variables when deploying to production
# https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration

### CUSTOM VARS

### END CUSTOM VARS

###> symfony/framework-bundle ###
# ...
###< symfony/framework-bundle ###
```

The fancy code comments around the framework-bundle section were added by Flex: it's so that it knows *where* the environment variables live for that library... basically so that it can *remove* them if we remove that bundle.

Our new section is just for clarity.

Add SLACK_WEBHOOK_ENDPOINT= and then our URL:

```
# This file is a "template" of which env vars need to be defined for your application
# Copy this file to .env file for development, create environment variables when deploying to production
# https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration

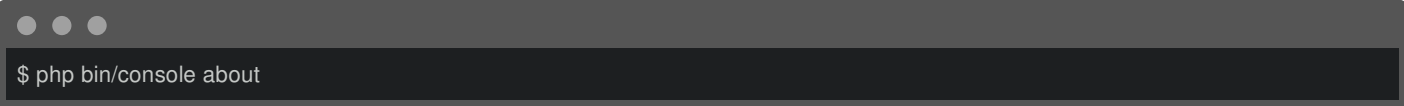
### CUSTOM VARS
SLACK_WEBHOOK_ENDPOINT=https://hooks.slack.com/services/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
### END CUSTOM VARS

###> symfony/framework-bundle ###
# ...
###< symfony/framework-bundle ###
```

And yep! That's all we need! Refresh! It works!

[Seeing all Environment Variables](#)

If you want to see *all* the environment variables that are currently set, there's a handy bin/console command for that:



```
$ php bin/console about
```

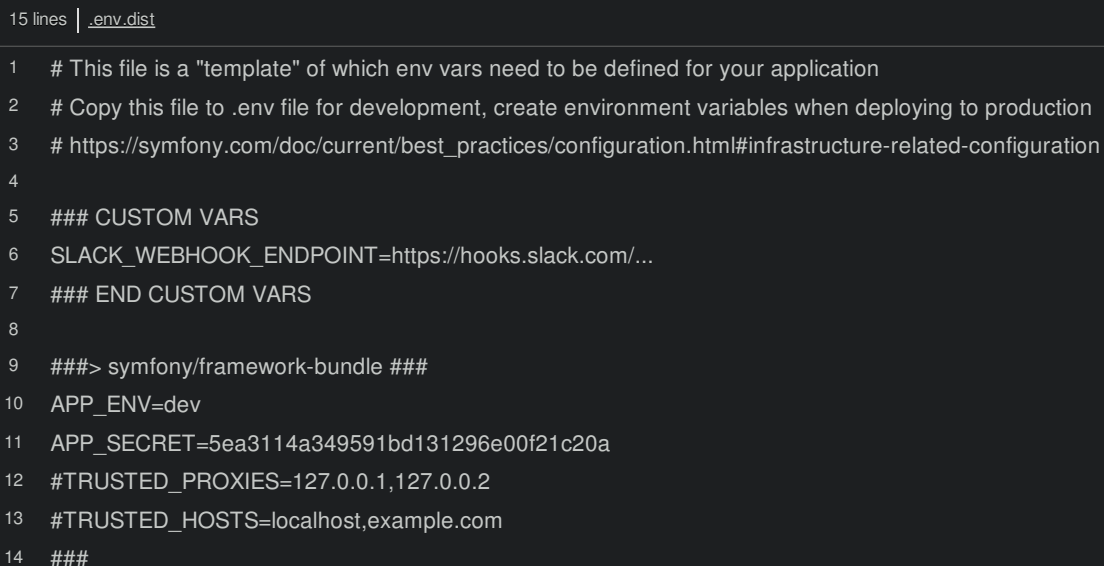
This shows your Symfony version, some system info and - hello! - environment variables!

[Updating .env.dist](#)

Tip

New projects will *not* have a .env.dist file. Instead, your .env file *is* committed to your repository and should hold sensible, but not "secret" default values. To override these defaults with values specific to your machine, create a .env.local file. This file *will* be ignored by git.

In addition to the .env file, there is *another* file: .env.dist. Copy our new section, open that file, and paste! Remove the sensitive part of the URL:



```
15 lines | .env.dist
1 # This file is a "template" of which env vars need to be defined for your application
2 # Copy this file to .env file for development, create environment variables when deploying to production
3 # https://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
4
5 ### CUSTOM VARS
6 SLACK_WEBHOOK_ENDPOINT=https://hooks.slack.com/...
7 ### END CUSTOM VARS
8
9 ###> symfony/framework-bundle ###
10 APP_ENV=dev
11 APP_SECRET=5ea3114a349591bd131296e00f21c20a
12 #TRUSTED_PROXIES=127.0.0.1,127.0.0.2
13 #TRUSTED_HOSTS=localhost,example.com
14 ###
```

This file is *not* read by Symfony: it's just meant to be a *template* file that contains all the environment variables our app needs. Why? Well, this file *will* be committed to the repository... but .env will *not*: it's in our .gitignore. So, when a new dev works on the project for the first time, they can copy .env.dist to .env and then fill in their custom values.

And... yea! That's basically it! There is one fancy syntax in config files to *read* environment variables, and a .env file to help *set* them during development.

[Environment Variables on Production](#)

But... what about on production? Let's talk a bit more about this.

Chapter 16: Env Var Tricks & on Production

When you deploy to production, you're *supposed* to set *all* these environment variables *correctly*. If you look back at index.php:

```
40 lines | public/index.php
... lines 1 - 9
10 // The check is to ensure we don't use .env in production
11 if (!isset($_SERVER['APP_ENV'])) {
12     if (!class_exists(Dotenv::class)) {
13         throw new \RuntimeException('APP_ENV environment variable is not defined. You need to define environment variables for confi
14     }
15     (new Dotenv())->load(__DIR__.'/../.env');
16 }
... lines 17 - 40
```

If the APP_ENV environment variable is set already, it knows to *skip* loading the .env file.

Tip

If you start a new project today, you won't see this APP_ENV logic. It's been moved to a config/bootstrap.php file.

In reality... in a lot of server environments, setting environment variables can be a *pain*. You can do it in your Apache virtual host or in PHP-FPM. Oh, and you'll need to make sure it's set at the command line too, so you can run bin/console.

If you use a Platform-as-a-Service like Platform.sh or Heroku, then setting environment variables is *super* easy! Lucky you!

But if setting environment variable is tough in your situation, well, you *could* still use the .env file. I mean if we deployed right now, we could create this file, put all the real values inside, and Symfony would use that! Well, if you're planning on doing this, make sure to move the dotenv library from the require-dev section of your composer.json to require by removing and re-adding it:

```
$ composer remove symfony/dotenv
$ composer require symfony/dotenv
```

The reason that using .env isn't *recommended* is mostly because the logic to parse this file isn't optimized: it's not *meant* for production! So, you'll lose a *small* amount of performance - probably just a couple of milliseconds, but you can profile it to be sure.

Tip

The performance cost of .env has been shown to be low. It *is* ok to use a .env file in production if that's the most convenient way for you to set environment variables.

Casting Environment Variables

But... there is *one* other limitation of environment variables that affects *everyone*: environment variables are *always* strings! But, what if you need an environment variable that's set to true or false? Well... when you *read* it with the special syntax, "false" will literally be the *string* "false". Boo!

Don't worry! Environment variables have *one* more trick! You can *cast* values by prefixing the name with, for example, string::

```
3 lines | config/packages/nexy_slack.yaml
```

```
1 nexy_slack:
2   endpoint: '%env(string:SLACK_WEBHOOK_ENDPOINT)%'
```

Well, this is *already* a string, but you get the idea!

To show some better examples, Google for Symfony Advanced Environment Variables to find a [blog post](#) about this feature. Cooooool. This DATABASE_PORT should be an int so... we cast it! You can also use bool or float.

[Setting Default Environment Variables](#)

This is great... but then, the Symfony devs went *crazy*. First, as you'll see in this blog post, you can set *default* environment variable values under the parameters key. For example, by adding an env(SECRET_FILE) parameter, you've just defined a *default* SECRET_FILE environment value. If a *real* SECRET_FILE environment variable were set, it would override this.

[Custom Processing](#)

More importantly, there are 5 *other* prefixes you can use for special processing:

- First, resolve: will resolve parameters - the %foo% things - if you have them *inside* your environment variable;
- Second, you can use file: to return the *contents* of a file, when that file's path is stored in an environment variable;
- Third, base64: will base64_decode a value: that's handy if you have a value that contains line breaks or special characters: you can base64_encode it to make it easier to *set* as an environment variable;
- Fourth, constant: allows you to read PHP constants;
- And finally, json: will, yep, call your friend Jason on the phone. Hey Jason! I mean, it will json_decode() a string.

And, ready for the *coolest* part? You can *chain* these: like, open a file, and then decode its JSON:

```
app.secrets: '%env(json:file:SECRETS_FILE)%'
```

Actually, sorry, there's more! You can even create your *own*, custom prefix - like blackhole: and write your own custom processing logic.

Ok, I'll shut up already about environment variables! They're cool, yadda, yadda, yadda.

Let's move on to a *super* fun, *super* unknown "extra" with autowiring.

Chapter 17: Bonus! LoggerTrait & Setter Injection

What if we wanna send Slack messages from somewhere *else* in our app? This is the *same* problem we had before with markdown processing. Whenever you want to re-use some code - or just organize things a bit better - take that code and move it into its own *service* class.

Since this is *such* an important skill, let's do it: in the Service/ directory - though we could put this anywhere - create a new class: SlackClient:

```
32 lines | src/Service/SlackClient.php
... lines 1 - 2
3 namespace App\Service;
... lines 4 - 7
8 class SlackClient
9 {
... lines 10 - 30
31 }
```

Give it a public function called, how about, sendMessage() with arguments \$from and \$message:

```
32 lines | src/Service/SlackClient.php
... lines 1 - 7
8 class SlackClient
9 {
... lines 10 - 18
19 public function sendMessage(string $from, string $message)
20 {
... lines 21 - 29
30 }
31 }
```

Next, copy the code from the controller, paste, and make the from and message parts dynamic:

```
32 lines | src/Service/SlackClient.php
... lines 1 - 7
8 class SlackClient
9 {
... lines 10 - 18
19 public function sendMessage(string $from, string $message)
20 {
... lines 21 - 24
25     $message = $this->slack->createMessage()
26         ->from($from)
27         ->withIcon(':ghost:')
28         ->setText($message);
29     $this->slack->sendMessage($message);
30 }
31 }
```

Oh, but let's rename the variable to \$slackMessage - having *two* \$message variables is no fun.

At this point, we just need the Slack client service. You know the drill: create a constructor! Type-hint the argument with Client from Nexy\Slack:

```

32 lines | src/Service/SlackClient.php
... lines 1 - 5
6 use Naxy\Slack\Client;
7
8 class SlackClient
9 {
... lines 10 - 13
14 public function __construct(Client $slack)
15 {
... line 16
17 }
... lines 18 - 30
31 }

```

Then press Alt+Enter and select "Initialize fields" to create that property and set it:

```

32 lines | src/Service/SlackClient.php
... lines 1 - 5
6 use Naxy\Slack\Client;
7
8 class SlackClient
9 {
... lines 10 - 11
12 private $slack;
13
14 public function __construct(Client $slack)
15 {
16     $this->slack = $slack;
17 }
... lines 18 - 30
31 }

```

Below, celebrate! Use `$this->slack`:

```

32 lines | src/Service/SlackClient.php
... lines 1 - 5
6 use Naxy\Slack\Client;
7
8 class SlackClient
9 {
... lines 10 - 11
12 private $slack;
13
14 public function __construct(Client $slack)
15 {
16     $this->slack = $slack;
17 }
18
19 public function sendMessage(string $from, string $message)
20 {
... lines 21 - 28
29     $this->slack->sendMessage($message);
30 }
31 }

```

In about one minute, we have a completely functional new service. Woo! Back in the controller, type-hint the new

SlackClient:

```
90 lines | src/Controller/ArticleController.php
... lines 1 - 5
6 use App\Service\SlackClient;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 36
37 public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack)
38 {
... lines 39 - 75
76 }
... lines 77 - 88
89 }
```

And below... simplify: `$slack->sendMessage()` and pass it the from - Khan - and our message. Clean up the rest of the code:

```
90 lines | src/Controller/ArticleController.php
... lines 1 - 5
6 use App\Service\SlackClient;
... lines 7 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 36
37 public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack)
38 {
39     if ($slug === 'khaaaaaan') {
40         $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
41     }
... lines 42 - 75
76 }
... lines 77 - 88
89 }
```

And I don't *need* to, but I'll remove the old use statement:

```
94 lines | src/Controller/ArticleController.php
... lines 1 - 5
6 use Nexy\Slack\Client;
... lines 7 - 94
```

Yay refactoring! Does it work? Refresh! Of course - we *rock*!

Setter Injection

Now let's go a step further... In `SlackClient`, I want to log a message. But, we *already* know how to do this: add a second constructor argument, type-hint it with `LoggerInterface` and, we're done!

But... there's *another* way to autowire your dependencies: *setter* injection. Ok, it's just a fancy-sounding word for a simple concept. Setter injection is less common than passing things through the constructor, but sometimes it makes sense for *optional* dependencies - like a logger. What I mean is, if a logger was not passed to this class, we could *still* write our code so that it works. It's not *required* like the Slack client.

Anyways, here's how setter injection works: create a public function `setLogger()` with the normal `LoggerInterface` `$logger` argument:

```

40 lines | src/Service/SlackClient.php
... lines 1 - 5
6 use Psr\Log\LoggerInterface;
7
8 class SlackClient
9 {
... lines 10 - 21
22     public function setLogger(LoggerInterface $logger)
23     {
... line 24
25     }
... lines 26 - 38
39 }

```

Create the property for this: there's no shortcut to help us this time. Inside, say `$this->logger = $logger`:

```

40 lines | src/Service/SlackClient.php
... lines 1 - 5
6 use Psr\Log\LoggerInterface;
7
8 class SlackClient
9 {
... lines 10 - 14
15     private $logger;
... lines 16 - 21
22     public function setLogger(LoggerInterface $logger)
23     {
24         $this->logger = $logger;
25     }
... lines 26 - 38
39 }

```

In `sendMessage()`, let's use it! Start with `if ($this->logger)`. And inside, `$this->logger->info()`:

```

40 lines | src/Service/SlackClient.php
... lines 1 - 7
8 class SlackClient
9 {
... lines 10 - 14
15     private $logger;
... lines 16 - 26
27     public function sendMessage(string $from, string $message)
28     {
29         if ($this->logger) {
30             $this->logger->info('Beaming a message to Slack!');
31         }
... lines 32 - 37
38     }
39 }

```

Bah! No auto-complete: with setter injection, we need to help PhpStorm by adding some PHPDoc on the property: it will be `LoggerInterface` or - in theory - `null`:

40 lines | [src/Service/SlackClient.php](#)

```
... lines 1 - 5
6 use Psr\Log\LoggerInterface;
7
8 class SlackClient
9 {
10     ... lines 10 - 11
11
12     /**
13      * @var LoggerInterface|null
14      */
15     private $logger;
16     ... lines 16 - 38
39 }
```

Now it auto-completes ->info(). Say, "Beaming a message to Slack!":

40 lines | [src/Service/SlackClient.php](#)

```
... lines 1 - 7
8 class SlackClient
9 {
10     ... lines 10 - 11
11
12     /**
13      * @var LoggerInterface|null
14      */
15     private $logger;
16     ... lines 16 - 26
27     public function sendMessage(string $from, string $message)
28     {
29         if ($this->logger) {
30             $this->logger->info('Beaming a message to Slack!');
31         }
32     ... lines 32 - 37
38     }
39 }
```

In practice, the if statement isn't needed: when we're done, Symfony *will* pass us the logger, *always*. But... I'm coding *defensively* because, from the perspective of this class, there's no guarantee that whoever is using it will call setLogger().

So... is Symfony smart enough to call this method automatically? Let's find out - refresh! Our class still works... but check out the profiler and go to "Logs". Bah! Nothing is logged yet!

[The @required Directive](#)

Yep, Symfony's autowiring is not *that* magic - and that's on purpose: it only autowires the __construct() method. But... it would be *pretty* cool if we could somehow say:

Hey container! How are you? Oh, I'm *wonderful* - thanks for asking. Anyways, after you instantiate SlackClient, could you *also* call setLogger()?

And... yeah! That's not only *possible*, it's *easy*. Above setLogger(), add /** to create PHPDoc. You can keep or delete the @param stuff - that's *only* documentation. But here's the magic: add @required:

```

43 lines | src/Service/SlackClient.php
... lines 1 - 7
8  class SlackClient
9  {
... lines 10 - 21
22  /**
23   * @required
24   */
25  public function setLogger(LoggerInterface $logger)
26  {
... line 27
28  }
... lines 29 - 41
42  }

```

As *soon* as you put `@required` above a method, Symfony will *call* that method before giving us the object. And thanks to autowiring, it will pass the logger service to the argument.

Ok, move over and... try it! There's the Slack message. And... in the logs... yes! We are logging!

The LoggerTrait

But... I have *one* more trick to show you. I like logging, so I need this service pretty often. What if we used the `@required` feature to create... a `LoggerTrait`? That would let us log messages with just *one* line of code!

Check this out: in `src/`, create a new `Helper` directory. But again... this directory could be named *anything*. Inside, add a new PHP Class. Actually, change this to be a trait, and call it `LoggerTrait`:

```

28 lines | src/Helper/LoggerTrait.php
... line 1
2  namespace App\Helper;
... lines 3 - 5
6  trait LoggerTrait
7  {
... lines 8 - 26
27 }

```

Ok, let's move the logger property to the trait... as well as the `setLogger()` method. I'll retype the "e" on `LoggerInterface` and hit Tab to get the use statement:

```

28 lines | src/Helper/LoggerTrait.php
... lines 1 - 3
4  use Psr\Log\LoggerInterface;
5
6  trait LoggerTrait
7  {
8      /**
9       * @var LoggerInterface|null
10     */
11     private $logger;
12
13     /**
14     * @required
15     */
16     public function setLogger(LoggerInterface $logger)
17     {
18         $this->logger = $logger;
19     }
20
21     ... lines 20 - 26
27 }

```

Next, add a new function called `logInfo()` that has two arguments: a `$message` and an array argument called `$context` - make it optional:

```

28 lines | src/Helper/LoggerTrait.php
... lines 1 - 5
6  trait LoggerTrait
7  {
8      ... lines 8 - 20
21     private function logInfo(string $message, array $context = [])
22     {
23         ... lines 23 - 25
26     }
27 }

```

We haven't used it yet, but all the log methods - like `info()` - have an optional second argument where you can pass extra information. Inside the method: let's keep coding defensively: if `($this->logger)`, then `$this->logger->info($message, $context)`:

```

28 lines | src/Helper/LoggerTrait.php
... lines 1 - 5
6  trait LoggerTrait
7  {
8      ... lines 8 - 20
21     private function logInfo(string $message, array $context = [])
22     {
23         if ($this->logger) {
24             $this->logger->info($message, $context);
25         }
26     }
27 }

```

Now, go back to `SlackClient`. Thanks to the trait, if we ever need to log something, all we need to do is add `use LoggerTrait`:

32 lines | [src/Service/SlackClient.php](#)

... lines 1 - 4

5 use App\Helper\LoggerTrait;

... lines 6 - 7

8 class SlackClient

9 {

10 use LoggerTrait;

... lines 11 - 30

31 }

Then, below, use `$this->logInfo()`. Pass the message... and, let's even pass some extra information - how about a message key with our text:

32 lines | [src/Service/SlackClient.php](#)

... lines 1 - 4

5 use App\Helper\LoggerTrait;

... lines 6 - 7

8 class SlackClient

9 {

10 use LoggerTrait;

... lines 11 - 18

19 public function sendMessage(string \$from, string \$message)

20 {

21 \$this->logInfo('Beaming a message to Slack!', [

22 'message' => \$message

23]);

... lines 24 - 29

30 }

31 }

And that's it! Thanks to the trait, Symfony will automatically call the `setLogger()` method. Try it! Move over and... refresh!

We get the Slack message and... in the profiler, yes! And this time, the log message has a bit more information.


I hope you *love* the LoggerTrait idea.

Chapter 18: MakerBundle

For our last trick, I want to introduce a bundle that's going to make our life *awesome*. And, for the first time, *we* are going to hook *into* Symfony.

[Installing Maker Bundle](#)

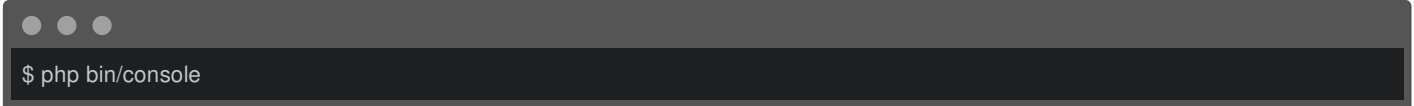
First, find your terminal, and install that bundle:



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

Yep! That's a Flex alias for symfony/maker-bundle. And, in this case, "make" means - "make your life easier by *generating* code".

We know that the *main* purpose of a bundle is to give us *services*. And, that's true in this case too... but the *purpose* of these services isn't for us to use them directly, like in our controller. Nope, the purpose of these services is that they give us new bin/console commands:




```
$ php bin/console
```

Nice! About 10 new commands, capable of generating all *kinds* of things. And, more make commands are still being added.

[Generating a new Command](#)

So... let's try one! Let's use the MakerBundle to create our very *own*, custom bin/console command. Use:



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

This will ask us for a command name - how about article:stats - we'll create a command that will return some stats about an article. And... it's done! We now have a shiny new src/Command/ArticleStatsCommand.php file. Open it!

37 lines | [src/Command/ArticleStatsCommand.php](#)

... lines 1 - 2

```
3 namespace App\Command;
4
5 use Symfony\Component\Console\Command\Command;
6 use Symfony\Component\Console\Input\InputArgument;
7 use Symfony\Component\Console\Input\InputInterface;
8 use Symfony\Component\Console\Input\InputOption;
9 use Symfony\Component\Console\Output\OutputInterface;
10 use Symfony\Component\Console\Style\SymfonyStyle;
11
12 class ArticleStatsCommand extends Command
13 {
14     protected static $defaultName = 'article:stats';
15
16     protected function configure()
17     {
18         $this
19             ->setDescription('Add a short description for your command')
20             ->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
21             ->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
22         ;
23     }
24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
28         $argument = $input->getArgument('arg1');
29
30         if ($input->getOption('option1')) {
31             // ...
32         }
33
34         $io->success('You have a new command! Now make it your own! Pass --help to see your options.');
```

Hey! It even added some example code to get us started! Run:

```
$ php bin/console
```

And on top... yes! Symfony *already* sees our new article:stats command. Sweet! Um... so... let's try it!

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

It doesn't do much... yet - but it's already working.

[Service autoconfigure](#)

But... how does Symfony *already* know about this new command? I mean, is it scanning all of our files looking for command classes? Actually, no! And that's a good thing - that would be *super* slow!

Here's the answer. Remember: all of our classes in src/ are loaded as services:

```

37 lines | config/services.yaml
... lines 1 - 5
6  services:
... lines 7 - 22
23  # makes classes in src/ available to be used as services
24  # this creates a service per class whose id is the fully-qualified class name
25  App\:
26      resource: '../src/*'
27      exclude: '../src/{Entity,Migrations,Tests}'
... lines 28 - 37

```

Notice that our new class extends Symfony's base Command class:

```

37 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 4
5  use Symfony\Component\Console\Command\Command;
... lines 6 - 11
12 class ArticleStatsCommand extends Command
... lines 13 - 35
36 }

```

When the service was registered, Symfony *noticed* this and made sure that it included it as a command. This nice feature has a name - autoconfigure. It's not too important, but just like autowiring, this is *activated* thanks to a little bit of config in our services.yaml file:

```

37 lines | config/services.yaml
... lines 1 - 5
6  services:
7      # default configuration for services in *this* file
8      _defaults:
... line 9
10     autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
... lines 11 - 37

```

It's just *another* way that you can avoid configuration, and keep working!

Next, let's have fun and make our command much more awesome!

Chapter 19: Fun with Commands

Time to make our command a bit more fun! Give it a description: "Returns some article stats":

```
47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 15
16     protected function configure()
17     {
18         $this
19             ->setDescription('Returns some article stats!')
... lines 20 - 21
22     };
23 }
... lines 24 - 45
46 }
```

Each command can have *arguments* - which are strings passed after the command and *options*, which are prefixed with --, like --option1:

```
$ php bin/console article:stats arg1 arg2 --option1 --opt2=khan
```

Rename the argument to slug, change it to InputArgument::REQUIRED - which means that you *must* pass this argument to the command, and give it a description: "The article's slug":

```
47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 15
16     protected function configure()
17     {
18         $this
19             ->setDescription('Returns some article stats!')
20             ->addArgument('slug', InputArgument::REQUIRED, 'The article\'s slug')
... line 21
22     };
23 }
... lines 24 - 45
46 }
```

Rename the *option* to format: I want to be able to say --format=json to get the article stats as JSON. Change this to VALUE_REQUIRED: instead of just --format, this means we need to say --format=something. Update its description, *and* give it a default value: text:

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 15
16     protected function configure()
17     {
18         $this
19             ->setDescription('Returns some article stats!')
20             ->addArgument('slug', InputArgument::OPTIONAL, 'The article\'s slug')
21             ->addOption('format', null, InputOption::VALUE_REQUIRED, 'The output format', 'text')
22     };
23 }
... lines 24 - 45
46 }

```

Perfect! We're not *using* these options yet, but we can already go back and run the command with a `--help` flag:

```
$ php bin/console article:stats --help
```

Actually, you can add `--help` to *any* command to get all the info about it - like the description, arguments and options... including a bunch of options that apply to *all* commands.

Customizing our Command

Ok, so the `configure()` method is where we set things up. But `execute()` is where the magic happens. We can do *whatever* we want here!

To get the argument value, update the `getArgument()` call to `slug` and rename the variable too:

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
28         $slug = $input->getArgument('slug');
... lines 29 - 44
45     }
46 }

```

Let's just invent some article "data": give this array a `slug` key and, how about, hearts set to a random number between 10 and 100:

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
28         $slug = $input->getArgument('slug');
29
30         $data = [
31             'slug' => $slug,
32             'hearts' => rand(10, 100),
33         ];
... lines 34 - 44
45     }
46 }

```

Clear out the rest of the code, and then add a switch statement on `$input->getOption('format')`. Here's the plan: we're going to support two different formats: text - don't forget the break - and json:

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
28         $slug = $input->getArgument('slug');
29
30         $data = [
31             'slug' => $slug,
32             'hearts' => rand(10, 100),
33         ];
34
35         switch ($input->getOption('format')) {
36             case 'text':
... line 37
38                 break;
39             case 'json':
... line 40
41                 break;
... lines 42 - 43
44         }
45     }
46 }

```

If someone tries to use a different format, yell at them!

What kind of crazy format is that?

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25 protected function execute(InputInterface $input, OutputInterface $output)
26 {
27     $io = new SymfonyStyle($input, $output);
28     $slug = $input->getArgument('slug');
29
30     $data = [
31         'slug' => $slug,
32         'hearts' => rand(10, 100),
33     ];
34
35     switch ($input->getOption('format')) {
36         case 'text':
... line 37
38             break;
39         case 'json':
... line 40
41             break;
42         default:
43             throw new \Exception('What kind of crazy format is that!?');
44     }
45 }
46 }

```

Printing Things

Notice that `execute()` has two arguments: `$input` and `$output`:

```

47 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 6
7 use Symfony\Component\Console\Input\InputInterface;
... line 8
9 use Symfony\Component\Console\Output\OutputInterface;
... lines 10 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25 protected function execute(InputInterface $input, OutputInterface $output)
26 {
... lines 27 - 44
45 }
46 }

```

Input lets us *read* arguments and options. And, you can even use it to ask questions interactively. `$output` is all about *printing* things. To make both of these even *easier* to use, we have a special `SymfonyStyle` object that's *full* of shortcut methods:

47 lines | [src/Command/ArticleStatsCommand.php](#)

... lines 1 - 11

```
12 class ArticleStatsCommand extends Command
13 {
    ... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
    ... lines 28 - 44
45     }
46 }
```

For example, to print a list of things, just say `$io->listing()` and pass the array:

47 lines | [src/Command/ArticleStatsCommand.php](#)

... lines 1 - 11

```
12 class ArticleStatsCommand extends Command
13 {
    ... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
    ... lines 28 - 34
35         switch ($input->getOption('format')) {
36             case 'text':
37                 $io->listing($data);
38                 break;
    ... lines 39 - 43
44         }
45     }
46 }
```

For json, to print raw text, use `$io->write()` - then `json_encode($data)`:

47 lines | [src/Command/ArticleStatsCommand.php](#)

... lines 1 - 11

```
12 class ArticleStatsCommand extends Command
13 {
    ... lines 14 - 24
25     protected function execute(InputInterface $input, OutputInterface $output)
26     {
27         $io = new SymfonyStyle($input, $output);
    ... lines 28 - 34
35         switch ($input->getOption('format')) {
36             case 'text':
37                 $io->listing($data);
38                 break;
39             case 'json':
40                 $io->write(json_encode($data));
41                 break;
    ... lines 42 - 43
44         }
45     }
46 }
```


And... we're done! Let's try this out! Find your terminal and run:

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

Nice! And now pass `--format=json`:

```
$ php bin/console article:stats khaaaaaan --format=json
```

Woohoo!

Printing a Table

But... this listing isn't very helpful: it just prints out the *values*, not the keys. The article has 88... what?

Instead of using listing, let's create a *table*.

Start with an empty `$rows` array. Now loop over the data as `$key => $val` and start adding rows with `$key` and `$val`:

```
51 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25 protected function execute(InputInterface $input, OutputInterface $output)
26 {
27     $io = new SymfonyStyle($input, $output);
... lines 28 - 34
35     switch ($input->getOption('format')) {
36         case 'text':
37             $rows = [];
38             foreach ($data as $key => $val) {
39                 $rows[] = [$key, $val];
40             }
... line 41
42         break;
... lines 43 - 47
48     }
49 }
50 }
```

We're doing this because the `SymfonyStyle` object has an awesome method called `->table()`. Pass it an array of headers - Key and Value, then `$rows`:

```

51 lines | src/Command/ArticleStatsCommand.php
... lines 1 - 5
6 use Symfony\Component\Console\Input\InputArgument;
... lines 7 - 11
12 class ArticleStatsCommand extends Command
13 {
... lines 14 - 24
25 protected function execute(InputInterface $input, OutputInterface $output)
26 {
27     $io = new SymfonyStyle($input, $output);
... lines 28 - 34
35     switch ($input->getOption('format')) {
36         case 'text':
37             $rows = [];
38             foreach ($data as $key => $val) {
39                 $rows[] = [$key, $val];
40             }
41             $io->table(['Key', 'Value'], $rows);
42             break;
... lines 43 - 47
48     }
49 }
50 }

```

Let's rock! Try the command again without the `--format` option:

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

Yes! So much better! And yea, that `$io` variable has a *bunch* of other features, like interactive questions, a progress bar and more. Not only are commands *fun*, but they're super easy to create thanks to MakerBundle.

Oh my gosh, you did it! You made it through Symfony Fundamentals! This was serious work that will *seriously* unlock you for *everything* else you do with Symfony! We now understand the configuration system and - most importantly - *services*. Guess what? Commands are services. So if you needed your `SlackClient` service, you would just add a `__construct()` method and autowire it!

Tip

When you do this, you need to call `parent::__construct()`. Commands are a rare case where there is a parent constructor!

With our new knowledge, let's keep going and start mastering features, like the Doctrine ORM, form system, API stuff and a lot more.

Alright guys, seeya next time!

