Symfony 6 Fundamentals: Services, Config & Environments

Chapter 1: Bundles!

Hey friends! Welcome back to Episode 2 of our Symfony 6 tutorial series. This is the one where we *seriously* level-up and unlock our potential to do *anything* we want. That's because, in this course, we're diving into the fundamentals behind *everything* in Symfony. We're talking about services, bundles, configuration, environments, environment variables - the stuff that *truly* makes Symfony *tick*. We're gonna throw open Symfony's hood and find out what's inside.

Site Setup!

To get the *most* fundamentals out of this fundamentals tutorial, I warmly invite you to cozy up to a fire, download the course code from this page and code along with me. It'll be fun! After you unzip the file, you'll find a `start/` directory with the same code that you see here. Follow our hand-crafted, locally-sourced `README.md` file for all of the setup instructions. The *last* step will be to open a terminal, move into the project and run

```
symfony serve -d
```

to start a local web server at `https://127.0.0.1:8000`. I'll cheat and click that link to see our site. It is... Mixed Vinyl! Our new startup where users can build their own custom "mixtape" - I'm thinking MMMBop followed by some Spice Girls - except that we deliver it straight to your door on a freshly pressed vinyl record. We even throw in that musty old record collection smell for free!

Services: Services Everywhere

In the previous tutorial, we talked briefly about how *everything* in Symfony is actually done by a *service*. And that the word "service" is a fancy term for a simple concept: a service is an object that does work.

For example, in `src/Controller/SongController.php`, we leveraged Symfony's Logger service to log a message:

```php
// src/Controller/SongController.php
// ... lines 1 - 4
use Psr\Log\LoggerInterface;
// ... lines 6 - 10
class SongController extends AbstractController
{
// ... line 13
    public function getSong(int $id, LoggerInterface $logger): Response
    {
// ... lines 16 - 22
        $logger->info('Returning API response for song {song}', [
            'song' => $id,
        ]);
// ... lines 26 - 27
    }
}
```

And, though we don't have the code in `VinylController` anymore, we briefly used the Twig service to directly render a Twig template:

```php
// src/Controller/VinylController.php
// ... lines 1 - 9
class VinylController extends AbstractController
{
// ... line 12
    public function homepage(): Response
    {
// ... lines 15 - 23
        return $this->render('vinyl/homepage.html.twig', [
// ... lines 25 - 26
        ]);
    }
// ... lines 29 - 38
}
```

So a service is just an object that does work... and *every* bit of work that's done in Symfony is done by a service. Heck, even the core code that figures which route matches the current URL is a service, called the "router" service.

Hello Bundles

So the next question is: where do these services come from? The answer to that is mordor. I mean *bundles*... services come from bundles.

Open up `config/bundles.php`:

config/bundles.php

```php
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
    Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
    Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true],
    Symfony\WebpackEncoreBundle\WebpackEncoreBundle::class => ['all' => true],
    Symfony\UX\Turbo\TurboBundle::class => ['all' => true],
];
```

This isn't a file that you need to look at or worry about much, but this is where your bundles are *activated*.

Very simply: bundles are Symfony plugins. They're just PHP code... but they hook *into* Symfony. And thanks to the recipe system, when we install a new bundle, that bundle is *automatically* added to this file, which is how we already have 8 bundles here. When we started our project, we only had 1!

So a bundle is a Symfony plugin. And bundles can give us several things... though they largely exist for one reason: to give us *services*. For example, this TwigBundle up here gives us the Twig *service*. If we removed this line, the Twig service would no longer exist and our application would *explode*... since we *are* rendering templates. This `render()` line would no longer work. And MonologBundle is what gives us the Logger service that we're using in `SongController`.

So by adding more bundles into our application, we're getting more *services*, and services are tools! Need more services? Install more bundles! It's like Neo in the best, I mean first Matrix movie.

Next... let's teach our app some Kung fu by installing a *new* bundle that gives us a *new* service to solve a *new* problem.

Chapter 2: New Bundle, New Service: KnpTimeBundle

On our site, you can create your *own* vinyl mix. (Or you'll *eventually* be able to do this... right now, this button doesn't do anything). But another great feature of our site is the ability to browse *other* user's mixes.

Now that I'm looking at this, it might be useful if we could see *when* each mix was created.

If you don't remember *where* in our code this page was built, you can use a trick. Down on the web debug toolbar, hover over the 200 status code. Ah, ha! This shows us that the controller behind this page is `VinylController::browse`.

Cool! Go open up `src/Controller/VinylController.php`. *Here* is the `browse` action:

```php
// src/Controller/VinylController.php
// ... lines 1 - 9
class VinylController extends AbstractController
{
// ... lines 12 - 29
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;
        $mixes = $this->getMixes();

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre,
            'mixes' => $mixes,
        ]);
    }
// ... lines 41 - 65
}
```

By the way, I *did* update the code a *little* bit since episode one... so make sure you've got a fresh copy if you're coding along with me.

This method calls `$this->getMixes()`... which is a private function I created down at the bottom:

```
src/Controller/VinylController.php
// ... lines 1 - 9
class VinylController extends AbstractController
{
// ... lines 12 - 41
    private function getMixes(): array
    {
        // temporary fake "mixes" data
        return [
            [
                'title' => 'PB & Jams',
                'trackCount' => 14,
                'genre' => 'Rock',
                'createdAt' => new \DateTime('2021-10-02'),
            ],
            [
                'title' => 'Put a Hex on your Ex',
                'trackCount' => 8,
                'genre' => 'Heavy Metal',
                'createdAt' => new \DateTime('2022-04-28'),
            ],
            [
                'title' => 'Spice Grills - Summer Tunes',
                'trackCount' => 10,
                'genre' => 'Pop',
                'createdAt' => new \DateTime('2019-06-20'),
            ],
        ];
    }
}
```

This returns a *big* array of fake data that represents the mixes we're going to render on the page. Eventually, we'll get this from a *dynamic* source, like a database.

Printing Dates in Twig

Notice that each mix has a `createdAt` date field. We get these mixes up in `browse()`... and pass them as a `mixes` variable into `vinyl/browse.html.twig`. Let's jump into that template.

Down here, we use Twig's `for` loop to loop over `mixes`. Simple enough!

```
templates/vinyl/browse.html.twig
// ... lines 1 - 3
<div class="container">
// ... lines 5 - 25
    <div>
        <h2 class="mt-5">Mixes</h2>
        <div class="row">
            {% for mix in mixes %}
            <div class="col col-md-4">
                <div class="mixed-vinyl-container p-3 text-center">
                    <img src="https://via.placeholder.com/300" data-src="https://via.placeholder.com/300" alt="Square placeholder img">
                    <p class="mt-2"><strong>{{ mix.title }}</strong></p>
                    <span>{{ mix.trackCount }} Tracks</span>
                    |
                    <span>{{ mix.genre }}</span>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>
// ... lines 43 - 44
```

Let's *also* now print the "created at" date. Add a `|`, another `<span>` and then say `{{ mix.createdAt }}`.

There's just one problem. If you look at `createdAt`... it's a `DateTime` object. And you can't just *print* `DateTime` objects... you'll get a big error reminding you... that you can't just print `DateTime` objects. Cruel world!

Fortunately, Twig has a handy `date` filter. We talked about filters briefly in the first episode: we using them by adding a `|` after some value and then the name of the filter. This *particular* filter also takes an argument, which is the *format* the date should be printed. To keep things simple, let's use `Y-m-d`, or "year-month-day".

```
templates/vinyl/browse.html.twig
// ... lines 1 - 3
<div class="container">
// ... lines 5 - 25
    <div>
        <h2 class="mt-5">Mixes</h2>
        <div class="row">
            {% for mix in mixes %}
            <div class="col col-md-4">
                <div class="mixed-vinyl-container p-3 text-center">
// ... lines 32 - 35
                    <span>{{ mix.genre }}</span>
                    |
                    <span>{{ mix.createdAt|date('Y-m-d') }}</span>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>
// ... lines 45 - 46
```

Head over and refresh and... okay! We can now see *when* each was created, though the format isn't very attractive. We *could* do more work to spruce this up... but it would be *way* cooler if we could print this out in the "ago" format.

You've probably seen it before.... like for comments on a blog post... they say something like "posted three months ago" or "posted 10 minutes ago".

So... the question is: How *can* we convert a `DateTime` object into that nice "ago" format? Well, that sounds like *work* to me and, as I said earlier, *work* in Symfony is done by a service. So the *real* question is: Is there a *service* in Symfony that can convert `DateTime` objects to the "ago" format? The answer is... no. But there *is* a third party bundle that can give us that service.

Installing KnpTimeBundle

Go to https://github.com/KnpLabs/KnpTimeBundle. If you look at this bundle's documentation, you'll see that it gives us a service that can do that conversion. So... let's get it installed!

Scroll to the `composer require` line, copy that, spin over to our terminal, and paste.

```
composer require knplabs/knp-time-bundle
```

Cool! This grabbed `knplabs/knp-time-bundle`... as well as `symfony/translation`: Symfony's translation component, which is a dependency of `KnpTimeBundle`. Near the bottom, it *also* configured two recipes. Let's see what those did. Run:

```
git status
```

Awesome! Any time you install a third party package, Composer will *always* modify your `composer.json` and `composer.lock` files. This *also* updated the `config/bundles.php` file:

```php
// config/bundles.php
<?php

return [
// ... lines 4 - 11
    Knp\Bundle\TimeBundle\KnpTimeBundle::class => ['all' => true],
];
```

That's because we just installed a bundle - `KnpTimeBundle` - and its recipe handled that automatically. It also looks like the translation recipe added a config file and a `translations/` directory. The translator *is* needed to use KnpTimeBundle... but we won't need to work with it directly.

So... what did installing this new bundle give us? Services of course! Let's find and use those next!

Chapter 3: Finding & Using the Services from a Bundle

We just installed KnpTimeBundle. Hooray! Um... but... uh... what does that mean? What did doing that *give* us?

The *number* one thing that a bundle gives us is... services! What services does *this* bundle give us? Well, we could, of course, read the documentation, blah, blah. Well, ok, you *should* do that... but, come on! Let's venture ahead recklessly and learn by exploring!

In the last tutorial, we learned about a command that shows us all of the services in our app: `debug:autowiring`:

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

For example, if we search for "logger", there's apparently a service called `LoggerInterface`. We *also* learned that we can autowire any service in this list into our controller by using its *type*. By using this `LoggerInterface` type - which is actually `Psr\Log\LoggerInterface` - Symfony knows to pass us this service. Then, down here, we call methods on it like `$logger->info()`.

We installed `KnpTimeBundle` a moment ago, so let's search for "time":

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

And... hey! Look at this! We have a new `DateTimeFormatter` service! That's from the new bundle and I bet that's what we're looking for. Let's go use it in our controller.

Using the New DateTimeFormatter Service

The type-hint we need is `Knp\Bundle\TimeBundle\DateTimeFormatter`. Ok! In `VinylController`, find `browse()`, then add the new argument.

By the way, the *order* of the arguments does *not* matter... except when it comes to *optional* arguments. I made the `$slug` argument optional and you typically need your optional arguments at the *end* of the list. So I'll add `DateTimeFormatter` right here and hit "tab" to add the `use` statement on top.

We can *name* the argument anything we want, like `$sherlockHolmes` or `$timeFormatter`:

src/Controller/VinylController.php

```php
// ... lines 1 - 4
use Knp\Bundle\TimeBundle\DateTimeFormatter;
// ... lines 6 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(DateTimeFormatter $timeFormatter, string $slug = null): Response
    {
// ... lines 34 - 45
    }
// ... lines 47 - 71
}
```

To use this, loop over the mixes - `foreach ($mixes as $key => $mix)`:

src/Controller/VinylController.php

```php
// ... lines 1 - 4
use Knp\Bundle\TimeBundle\DateTimeFormatter;
// ... lines 6 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(DateTimeFormatter $timeFormatter, string $slug = null): Response
    {
// ... lines 34 - 36
        foreach ($mixes as $key => $mix) {
// ... line 38
        }
// ... lines 40 - 45
    }
// ... lines 47 - 71
}
```

then, on each, add a new `ago` key: `$mixes[$key]['ago'] =`... and this is where we need the new service. How do we *use* the `DateTimeFormatter`? I have no idea! But we used its type, so PhpStorm should tell us what methods it has. Type `$timeFormatter->`... and ok! It has 4 public methods.

The one *we* want is `formatDiff()`. Pass it the "from" time... which is `$mix['createdAt']`:

```
src/Controller/VinylController.php
// ... lines 1 - 4
use Knp\Bundle\TimeBundle\DateTimeFormatter;
// ... lines 6 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(DateTimeFormatter $timeFormatter, string $slug = null): Response
    {
// ... lines 34 - 36
        foreach ($mixes as $key => $mix) {
            $mixes[$key]['ago'] = $timeFormatter->formatDiff($mix['createdAt']);
        }
// ... lines 40 - 45
    }
// ... lines 47 - 71
}
```

That's *all* we need! We're looping over these `$mixes`, taking the `createdAt` key, which is a `DateTime` object, passing it to the `formatDiff()` method, which should return a string in the "ago" format. To see if this is working, below, `dd($mixes)`:

```php
// src/Controller/VinylController.php
// ... lines 1 - 4
use Knp\Bundle\TimeBundle\DateTimeFormatter;
// ... lines 6 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(DateTimeFormatter $timeFormatter, string $slug = null): Response
    {
// ... lines 34 - 36
        foreach ($mixes as $key => $mix) {
            $mixes[$key]['ago'] = $timeFormatter->formatDiff($mix['createdAt']);
        }
        dd($mixes);
// ... lines 41 - 45
    }
// ... lines 47 - 71
}
```

Let's try it! Spin over, refresh... and let's open it up. Yes! Look at that: `"ago" => "7 months ago"`... `"ago" => "18 days ago"`... It *works*. So remove that dump:

```php
// src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(DateTimeFormatter $timeFormatter, string $slug = null): Response
    {
// ... lines 34 - 36
        foreach ($mixes as $key => $mix) {
            $mixes[$key]['ago'] = $timeFormatter->formatDiff($mix['createdAt']);
        }

        return $this->render('vinyl/browse.html.twig', [
// ... lines 42 - 43
        ]);
    }
// ... lines 46 - 70
}
```

And now that each mix has a new `ago` field, in `browse.html.twig`, replace the `mix.createdAt|date` code with `mix.ago`:

```
templates/vinyl/browse.html.twig
// ... lines 1 - 3
<div class="container">
// ... lines 5 - 25
    <div>
        <h2 class="mt-5">Mixes</h2>
        <div class="row">
            {% for mix in mixes %}
            <div class="col col-md-4">
                <div class="mixed-vinyl-container p-3 text-center">
// ... lines 32 - 35
                    <span>{{ mix.genre }}</span>
                    |
                    <span>{{ mix.ago }}</span>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>
// ... lines 45 - 46
```

And now... *much* better.

So: we had a problem... and *knew* it needed to be solved by a service... because services do work. We didn't *have* a service that did what we needed *yet*, so we went out, found one, and installed it. Problem *solved*! Symfony itself has a *ton* of different packages, and each of them gives us several services. But sometimes you'll need a third party bundle like this one to get the job done. Typically, you can just search online for the problem you're trying to solve, plus "Symfony bundle", to find it.

Using the ago Twig Filter

In addition to the nice `DateTimeFormatter` service that we just used, this bundle *also* gave us *another* service. But, this isn't a service that we're meant to use directly, like in the controller. Nope! This service is meant to be used by Twig *itself*... to power a brand new Twig filter! That's right! You can add custom functions, filters... or *anything* to Twig.

To see the new filter, let's try another useful debugging command:

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

This prints a list of *all* of the functions, filters, and tests in Twig, along with the *one* global Twig variable we have. If you go up to Filters, there's a new one called "ago"! That was *not* there before we installed `KnpTimeBundle`.

So, all of the work we did in our controller is perfectly fine ... but it turns out that there's an easier way to do all of this. Delete the `foreach`... remove the `DateTimeFormatter` service... and, though it's optional, clean up the extra `use` statement on top:

```php
// src/Controller/VinylController.php
// ... lines 1 - 9
class VinylController extends AbstractController
{
// ... lines 12 - 29
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;
        $mixes = $this->getMixes();

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre,
            'mixes' => $mixes,
        ]);
    }
// ... lines 41 - 65
}
```

In `browse.html.twig`, we don't have an `ago` field anymore... but we still have a `createdAt` field. Instead of piping this into the `date` filter, pipe it to `ago`:

```
templates/vinyl/browse.html.twig
// ... lines 1 - 3
<div class="container">
// ... lines 5 - 25
    <div>
        <h2 class="mt-5">Mixes</h2>
        <div class="row">
            {% for mix in mixes %}
            <div class="col col-md-4">
                <div class="mixed-vinyl-container p-3 text-center">
// ... lines 32 - 35
                    <span>{{ mix.genre }}</span>
                    |
                    <span>{{ mix.createdAt|ago }}</span>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>
// ... lines 45 - 46
```

That's all we need! Back over on the site refresh and... we get the *exact* same result.

By the way, we won't do it in this tutorial, but by the end, you'll be able to easily follow the documentation to create your own custom Twig functions and filters.

Ok, so our app does *not* have a database yet... and it won't until the *next* episode. But to make things more interesting, let's get our mixes data by making an HTTP call to a special GitHub repository. That's next.

Chapter 4: The HTTP Client Service

We don't have a database yet... and we'll save that for a *future* tutorial. But to make things a bit more fun, I've created a GitHub repository - https://github.com/SymfonyCasts/vinyl-mixes - with a `mixes.json` file that holds a *fake* database of vinyl mixes. Let's make an HTTP request from our Symfony app *to* this file and use *that* as our temporary database.

So... how *can* we make HTTP requests in Symfony? Well, making an HTTP request is *work*, and - say it with me now - "Work is done by a service". So the next question is: Is there already a service in our app that can make HTTP requests?

Let's find out! Spin over to your terminal and run:

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

to search the services for "http". We *do* get a bunch of results, but... nothing that looks like an HTTP client. And, that's correct. There is *not* currently any service in our app that can make HTTP requests.

Installing the HTTPClient Component

But, we can install *another* package to give us one. At your terminal, type:

```
composer require symfony/http-client
```

But, before we run that, I want to show you *where* this command comes from. Search for "symfony http client". One of the top results is Symfony.com's documentation that teaches about an HTTP Client component. Remember: Symfony is a collection of *many* different libraries, called components. And *this* one helps us make HTTP requests!

Near the top, you'll see a section called "Installation", and *there's* the line from our terminal!

Anyways, if we run that... cool! Once it finishes, try that `debug:autowiring` command again:

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

And... here it is! Right at the bottom: `HttpClientInterface`, which

> *“Provides flexible methods for requesting HTTP resources synchronously or asynchronously.”*

The Super Smart FrameworkBundle

Woo! We just got a new service! *That* means that we must have just installed a new bundle, right? Because... bundles give us services? Well... go check out `config/bundles.php`:

```
config/bundles.php
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
    Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
    Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true],
    Symfony\WebpackEncoreBundle\WebpackEncoreBundle::class => ['all' => true],
    Symfony\UX\Turbo\TurboBundle::class => ['all' => true],
    Knp\Bundle\TimeBundle\KnpTimeBundle::class => ['all' => true],
];
```

Woh! There's *no* new bundle here! Try running

```
git status
```

Yea... that *only* installed a normal PHP package. Inside `composer.json`, here's the new package... But it's *just* a "library": not a *bundle*.

```
composer.json
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 15
        "symfony/http-client": "6.1.*",
// ... lines 17 - 24
    },
// ... lines 26 - 84
}
```

So, *normally*, if you install "just" a PHP library, it gives you PHP classes, but it doesn't hook into Symfony to give you new *services*. What we just saw is a special trick that many of the Symfony components use. The *main* bundle in our app is `framework-bundle`. In fact, when we started our project, this was the *only* bundle we had. `framework-bundle` is *smart*. When you install a new Symfony component - like the HTTP Client component - that bundle *notices* the new library and automatically adds the services for it.

So the new service comes from `framework-bundle`... which adds that as soon as it detects that the `http-client` package is installed.

Using the HttpClientInterface service

Anyways, time to use the new service. The type we need is `HttpClientInterface`. Head over to `VinylController.php` and, up here in the `browse()` action, autowire `HttpClientInterface` and let's name it `$httpClient`:

```
src/Controller/VinylController.php
// ... lines 1 - 7
use Symfony\Contracts\HttpClient\HttpClientInterface;
// ... lines 9 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(HttpClientInterface $httpClient, string $slug = 
null): Response
    {
// ... lines 34 - 41
    }
// ... lines 43 - 67
}
```

Then, instead of calling `$this->getMixes()`, say `$response = $httpClient->`. This lists all of its methods... we *probably* want `request()`. Pass this `GET`... and then I'll paste the URL: you can copy this from the code block on this page. It's a direct link to the content of the `mixes.json` file:

```php
// src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(HttpClientInterface $httpClient, string $slug = null): Response
    {
// ... line 34
        $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');
// ... lines 36 - 41
    }
// ... lines 43 - 67
}
```

Cool! So we make the request and it returns a response containing the `mixes.json` data that we see here. Fortunately, this data has all of the same keys as the old data we were using down here... so we should be able to swap it in super easily. To get the mix data from the response, we can say `$mixes = $response->toArray()`:

```php
// src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(HttpClientInterface $httpClient, string $slug = null): Response
    {
// ... line 34
        $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');
        $mixes = $response->toArray();
// ... lines 37 - 41
    }
// ... lines 43 - 67
}
```

a handy method that JSON decodes the data for us!

Moment of truth! Move over, refresh and... it works! We now have *six* mixes on the page. And... super cool! A new icon showed up on the web debug toolbar: "Total requests: 1". The HTTP Client service hooks *into* the web debug toolbar to add this, which is pretty awesome. If we click it, we can see info about the request and the response. I *love* that.

To celebrate this working, spin back over and remove the hardcoded `getMixes()` method:

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... line 13
    public function homepage(): Response
    {
// ... lines 16 - 28
    }
// ... lines 30 - 31
    public function browse(HttpClientInterface $httpClient, string $slug = null): Response
    {
// ... lines 34 - 41
    }
}
```

The only problem I can think of now is that, every single time someone visits our page, we're making an HTTP request to GitHub's API... and HTTP requests are slow! To make matters worse, once our site becomes super popular - which won't take long - GitHub's API will probably start rate limiting us.

To fix this, let's leverage *another* Symfony service: the cache service.

Chapter 5: The Cache Service

Now when we refresh the browse page, the mixes are coming from a repository on GitHub! We make an HTTP request to the GitHub API, that fetches this file right here, we call `$response->toArray()` to *decode* that JSON into a `$mixes` array... and then we render *that* in the template. Yup, this file on GitHub is our temporary fake database!

One practical problem is that *every single* page load is now making an HTTP request... and HTTP requests are *slow*. If we deployed this to production, our site would be *so* popular, of course, that we'd pretty quickly hit our GitHub API limit. And *then* this page would *explode*.

So... I'm thinking: what if we cache the result? We could make this HTTP request, then cache the data for 10 minutes, or even an hour. That just *might* work! How do we *cache* things in Symfony? You guessed it: with a service! Which service? I dunno! So let's go find out.

Finding the Cache Service

Run:

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

to search for services with "cache" in their name. And... yes! There are, in fact, *several*! There's one called `CacheItemPoolInterface`, and another called `StoreInterface`. Some of these aren't *exactly* what we're looking for, but `CacheItemPoolInterface`, `CacheInterface`, and `TagAwareCacheInterface` *are* all different services that you can use for caching. They all *effectively* do the same thing... but the easiest to use is `CacheInterface`.

So let's grab that.... by doing our fancy autowiring trick! Add another argument to our method typed with `CacheInterface` (make sure you get the one from `Symfony\Contracts\Cache`) and call it, how about, `$cache`:

```
src/Controller/VinylController.php
// ... lines 1 - 7
use Symfony\Contracts\Cache\CacheInterface;
// ... lines 9 - 11
class VinylController extends AbstractController
{
// ... lines 14 - 32
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
// ... lines 35 - 46
    }
}
```

To *use* the `$cache` service, copy these two lines from before, delete them, and replace them with `$mixes = $cache->get()`, as if you're going to fetch some key out of the cache. We can invent whatever cache key we want: how about `mixes_data`.

Symfony's cache object works in a unique way. We call `$cache->get()` and pass it this key. If that result already exists in the cache, it will be returned *immediately*. If it does *not* exist in the cache yet, then it will call our *second* argument, which is a function. In here, our job is to return the data that *should* be cached. Paste in the two lines of code that we copied earlier. This `$httpClient` is undefined, so we need to add `use ($httpClient)` to bring it into scope.

There we go! And instead of setting the `$mixes` variable, just `return` this `$response->toArray()` line:

```
src/Controller/VinylController.php
// ... lines 1 - 11
class VinylController extends AbstractController
{
// ... lines 14 - 32
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
// ... lines 35 - 36
        $mixes = $cache->get('mixes_data', function() use ($httpClient) {
            $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');

            return $response->toArray();
        });
// ... lines 42 - 46
    }
}
```

If you haven't used Symfony's caching service before, this might look strange! But I love it! The first time we refresh the page, there won't be any `mixes_data` in the cache yet. So it will call our function, return the result, and then the cache system will store *that* in the cache. The *next* time we refresh the page, the key *will* be in the cache, and it will return the result *immediately*. So we don't need any "if" statements to see if something is already in the cache... just this!

Debugging with the Cache Profiler

But... will it blend? Let's go find out. Refresh and... beautiful! The first refresh *still* made the HTTP request like normal. Down on the web debug toolbar, we can see that there were *three* cache calls and *one* cache write. Open this in a new tab to jump into the cache section of the profiler.

So cool: this shows us that there was one call to the cache for `mixes_data`, one cache *write*, and one cache *miss*. A cache "miss" means that it called our function and wrote that to the cache.

On the next refresh, watch this icon here. It disappears! That's because there was *no* HTTP request. If you open the Cache profiler again, this time there was one read and one hit. That hit means that the result was loaded from the cache and it did *not* make an HTTP request. That's exactly what we wanted!

Setting the Cache Lifetime

Now, you *might* be wondering: how long will this info *stay* in the cache? Right now... *forever*. Ooooh. That's the default.

To make it expire *sooner* than forever, give the function a `CacheItemInterface` argument - make sure to hit "tab" to add that use statement - and call it `$cacheItem`. Now we can say `$cacheItem->expiresAfter()` and, to make it easy, say `5`:

```
src/Controller/VinylController.php
// ... lines 1 - 4
use Psr\Cache\CacheItemInterface;
// ... lines 6 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 33
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
// ... lines 36 - 37
        $mixes = $cache->get('mixes_data', function(CacheItemInterface $cacheItem) use ($httpClient) {
            $cacheItem->expiresAfter(5);
// ... lines 40 - 42
        });
// ... lines 44 - 48
    }
}
```

The item will expire after 5 seconds.

Clearing the Cache

Unfortunately, if we try this, the item that's *already* in the cache is set to *never* expire. So... this won't actually work until we clear the cache. But... where *is* the cache being stored? Another great question! We'll talk more about that in a second... but, by default, it's stored in `var/cache/dev/`... along with a bunch of other cache files that help Symfony do its job.

We *could* delete this directory manually to clear the cache... but Symfony has a better way! It is, of course, another `bin/console` command.

Symfony has a bunch of different "categories" of cache called "cache pools". If you run:

```
php bin/console cache:pool:list
```

you'll see all of them. Most of these are meant for *Symfony* to use internally. The cache pool that *we're* using is called `cache.app`. To clear that, run:

```
php bin/console cache:pool:clear cache.app
```

Thats it! This isn't something you'll need to do very often, but it's good to know, just in case.

Okay, check this out. When we refresh... we get a cache *miss* and you can see that it *did* make an HTTP call. But if we refresh again really quickly... it's gone! Refresh again and... it's back! That's because the five seconds just expired.

Ok team: we're now leveraging an HTTP client service *and* cache service... both of which were prepared *for* us by one of our bundles so that we can just... use them!

But, I do have a question. What if we need to *control* these services? For example, how could we tell the cache service that, instead of saving things onto the filesystem in this directory, we want to store things in Redis... or memcache? Let's explore the idea of *controlling* our services through configuration next.

Chapter 6: Bundle Config (to Control Bundle Services)

We're now using the `HttpClientInterface` and `CacheInterface` services. Yay! But *we* aren't actually responsible for *instantiating* these service objects. Nope, they're created by something else (we'll talk about that in a few minutes), and then just passed to us.

That's *great* because all of these services - the "tools" of our app - come ready to use, out-of-the-box. But... if something *else* is responsible for instantiating these service objects, how can we control them?

Introducing: bundle configuration!

Bundle Configuration

Go check out the `config/packages/` directory. This has a number of different YAML files, all of which are loaded *automatically* by Symfony when it first boots up. These files all have exactly *one* purpose: to configure the *services* that each bundle gives us.

Open up `twig.yaml`:

config/packages/twig.yaml

```yaml
twig:
    default_path: '%kernel.project_dir%/templates'

when@test:
    twig:
        strict_variables: true
```

For now, ignore this `when@test`: we're going to talk about that in a few minutes. This file has a root key called `twig`. And so, the entire purpose of this file is to control the services provide by the "Twig" bundle. And, it's not the filename - `twig.yaml` - that's important. I could rename this to `pineapple_pizza.yaml` and it would work *exactly* the same *and* be delicious. I don't care what you think.

When Symfony loads this file, it sees this root key - `twig` - and says:

> *“Oh, okay. I'm going to pass whatever configuration is below to TwigBundle.”*

And remember! Bundles give us *services*. Thanks to this config, when TwigBundle is preparing its services, Symfony passes it this configuration and TwigBundle *uses* it to decide *how* its services should be instantiated... like what class names to use for each service... or what first second or third constructor arguments to pass.

For example, if we changed the `default_path` to something like `%kernel.project_dir%/views`, the result is that the Twig service that renders templates would now be pre-configured to look in that directory.

The point is: the config in these files give us the power to control the services that each bundle provides.

Let's check out another one: `framework.yaml`:

```yaml
config/packages/framework.yaml
# see https://symfony.com/doc/current/reference/configuration/framework.html
framework:
    secret: '%env(APP_SECRET)%'
    #csrf_protection: true
    http_method_override: false

    # Enables session support. Note that the session will ONLY be started if you read or write from it.
    # Remove or comment this section to explicitly disable session support.
    session:
        handler_id: null
        cookie_secure: auto
        cookie_samesite: lax
        storage_factory_id: session.storage.factory.native

    #esi: true
    #fragments: true
    php_errors:
        log: true

when@test:
    framework:
        test: true
        session:
            storage_factory_id: session.storage.factory.mock_file
```

Because the root key is `framework`, all of this config is passed to FrameworkBundle... which uses it to configure the services it provides.

And, as I mentioned, the filename doesn't matter... though the name *often* matches the root key... just for sanity reasons: like `framework` and `framework.yaml`. But that's not *always* the case. Open up `cache.yaml`:

config/packages/cache.yaml

```yaml
framework:
    cache:
        # Unique name of your app: used to compute stable namespaces for cache keys.
        #prefix_seed: your_vendor_name/app_name

        # The "app" cache stores to the filesystem by default.
        # The data in this cache should persist between deploys.
        # Other options include:

        # Redis
        #app: cache.adapter.redis
        #default_redis_provider: redis://localhost

        # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
        #app: cache.adapter.apcu

        # Namespaced pools use the above "app" backend by default
        #pools:
            #my.dedicated.cache: null
```

Woh! This is... just *more* config for FrameworkBundle! It lives in its own file... just because it's nice to have a separate file to control the cache.

Debugging the Available Bundle Config

At this point, you might be asking yourself:

> *“Ok, cool... but what config keys are we allowed to put here? Where can I find which options are available?”*

Great question! Because... you can't just "invent" whatever keys you want: that would throw an error. First, yes, you can, *of course*, read the documentation. But there's *another* way: and it's one of my *favorite* things about Symfony's config system.

If you want to know what configuration you can pass to "Twig" bundle, there are two `bin/console` commands to help you. The first is:

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

This will print out all of the *current* configuration under the `twig` key, *including* any *default* values that the bundle is adding. You can see our `default_path` set to the `templates/` directory, which comes from our config file. This `%kernel.project_dir%` is just a fancy way to point to the root of our project. More on that later.

Try this: change the value to `views`, re-run that command and... yup! We see "views" in the output. Let me go ahead and change that back.

So `debug:config` shows us all of the *actual*, current config for a specific bundle, like `twig`... which is especially handy since it also shows you *defaults* added by the bundle. It's a great way to see *what* you can configure. For example, apparently we can add a global variable to Twig via this `globals` key!

The second command is similar: Instead of `debug:config`, it's `config:dump`:

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

`debug:config` shows you the *current* configuration... but `config:dump` shows you a giant tree of *example* configuration, which includes *everything* that's possible. Here you can see `globals` with some examples of how you could use that key. This is a great way to see *every* potential option that you can pass to a bundle... to help it configure its services.

Let's use this new knowledge to see if we can "teach" the cache service to store its files somewhere else. That's next.

Chapter 7: Configuring the Cache Service

So... I want to know how I can configure the cache service... like to store the cache somewhere else. In the real world, we can just search for "How do I configure Symfony's cache service". But... we can *also* figure this out on our own, by using the commands we just learned.

We already noticed there's a `cache.yaml` file:

config/packages/cache.yaml

```yaml
framework:
    cache:
        # Unique name of your app: used to compute stable namespaces for cache keys.
        #prefix_seed: your_vendor_name/app_name

        # The "app" cache stores to the filesystem by default.
        # The data in this cache should persist between deploys.
        # Other options include:

        # Redis
        #app: cache.adapter.redis
        #default_redis_provider: redis://localhost

        # APCu (not recommended with heavy random-write workloads as memory fragmentation can cause perf issues)
        #app: cache.adapter.apcu

        # Namespaced pools use the above "app" backend by default
        #pools:
            #my.dedicated.cache: null
```

It looks like FrameworkBundle is responsible for creating the cache service... and it has a sub `cache` key where we can pass *some* values to control it. All of this is commented-out at the moment.

To get more information about FrameworkBundle, run:

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

FrameworkBundle is the main bundle inside of Symfony. So you can see that this dumps... wow... a *ton*. FrameworkBundle provides a *lot* of services... so there's a lot of config.

Debugging the Cache Config

To... zoom in a bit, re-run the command again, passing `framework` *and* then `cache` to filter for that sub-key:

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

And... cool! This may not always be *super* understandable, but it's a great starting point. This definitely just helped us answer the question:

> *“Why does the cache system store stuff in the var/cache directory?”*

Because... there's a `directory` key that defaults to `%kernel.cache_dir%`... which is a fancy way of pointing at the `/var/cache/dev` directory. And then we see `/pools/app`, which is the actual directory that holds our cache.

Using dump() and the Profiler

So here's the goal: instead of caching things to the filesystem, I want to change the cache system to store somewhere else. Before we do that, go into `VinylController` and, so we can see the *result* of the change we're about to make, `dump($cache)`. We've been using `dd()` so far, which stands for "dump and die". But in this case I want `dump()`... but let the page load.

```
src/Controller/VinylController.php
// ... lines 1 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 33
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
// ... lines 36 - 37
        dump($cache);
// ... lines 39 - 49
    }
}
```

Refresh now. Wait, where *is* my dump? This is a... feature! When you use `dump()`, you won't actually see it on the page: it hides down here on the web debug toolbar. If you look there, the cache is some sort of `TraceableAdapter`. But inside of *that*, there's an object called `FilesystemAdapter`. That's proof that the cache system is *saving* to the filesystem.

Configuring the Cache Adapter

To make this store somewhere else, go into `cache.yaml` and change this `app` key. You can set this to a number of different special strings, called adapters. If we wanted to store our cache in Redis, we would use `cache.adapter.redis`.

To make things really easy, use `cache.adapter.array`:

```
config/packages/cache.yaml
framework:
    cache:
// ... lines 3 - 10
        app: cache.adapter.array
// ... lines 12 - 20
```

The `array` adapter is a *fake* cache where it *does* store things... but it only lives for the duration of the request. So, at the end of each request, it forgets everything. It's a fake cache, but it's enough to see how changing this key will affect the cache service itself.

Watch what happens. Currently, we have a `FilesystemAdapter`. When we refresh... the cache is an `ArrayAdapter`! And since the `ArrayAdapter` forgets its cache at the end of the request, you can see that every single request *does* now makes an HTTP request.

Takeaway: It's all about Controlling how Services are Instantiated

If you're a little confused by this, let me try to clear things up. The point of this chapter is *not* to teach you how to change this *specific* key in the cache file. Ultimately, if you need to configure something in Symfony, you'll just search the docs... which will tell you exactly what to do and which key to change.

Nope, the big takeaway is that the *sole* purpose of these config files is to *configure* the *services* in our app. Each time you change a key in *any* of these files, the end result is that you just changed how some service is *instantiated*. Tweaking a key may change the entire class name of a service object, like in this case, or it may change the 2nd or 3rd constructor argument that will be passed when the service is instantiated. It doesn't really matter *what* changes, as long as you realize that this config is *all* about services and how they're instantiated.

In fact, *none* of this config can be read directly from your app. You couldn't, for example, ask for the "cache" configuration from inside of a controller. Nope, Symfony reads this config, uses it to configure how each service object will be instantiated, then throws it away. Services are supreme.

Next, sometimes you'll need certain configuration to be *different* based on whether you're developing locally or running on production. Symfony has a system for this called "environments". Let's learn *all* about that.

Chapter 8: debug:container & How Autowiring Works

Ok, I lied. *Before* we talk about environments, I need to come clean about something: I have *not* been showing you all of the services in Symfony. Not even close.

Head over your terminal and run our *favorite* command:

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

We know that all of these services are floating around in Symfony, waiting for us to ask for them. *And* we know that bundles give us services. The Twig service down here comes from TwigBundle.

And since each service is an *object*, something *somewhere* must be responsible for *instantiating* these objects. The question is: "Who?" And the answer is... the service container!

Hello Service Container

It turns out that all of the services aren't really... "floating around": they all live inside something called the "container". And there are *way* more services in the container than `debug:autowiring` has been telling us about. Ooh... secrets! This time, run:

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

And... whoa! This prints out a *huge* list. It's so big, it's hard to see everything. Let me make my font smaller. Much better!

This is the full list of all of the services in our app... or in the "container". The container is basically a giant "array" where each service has a unique name that points to its service object.

For example, down here... there we go... we can see that there's a service whose unique name - or "id" is `twig`.

Knowing that the id of the Twig service is `twig` is not *usually* important, but it *is* useful to understand that each service has a unique id... and that you can see all of them inside the `debug:container` command.

The Container Creates Objects

And really, the container might be better-described as a big array of *instructions* on how to instantiate services, *if* and *when* something asks for them. For instance, the container knows *exactly* how to instantiate this Twig service. It knows that its class is `Twig\Environment`. And even though you can't see it on this list, it knows the *exact* arguments to pass to its constructor. The moment someone *needs* the Twig service, the container instantiates it and returns it.

Yup, when we autowire a service, we're basically saying:

> *“Hey container, can you please give me the HTTP Client service?”*

If nothing in our code has asked for that service yet during this request, the container will create it. But if something *has* already asked for it, then the container will simply return the one it *already* created. This means that if we ask for the HTTP Client service in *ten* different places, the container will only create and return the same *one* instance. Pretty cool!

How Autowiring Works

Anyway, `debug:container` shows us *all* of the services that the container knows how to instantiate. *But* `debug:autowiring` only shows us a *fraction* of those services. Why?

Well, it turns out that not *all* services are autowireable. Many of the items in this list are low-level services that just exist to help *other* services do their job. You'll probably never need to use these low-level services directly... and you actually *cannot* fetch them via autowiring.

But, let's back up a minute. Now that we know a bit more, we can now learn exactly *how* Symfony's autowiring system works. It's beautifully simple.

As we've seen, the container is really an array where every service has an id that points to that service object. When Symfony sees this `HttpClientInterface` type - this is the full type that it sees, thanks to our `use` statement - in order to figure out *which* service in the container it needs to pass us, it simply looks for a service whose *ID* matches this string *exactly*. Let me show you!

Scroll towards the top of this list to find... a service whose ID is `Symfony\Contracts\HttpClient\HttpClientInterface`! The *vast* majority of the services in the container use the "snake case" naming strategy. But if a service is intended for *us* to use in our code, Symfony will add an *additional* service inside that matches its class or interface name.

Thanks to that, when we type-hint `HttpClientInterface`, Symfony looks in the container for a service whose id is `Symfony\Contracts\HttpClient\HttpClientInterface`, it finds it and passes it to us.

Service Aliases

But look over on the right side: it says that this is an alias for a different service ID. An "alias" is like a symbolic link. It means that when someone asks for the `HttpClientInterface` service, Symfony will *actually* pass us this *other* service.

We can use the same logic down here for the `CacheInterface` type. If we check the list, here's the service whose id matches that type. But, in reality, it's just an alias for a service called `cache.app`. So when we autowire `CacheInterface`, the `cache.app` service is what's *actually* being passed to us.

If you're feeling unsure, here are the three big takeaways. One: there are a *ton* of service objects floating around and they all live inside something called the "container". Each service has a unique id.

Two, only a *small* percentage of these are useful to us... and those are set up so that we can autowire them. Autowiring works by looking in the container for a service whose id exactly matches the type. When we run `debug:autowiring`, it's basically just showing us the services from this list whose id is a class or interface name. Those are the "autowireable services".

The third and final takeaway is that services also have an alias system... which just means that when we ask for the `CacheInterface` service, what it will *really* give us is the service whose id is `cache.app`.

If you're wondering how we could ever use a *non-autowireable* service in our code, that's a great question! It's somewhat rare, but we *will* learn how to do that later.

Next, let's talk about using different configuration locally versus production. Let's talk about *environments*.

Chapter 9: Environments

Our application is like a machine: it's a set of services and PHP classes that do work... and ultimately render some pages. But we can make our machine work *differently* by feeding it different *configuration*.

For example, in `SongController`, we're using the `$logger` service to log some information:

src/Controller/SongController.php

```php
// ... lines 1 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'], name: 'api_songs_get_one')]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
// ... lines 16 - 27
    }
}
```

If we feed the logger some configuration that says "log everything", it will log *everything*, including low level debug messages. But if we change the config to say "only log errors", then this will *only* log errors. In other words, the same machine can behave *differently* based on our configuration. And sometimes, like with logging, we might need that configuration to be different while we're developing locally versus on production.

To handle this, Symfony has an important concept called "environments". I don't mean environments like local vs staging vs beta vs production. A Symfony environment is a *set* of configuration.

For example, you can run your code in the `dev` environment with a set of config that's designed for development. Or you can run your app in the `prod` environment with a set of config that's optimized for production. Let me show you!

The APP_ENV Variable

In the root of our project, we have a `.env` file:

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

###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=4777a99cd6c61ce84969bd1338737c38
###
```

We're going to talk more about this file later. But see this `APP_ENV=dev`? This tells Symfony that the current environment is `dev`, which is *perfect* for local development. When we deploy to production, we'll change this to `prod`. More on that in a few minutes.

But... what *difference* does that make? What happens in our app when we change this from `dev` to `prod`? To answer, let me close some folders... and open `public/index.php`:

```
public/index.php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

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

Remember: this is our front controller. It's the first file that's executed on every request. We don't really care much about this file, but its job is important: it boots up Symfony.

What's interesting is that it *reads* the `APP_ENV` value and passes it as the first argument to this `Kernel` class. And... this `Kernel` class is actually *in* our code! It lives at `src/Kernel.php`.

Cool. So what I want to know *now* is: What does the first argument to `Kernel` control?

If we open the class we find... absolutely *nothing*. It's empty. That's because the majority of the logic lives in this trait. Hold "cmd" or "control" and click `MicroKernelTrait` to open that up.

The config/packages/{ENV} Directory

The job of the `Kernel` is to load all of the services and routes in our app. If you scroll down, it has a method called `configureContainer()`. Ooh! We know what the container is now! And check out what it does! It takes this `$container` object and imports `$configDir.'/{packages}/*.{php,yaml}'`. This line says:

> *"Yo container! I want to load all of the files from the `config/packages/` directory."*

It loads all of those files, and then it passes the configuration from each to whatever bundle is defined as the root key. But what's *really* interesting for environments is this next line: `import $configDir.'/{packages}/'.$this->environment.'/*.{php,yaml}'`. If you dug a little, you'd learn that `$this->environment` is equal to the first argument that's passed to `Kernel`!

In other words, in the `dev` environment, this will be `dev`. So, in addition to the *main* config files, this will *also* load anything in the `config/packages/dev/` directory. Yup, we can add *extra* config there that *overrides* the main configuration in the `dev` environment. For example, we could add logging config that tells the logger to log *everything*!

Below this, we also load a file called `services.yaml` and, if we have it, `services_dev.yaml`. We'll talk more about `services.yaml` real soon.

The when@{ENV} Config

So, if you want to add environment-specific configuration, you can put it in the correct environment directory. But there's one *other* way. It's a pretty new feature and we saw it at the bottom of `twig.yaml`. It's the `when@` syntax:

```
config/packages/twig.yaml
// ... lines 1 - 3
when@test:
    twig:
        strict_variables: true
```

In Symfony, by default, there are *three* environments: `dev`, `prod`, and then if you run automated tests, there's an environment called `test`. Inside of `twig.yaml`, by saying, `when@test`, it means that this configuration will only be loaded if the environment is equal to `test`.

The best example of this might be in `monolog.yaml`. `monolog` is the bundle that controls the logger service. It *does* have some configuration that's used in *all* environments. But, below this, it has `when@dev`. We won't talk too much about the specific `monolog` configuration, but this controls how log messages are handled. In the `dev` environment, this says that it should log *everything* and it should log to a file, using this fancy `%kernel.logs_dir%` syntax that we'll learn about soon.

Anyways, this points to a `var/logs/dev.log` file and the `level: debug` part means that it will log *every* single message to `dev.log`... regardless of how important or unimportant that message is.

Below this, for the `prod` environment, it's quite different. The most important line is `action_level: error`. That says:

> *"Hi Ms Logger! This app probably logs a ton of messages, but I only want you to actually save messages that are an `error` importance level or higher."*

That makes sense! In production, we don't want our log files filling up with tons and tons of debug messages. With this, we only log *error* messages.

The big point is this: by using these tricks, we can configure our services differently based on the environment.

Environment-Specific Routing

And, we can even do the same thing with routes! Sometimes you have entire routes that you only want to load in a certain environment. Back in `MicroKernelTrait`, if you go down, there's a method called `configureRoutes()`. *This* is what's responsible for loading *all* of our routes... and it's very similar to the other code. It loads `$configDir.'/{routes}/*.{php,yaml}'` as well as this `dev` environment directory, if you have one. We don't.

You can also use the `when@dev` trick. This file is responsible for registering the routes used by the web debug toolbar. We *don't want* the web debug toolbar in production... so these routes are *only* imported in the `dev` environment.

```yaml
# config/routes/web_profiler.yaml
when@dev:
    web_profiler_wdt:
        resource: '@WebProfilerBundle/Resources/config/routing/wdt.xml'
        prefix: /_wdt

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

Heck, certain *bundles* are only enabled in some environments! If you open `config/bundles.php`, we have the name of the bundle... and then on the right, the environments in which that bundle should be enabled. This `all` means *all* environments.... and *most* are enabled in *all* environments.

The `WebProfilerBundle` however - the bundle that gives us the web debug toolbar and profiler - is *only* loaded in the `dev` and `test` environments. Yup, the entire bundle - and the services it provides - are *never* loaded in the `prod` environment.

So, now that we understand the basics of environments, let's see if we can switch our application to the `prod` environment. And then, as a challenge, we'll configure our cache service *differently* in `dev`. That's next.

Chapter 10: The "prod" Environment

Our app is currently running in the `dev` environment. Let's switch it to `prod`... which is what you would use on production. Temporarily change `APP_ENV=dev` to `prod`:

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

then head over and refresh. Whoa! The web debug toolbar is *gone*. That... makes sense! The entire web profiler bundle is *not* enabled in the `prod` environment.

You'll also notice that the dump from our controller appears on the *top* of the page. The web profiler normally *captures* that and displays it down on the web debug toolbar. But... since that whole system isn't enabled anymore, it now dumps right where you call it.

And there are a lot of other differences, like the logger, which now behaves differently thanks to the configuration in `monolog.yaml`.

Clearing the prod Cache

The way pages are built has also changed. For example, Symfony caches a lot of files... but you don't notice that in the `dev` environment. That's because Symfony is super smart and rebuilds that cache automatically when we change certain files. *However*, in the `prod` environment, that doesn't happen.

Check it out! Open up `templates/base.html.twig`... and change the title on the page to `Stirred Vinyl`. If you go back over and refresh... look up here! *No change*! The Twig templates *themselves* are cached. In the `dev` environment, Symfony rebuilds that cache *for* us. But in the `prod` environment? Nope! We need to clear it manually.

How? At your terminal, run:

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

Notice it says that it's clearing the cache for the *prod* environment. So, just like how our *app* always runs in a specific environment, the console commands *also* run in a specific environment. And, it reads that same `APP_ENV` flag. So because we have `APP_ENV=prod` here, `cache:clear` knew that it should run in the *prod* environment and clear the cache for *that* environment.

Thanks to this, when we refresh... *now* the title updates. I'll change this back to our *cool* name, `Mixed Vinyl`.

Changing the Cache Adapter for prod Only

Let's try something else! Open up `config/packages/cache.yaml`. Our cache service currently uses the `ArrayAdapter`, which is a fake cache. That might be cool for development, but it won't be much help on production:

config/packages/cache.yaml

```yaml
framework:
    cache:
// ... lines 3 - 10
        app: cache.adapter.array
// ... lines 12 - 25
```

Let's see if we can switch that back to the filesystem adapter, but *only* for the `prod` environment. How? Down here, use `when@prod` and then repeat the same keys. So `framework`, `cache`, and then `app`. Set this to the adapter we want, which is called `cache.adapter.filesystem`:

```yaml
config/packages/cache.yaml
framework:
    cache:
// ... lines 3 - 10
        app: cache.adapter.array
// ... lines 12 - 20
when@prod:
    framework:
        cache:
            app: cache.adapter.filesystem
```

It's going to be *really* easy to see if this works because we're still dumping the cache service in our controller. Right now, it's an `ArrayAdapter`. If we refresh... surprise! It's *still* an `ArrayAdapter`. Why? Because we're in the *prod* environment... and pretty much any time you make a change in the `prod` environment, you need to rebuild your cache.

Go back to your terminal and run

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

again and now... got it - `FilesystemAdapter`!

But... let's *reverse* this config. Copy `cache.adapter.array` and change it to `filesystem`. We'll use *that* by default. Then at the bottom, change to `when@dev`, and *this* to `cache.adapter.array`:

```yaml
config/packages/cache.yaml
framework:
    cache:
// ... lines 3 - 10
        app: cache.adapter.filesystem
// ... lines 12 - 20
when@dev:
    framework:
        cache:
            app: cache.adapter.array
```

Why am I doing that? Well, that literally makes zero difference in the `dev` and `prod` environments. *But* if we decide to start writing tests later, which run in the *test* environment, with this new config, the test environment will use the same cache service as production... which is probably more realistic and better for testing.

To make sure this still works, clear the cache one more time. Refresh and... it does! We still have `FilesystemAdapter`. And... if we switch back to the `dev` environment in `.env`:

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

and refresh... yes! The web debug toolbar is back, and down here, we are once again using `ArrayAdapter`!

Now, in reality, you probably won't ever switch to the `prod` environment while you're developing locally. It's hard to work with... and there's just no point! The `prod` environment is really meant for production! And so, you *will* run that `bin/console cache:clear` command during deployment... but probably almost never on your local machine.

Before we go on, head into `VinylController`, go down to `browse()`, and take out that `dump()`:

```
src/Controller/VinylController.php
// ... lines 1 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 32
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;

        $mixes = $cache->get('mixes_data', function(CacheItemInterface $cacheItem) use ($httpClient) {
            $cacheItem->expiresAfter(5);
            $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');

            return $response->toArray();
        });

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre,
            'mixes' => $mixes,
        ]);
    }
}
```

Okay, status check! First, *everything* in Symfony is done by a service. Second, bundles *give* us services. And third, we can control *how* those services are instantiated via the different bundle configuration in `config/packages/`.

Now, let's go one important step further by creating our *own* service.

Chapter 11: Creating a Service

We know that bundles give us services and services do work. Ok. But what if we need to write our *own* custom code that does work? Should we... put that into our *own* service class? Absolutely! And it's a *great* way to organize your code.

We *are* already doing some work in our app. In the `browse()` action:

```
src/Controller/VinylController.php
// ... lines 1 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 32
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;

        $mixes = $cache->get('mixes_data', function(CacheItemInterface $cacheItem) use ($httpClient) {
            $cacheItem->expiresAfter(5);
            $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');

            return $response->toArray();
        });

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre,
            'mixes' => $mixes,
        ]);
    }
}
```

we make an HTTP request and cache the result. Putting this logic in our controller is *fine*. But by moving it into its own service class, it'll make the *purpose* of the code more clear, allow us to reuse it from multiple places... and even enable us to unit test that code if we want to.

Creating the Service Class

That sounds *amazing*, so let's do it! How do we create a service? In the `src/` directory, create a new PHP class *wherever* you want. It seriously doesn't matter what directories or subdirectories you create in `src/`: do whatever feels good for you.

For this example, I'll create a `Service/` directory - though again, you could call that `PizzaParty` or `Repository` - and inside of *that*, a new PHP class. Let's call it... how about `MixRepository`. "Repository" is a pretty common name for a service that returns data. Notice that when I create this, PhpStorm *automatically* adds the correct namespace. It doesn't matter *how* we organize our classes inside of `src/`... as long as our namespace matches the directory:

```
src/Service/MixRepository.php
<?php

namespace App\Service;
// ... lines 4 - 5

class MixRepository
{
// ... lines 9 - 17
}
```

One important thing about service classes: they have *nothing* to do with Symfony. Our controller class is a Symfony concept. But `MixRepository` is a class *we're* creating to organize our *own* code. That means... there are no rules! We don't need to extend a base class or implement an interface. We can make this class look and feel however we want. The power!

With that in mind, let's create a new `public function` called, how about, `findAll()` that will `return` an `array` of all of the mixes in our system. Back in `VinylController`, copy all of the logic that fetches the mixes and paste that here:

```
src/Service/MixRepository.php
// ... lines 1 - 4
use Psr\Cache\CacheItemInterface;

class MixRepository
{
    public function findAll(): array
    {
        $mixes = $cache->get('mixes_data', function(CacheItemInterface $cacheItem) use ($httpClient) {
            $cacheItem->expiresAfter(5);
            $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');

            return $response->toArray();
        });
    }
}
```

PhpStorm will ask if we want to add a `use` statement for the `CacheItemInterface`. We *totally* do! Then, instead of creating a `$mixes` variable, just `return`:

```
src/Service/MixRepository.php
// ... lines 1 - 4
use Psr\Cache\CacheItemInterface;

class MixRepository
{
    public function findAll(): array
    {
        return $cache->get('mixes_data', function(CacheItemInterface $cacheItem) use ($httpClient) {
            $cacheItem->expiresAfter(5);
            $response = $httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');

            return $response->toArray();
        });
    }
}
```

There *are* some undefined variables in this class... and those *will* be a problem. But ignore them for a minute: I *first* want to see if we can *use* our shiny new `MixRepository`.

Is our Service already in the Container?

Head into `VinylController`. Let's think: we somehow need to tell Symfony's service container about our new service so that we can then *autowire* it in the same way we're autowiring core services like `HtttpClientInterface` and `CacheInterface`.

Whelp, I have a surprise! Spin over to your terminal and run:

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

Scroll up to the top and... amaze! `MixRepository` is *already* a service in the container! Let me explain two things here. First, the `--all` flag is not *that* important. It *basically* tells this command to show you the core services like `$httpClient` and `$cache`, *plus* our own services like `MixRepository`.

Second, the container... *somehow* already saw our repository class and recognized it as a service. We'll learn *how* that happened in a few minutes... but for now, it's enough to know that our new `MixRepository` *is* already inside the container *and* its service *id* is the full class name. *That* means we can autowire it!

Autowiring the new Service

Back over in our controller, add a third argument type-hinted with `MixRepository` - hit tab to add the `use` statement - and call it... how about `$mixRepository`:

```
src/Controller/VinylController.php
// ... lines 1 - 4
use App\Service\MixRepository;
// ... lines 6 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 33
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, MixRepository $mixRepository, string $slug = null): Response
    {
// ... lines 36 - 43
    }
}
```

Then, down here, we don't need any of this `$mixes` code anymore. Replace it with `$mixes = $mixRepository->findAll()`:

```
src/Controller/VinylController.php
// ... lines 1 - 4
use App\Service\MixRepository;
// ... lines 6 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 33
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, MixRepository $mixRepository, string $slug = null): Response
    {
// ... lines 36 - 37
        $mixes = $mixRepository->findAll();
// ... lines 39 - 43
    }
}
```

How nice is that? Will it work? Let's find out! Refresh and... it *does*! Ok, working in this case means that we get an `Undefined variable $cache` message coming from `MixRepository`. But the fact that our code *got* here means that autowiring `MixRepository` *worked*: the container saw this, *instantiated* `MixRepository` and passed it to us so that we could use it.

So, we created a service and made it available for autowiring! We are *so* cool! But our new service needs the `$httpClient` and `$cache` services in order to do its job. How do we get those? The answer is one of the *most* important concepts in Symfony and object-oriented coding in general: dependency injection. Let's talk about that next.

Chapter 12: Dependency Injection

Our `MixRepository` service is *sort of* working. We can autowire it into our controller and the container is *instantiating* the object and passing it to us. We *prove* that over here because, when we run the code, it successfully calls the `findAll()` method.

But.... then it explodes. That's because, inside `MixRepository` we have two undefined variables. In order for our class to do its job, it needs two services: the `$cache` service and the `$httpClient` service.

Autowiring to Methods is a Controller-Only Superpower

I keep saying that there are many services floating around inside of Symfony, waiting for us to use them. That's *true*. *But*, you can't just grab them out of thin air from anywhere in your code. For example, there's no `Cache::get()` static method that you can call whenever you want that will return the `$cache` service object. Nothing like that exists in Symfony. And that's good! Allowing us to grab objects out of thin air is a recipe for writing bad code.

So how *can* we get access to these services? Currently, we only know one way: by autowiring them into our controller. *But* that *won't* work here. Autowiring services into a *method* is a superpower that *only* works for controllers.

Watch: if we added a `CacheInterface` argument... then went over and refreshed, we'd see:

> *"Too few arguments to function [...]findAll(), 0 passed [...] and exactly 1 expected."*

That's because *we* are calling `findAll()`. So if `findAll()` needs an argument, it is *our* responsibility to pass them: there's no Symfony magic. My point is: autowiring works in controller methods, but don't expect it to work for any *other* methods.

Manually Passing Services to a Method?

But one way we *might* get this to work is by adding both services to the `findAll()` method and then *manually* passing them in from the controller. This won't be the final solution, but let's try it.

I already have a `CacheInterface` argument... so now add the `HttpClientInterface` argument and call it `$httpClient`:

```
src/Service/MixRepository.php
// ... lines 1 - 5
use Symfony\Contracts\Cache\CacheInterface;
use Symfony\Contracts\HttpClient\HttpClientInterface;

class MixRepository
{
    public function findAll(HttpClientInterface $httpClient, CacheInterface $cache): array
    {
// ... lines 13 - 18
    }
}
```

Perfect! The code in this method is now happy.

Back over in our controller, for `findAll()`, pass `$httpClient` and `$cache`:

```
src/Controller/VinylController.php
// ... lines 1 - 12
class VinylController extends AbstractController
{
// ... lines 15 - 33
    public function browse(HttpClientInterface $httpClient, CacheInterface $cache, MixRepository $mixRepository, string $slug = null): Response
    {
// ... lines 36 - 37
        $mixes = $mixRepository->findAll($httpClient, $cache);
// ... lines 39 - 43
    }
}
```

And now... it works!

"Dependencies" Versus "Arguments"

So, on a high level, this solution makes sense. We know that we can autowire services into our controller... and then we just pass them into `MixRepository`. But if you think a bit deeper, the `$httpClient` and `$cache` services aren't really *input* to the `findAll()` function. They don't really make sense as arguments.

Let's look at an example. Pretend that we decide to change the `findAll()` method to accept a `string $genre` argument so the method will *only* return mixes for *that* genre. This argument makes perfect sense: passing different genres changes what it returns. The argument *controls* how the method *behaves*.

But the `$httpClient` and `$cache` arguments *don't* control how the function behaves. In reality, we would pass these *same* two values *every* time we call the method... *just* so things *work*.

Instead of arguments, these are really *dependencies* that the service *needs*. They're just stuff that *must* be available so that `findAll()` can do its job!

Dependency Injection & The Constructor

For "dependencies" like this, whether they're service objects or static configuration that your service needs, instead of passing them to the methods, we pass them into the *constructor*. Delete that pretend `$genre` argument... then add a `public function __construct()`. Copy the two arguments, delete them, and move them up here:

```
src/Service/MixRepository.php
// ... lines 1 - 8
class MixRepository
{
// ... lines 11 - 13
    public function __construct(HttpClientInterface $httpClient,
CacheInterface $cache)
    {
// ... lines 16 - 17
    }
// ... lines 19 - 28
}
```

Before we finish this, I need to tell you that autowiring works in *two* places. We already know that we can autowire arguments into our controller methods. But we can *also* autowire arguments into the `__construct()` method of any service. In fact, that's the *main* place that autowiring is meant to work! The fact that autowiring also works for controller methods is... kind of an "extra" just to make life nicer.

Anyways, autowiring works in the `__construct()` method of our services. So as long as we type-hint the arguments (and we have), when Symfony instantiates our service, it will pass us these two services. Yay!

And what do we *do* with these two arguments? We set them onto properties.

Create a `private $httpClient` property and a `private $cache` property. Then, down in the constructor, assign them: `$this->httpClient = $httpClient`, and `$this->cache = $cache`:

```
src/Service/MixRepository.php
// ... lines 1 - 8
class MixRepository
{
    private $httpClient;
    private $cache;

    public function __construct(HttpClientInterface $httpClient,
CacheInterface $cache)
    {
        $this->httpClient = $httpClient;
        $this->cache = $cache;
    }
// ... lines 19 - 28
}
```

So when Symfony instantiates our `MixRepository`, it passes us these two arguments and we store them on properties so we can use them later.

Watch! Down here, instead of `$cache`, use `$this->cache`. And then we don't need this `use ($httpClient)` over here... because we can say `$this->httpClient`:

```
src/Service/MixRepository.php
// ... lines 1 - 8
class MixRepository
{
// ... lines 11 - 19
    public function findAll(): array
    {
        return $this->cache->get('mixes_data', function(CacheItemInterface $cacheItem) {
// ... line 23
            $response = $this->httpClient->request('GET', 'https://raw.githubusercontent.com/SymfonyCasts/vinyl-mixes/main/mixes.json');
// ... lines 25 - 26
        });
    }
}
```

This service is now in *perfect* shape.

Back over in `VinylController`, now we can simplify! The `findAll()` method doesn't need any arguments... and so we don't even need to autowire `$httpClient` or `$cache` at all. I'm going to celebrate by removing those `use` statements on top:

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(MixRepository $mixRepository, string $slug = null): Response
    {
// ... lines 34 - 35
        $mixes = $mixRepository->findAll();
// ... lines 37 - 41
    }
}
```

Look how much easier that is! We autowire the *one* service we need, call the method on it, and... it even *works*! *This* is how we write services. We add any dependencies to the constructor, set them onto properties, and then *use* them.

Hello Dependency Injection!

By the way, what we just did has a fancy schmmancy name: "Dependency injection". But don't run away! That may be a scary... or at least "boring sounding" term, but it's a very simple concept.

When you're inside of a service like `MixRepository` and you realize you need *another* service (or maybe some config like an API key), to get it, create a constructor, add an argument for the thing you need, set it onto a property, and then use it down in your code. Yep! *That's* dependency injection.

Put simply, dependency injection says:

> *“If you need something, instead of grabbing it out of thin air, force Symfony to pass it to you via the constructor.”*

This is one of *the* most important concepts in Symfony... and we'll do this over and over again.

PHP 8 Property Promotion

Okay, *unrelated* to dependency injection and autowiring, there are two minor improvements that we can make to our service. The first is that we can add *types* to our properties: `HttpClientInterface` and `CacheInterface`:

src/Service/MixRepository.php

```php
// ... lines 1 - 8
class MixRepository
{
// ... lines 11 - 13
    public function __construct(HttpClientInterface $httpClient, CacheInterface $cache)
    {
        $this->httpClient = $httpClient;
        $this->cache = $cache;
    }
// ... lines 19 - 28
}
```

That doesn't change how our code works... it's just a nice, responsible way to do things.

But we can go further! In PHP 8, there's a new, shorter syntax for creating a property and setting it in the constructor like we're doing. It looks like this. First, I'll move my arguments onto multiple lines... just to keep things organized. Now add the word `private` in front of each argument. Finish by deleting the properties... as well as the inside of the method.

That might look weird at first, but as soon as you add `private`, `protected`, or `public` in front of a `__construct()` argument, that creates a property with this name and sets the argument *onto* that property:

```
src/Service/MixRepository.php
// ... lines 1 - 8
class MixRepository
{
    public function __construct(
        private HttpClientInterface $httpClient,
        private CacheInterface $cache
    ) {}
// ... lines 15 - 24
}
```

So it looks different, but it's the *exact* same as what we had before.

When we try it... yup! It still works.

Next: I keep saying that the container holds *services*. That's *true*! But it also holds one other thing - simple configuration called "parameters".

Chapter 13: Parameters

We know there's this *container* concept that holds all of our services... and we can see the full list of services by running:

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

Listing Parameters

Well, it turns out that the container holds one *other* thing: grudges. Seriously, don't expect to pull a prank on the service container and get away with it.

Ok, what it *really* holds, in addition to services, is *parameters*. These are simple configuration values, and we can see them by running a similar command:

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

These are basically variables that you can read and reference in your code. We don't need to worry about most of these, actually. They're *set* by internal things and *used* by internal things. But there *are* a few that start with `kernel` that are pretty interesting, like `kernel.project_dir`, which points to the *directory* of our project. Yep! If you ever need a way to refer to the directory of your app, *this* parameter can help.

Fetching Parameters from a Controller

So... how do we *use* these parameters? There are two ways. First, it's not super common, but you *can* fetch a parameter in your controller. For example, in `VinylController`, let's `dd($this->getParameter())` - which is a shortcut method from `AbstractController` -

and then `kernel.project_dir`. We even get some nice auto-completion thanks to the Symfony PhpStorm plugin!

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(MixRepository $mixRepository, string $slug = null): Response
    {
        dd($this->getParameter('kernel.project_dir'));
// ... lines 35 - 42
    }
}
```

And when we try it... yep! There it is!

Referencing Parameters with %parameter%

Now... delete that. This *works*, but most of the time, the way you'll use parameters is by referencing them in your *configuration files*. And we've seen this before! Open up `config/packages/twig.yaml`:

```
config/packages/twig.yaml
twig:
    default_path: '%kernel.project_dir%/templates'
// ... lines 3 - 7
```

Remember that `default_path`? That's referencing the `kernel.project_dir` parameter. When you're in *any* of these `.yaml` configuration files and you want to reference a parameter, you can use this special syntax: `%`, the name of the parameter, then another `%`.

Creating a new Parameter

Open up `cache.yaml`. We're setting `cache.adapter` to `filesystem` for *all* environments. Then, we're overriding it to be the `array` adapter in the dev environment only. Let's see if we can shorten this by creating a *new* parameter.

How *do* we create parameters? In any of these files, add a root key called `parameters`. Below that, you can just... invent a name. I'll call it `cache_adapter`, and set that to our value: `cache.adapter.filesystem`:

```
config/packages/cache.yaml
parameters:
    cache_adapter: 'cache.adapter.filesystem'
// ... lines 3 - 28
```

If you have a root `framework` key, Symfony will pass all of the config to FrameworkBundle. The same is true with the `twig` key and TwigBundle.

But `parameters` is special: anything under this will create a *parameter*.

So yea... we now have a new `cache.adapter` parameter... that we're not actually *using* yet. But we can already *see* it! Run:

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

Near the top... there it is - `cache_adapter`! To use this, down here for `app`, say `%cache_adapter%`:

```
config/packages/cache.yaml
parameters:
    cache_adapter: 'cache.adapter.filesystem'
// ... line 3
framework:
    cache:
// ... lines 6 - 13
        app: '%cache_adapter%'
// ... lines 15 - 28
```

That's *it*. Quick note: You may have noticed that *sometimes* I use quotes in YAML and *sometimes* I *don't*. Mostly, in YAML, you don't need to use quotes... but you always *can*. And if you're ever not sure if they're needed or not, better to be safe and *use* them.

Parameters *are* actually one example where quotes are *required*. If we didn't surround this with quotes, it would look like a special YAML syntax and throw an error.

Anyway, in the `dev` environment, instead of saying `framework`, `cache`, and `app`, all we need to do is override that parameter. I'll say `parameters`, then `cache_adapter`... and set it to `cache.adapter.array`:

```yaml
config/packages/cache.yaml
parameters:
    cache_adapter: 'cache.adapter.filesystem'
// ... line 3
framework:
    cache:
// ... lines 6 - 13
        app: '%cache_adapter%'
// ... lines 15 - 23
when@dev:
    parameters:
        cache_adapter: 'cache.adapter.array'
```

To see if that's working, spin over here and run another helper command:

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

Remember, `debug:config` will show you what your *current* configuration is under the `framework` key, and then the `cache` sub-key. And you can see here that `app` is set to `cache.adapter.array` - the resolved value for the parameter.

Let's check the value in the prod environment... just to make sure it's right there too. When you run any `bin/console` command, that command will execute in the same environment your app is running in. So when we ran `debug:config`, that ran in the *dev* environment.

To run the command in the *prod* environment, we *could* go over here and change `APP_ENV` to `prod` temporarily... but there's an *easier* way. You can *override* the environment when running any command by adding a flag at the end. For example:

```
php bin/console debug:config framework cache --env=prod
```

But before we try that, we always need to clear our cache first to see changes in the `prod` environment. Do *that* by running:

```
php bin/console cache:clear --env=prod
```

Now try:

```
php bin/console debug:config framework cache --env=prod
```

And... beautiful! It shows `cache.adapter.filesystem`. So, the container *also* holds parameters. This isn't a *super* important concept in Symfony, so, as long as you understand how they work, you're good.

Ok, let's turn back to dependency injection. We know that we can autowire services into the constructor of a service or into controller methods. But what if we need to pass something that's *not* autowireable? Like, what if we wanted to pass one of these *parameters* to a service? Let's find out how that works *next*.

Chapter 14: Manual Service Config in services.yaml

At your terminal, run:

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

One of the `kernel` parameters is called `kernel.debug`. In addition to environments, Symfony has this concept of "debug mode". It's *true* for the `dev` environment and *false* for `prod`. And, occasionally, it comes in handy!

Here's our new *challenge* (mostly to see if we can do it). Inside of `MixRepository`, I want to figure out if we're in debug mode. If debug mode is *true*, we will cache for *5 seconds*. If it's *false*, I want to cache for *60 seconds*:

src/Service/MixRepository.php

```php
// ... lines 1 - 8
class MixRepository
{
    public function __construct(
// ... lines 12 - 13
        private bool $isDebug
    ) {}

    public function findAll(): array
    {
        return $this->cache->get('mixes_data', function(CacheItemInterface $cacheItem) {
            $cacheItem->expiresAfter($this->isDebug ? 5 : 60);
// ... lines 21 - 23
        });
    }
}
```

Dependency Injection!

Let's back up for a minute. Suppose you're working inside of a service like `MixRepository`. Suddenly you realize that you need some *other* service like the logger. What do you do to get the logger? The answer: you do the dependency injection dance. You add a `private LoggerInterface $logger` argument and property... then you use it down in your code. You'll do this *tons* of times in Symfony.

Let me undo that... because we don't actually need the logger right now. But what we *do* need is similar. Right now we're inside of a service and we've suddenly realized that we need some configuration (the `kernel.debug` flag) to do our work. What do we do to get that config? The *same* thing! Add that as an argument to our constructor. Say `private bool $isDebug`, and down here, use it: if `$this->isDebug`, cache for 5 seconds, else cache for 60 seconds.

Non-Autowireable Arguments

But... there's a slight complication... and I bet you already know what it is. When we refresh the page... yikes! We get a `Cannot resolve argument` error. If you skip a bit, it says:

> *“Cannot autowire service `App\Service\MixRepository`: argument `$isDebug` of method `__construct()` is type-hinted `bool`, you should configure its value explicitly.”*

That makes sense. Autowiring only works for services. You can't have a bool `$isDebug` argument and expect Symfony to somehow realize that we want the `kernel.debug` parameter. I might be a wizard, but I don't have a spell for that. I *can* make a whole slice of pie disappear, though. With magic. Definitely.

Configuring MixRepository in services.yaml

How do we fix this? Open a file that we haven't looked at yet: `config/services.yaml`:

```yaml
config/services.yaml
# This file is the entry point to configure your own services.
# Files in the packages/ subdirectory configure your dependencies.

# Put parameters here that don't need to change on each machine where the app is deployed
# https://symfony.com/doc/current/best_practices.html#use-parameters-for-application-configuration
parameters:

services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App\:
        resource: '../src/'
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'

    # add more service definitions when explicit configuration is needed
    # please note that last definitions always *replace* previous ones
```

So far, we haven't needed to add any configuration for our `MixRepository` service. The container saw the `MixRepository` class as soon as we created it... and autowiring helped the container know which arguments to pass to the constructor. But now that we have a non-autowireable argument, we need to give the container a *hint*. And we do that in this file.

Head down to the bottom and add the full namespace of this class: `App\Service\MixRepository`:

```yaml
config/services.yaml
// ... lines 1 - 7
services:
// ... lines 9 - 25
    App\Service\MixRepository:
// ... lines 27 - 29
```

Below that, use the word `bind`. And below *that*, give the container a *hint* to tell it what to pass to the argument by saying `$isDebug` set to `%kernel.debug%`:

```yaml
config/services.yaml
// ... lines 1 - 7
services:
// ... lines 9 - 25
    App\Service\MixRepository:
        bind:
            '$isDebug': '%kernel.debug%'
```

I'm using `$isDebug` on purpose. That needs to *exactly* match the name of the argument in the class. Thanks to this, the container will pass the `kernel.debug` parameter value.

And when we try it... it works! The two service arguments are still autowired, but we filled in the one *missing* argument so that the container can instantiate our service. Nice!

I want to talk more about the purpose of this file and all of the configuration up here. It turns out that a lot of the magic we've been seeing related to services and autowiring can be explained by this code. That's *next*.

Chapter 15: All About services.yaml

When Symfony first boots up, it needs to get the full list of all of the services that should be in the container. That includes the service ID, its class name, and all of its constructor arguments. The first and biggest source of services are *bundles*. If you run

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

the vast majority of these services come from bundles. The *second* place the container gets services from is *our* code. And to learn about our services, Symfony reads `services.yaml`.

The Special _defaults Section

At the moment that Symfony starts parsing the first line of this file, *nothing* in our `src/` directory has been registered as a service in the container. This is really important. Adding our classes to the container is, in fact, the *job* of this file! And the way it does it is pretty amazing. Let's take a tour!

Notice that the config is under a `services` key. Like `parameters`, this is a special key. And, like its name suggests, anything under this is meant to configure *services*.

The first *sub-key* under this is `_defaults`. `_defaults` is a magic key that allows us to define some *default* options that will be added to *all* services that are registered in this file. So *every* service that we register below will *automatically* have `autowire: true` and `autoconfigure: true`:

```yaml
config/services.yaml
// ... lines 1 - 7
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
// ... lines 13 - 29
```

Let's look at an example. The most *basic* thing you can do under the `services` key is... register a service! That's what we're doing at the bottom. This tells the container that there should be an `App\Service\MixRepository` service in the container *and* we specified one option: `bind`.

```yaml
config/services.yaml
// ... lines 1 - 7
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.

// ... lines 14 - 24

    App\Service\MixRepository:
        bind:
            '$isDebug': '%kernel.debug%'
```

Services can actually have a *bunch* of options, including `autowire` and `autoconfigure`. So it would be *totally* legal to say, `autowire: true` and `autoconfigure: true` right here. This would work *just* fine. But thanks to the `_defaults` section, those aren't needed! The `_defaults` says:

> *"Unless it's been overridden on a specific service, set `autowire` and `autoconfigure` to `true` for all services in this file."*

And what does `autowire` *do*? Simple! It tells Symfony's container:

> *"Hey! Please try to guess my constructor arguments by looking at their type-hints."*

This feature is pretty awesome... which is why it's automatically turned on for all of our services. The other option - `autoconfigure` - is more subtle and we'll talk about it later.

Service Auto-Registration

All right, by the time we get to the `_defaults` line, we've established some default configuration... but we *haven't* actually registered any services yet. That's the job of the next section... and it's the key to *everything*:

```yaml
config/services.yaml
// ... lines 1 - 7
services:
// ... lines 9 - 13
    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App\:
        resource: '../src/'
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'
// ... lines 22 - 29
```

This special syntax says

> *"Please look inside the `src/` directory and automatically register all PHP classes as a service... except for these three things."*

This is why, *immediately* after we created the `MixRepository` class, it was *already* in the container! And thanks to the `_defaults` section, any services registered by this will *automatically* have `autowire: true` and `autoconfigure: true`. That's some serious team work! This mechanism is called "Service Auto-Registration".

But remember, every service in the container needs to have a unique ID. If you look back at `debug:container`, most of the service IDs are snake case. Let me zoom out a bit so it's easier to see. Better! So, for example, the `Twig` service has the snake case `twig` ID. But if you scroll up to the top of this list, our `MixRepository` ID is... the full *class* name.

Yep! When you use Service Auto-Registration, it uses the class name as *both* the class *and* the service *ID*. This is done for simplicity... but *also* for autowiring. When we try to autowire `MixRepository` into our controller or anywhere else, to figure out which service to pass us, Symfony will look for a service whose ID exactly matches `App\Service\MixRepository`. So Service Auto-Registration not only registers our classes as services, it does it in a way that makes them *autowireable*. That's awesome!

Auto-Registration of Non-Services?

Anyway, after this section here, *every* class in `src/` is now registered as a service in the container. Except, well... we don't want *every* class in `src/` to be a service.

There are really two types of classes in your app: "Service classes" that do work, and "model classes" - sometimes called "DTOs" - whose job is mostly to hold data - like a `Product` class with `name` and `price` properties. We want the container to handle instantiating our services. But for model classes, *we* will create them whenever we need them - like with `$product = new Product()`. So, these will *not* be services in the container.

In the next tutorial, we'll create Doctrine entity classes, which are model classes for the database. These will live in the `src/Entity/` directory... and since they're not meant to be services, that directory is excluded. So we register *everything* in the `src/` directory as a service, *except* for these three things.

But.. fun fact! This `exclude` key is *not* that important. Heck, you could delete it and everything would *still* work! If you accidentally register something as a service that isn't *meant* to be a service, *no worries*! Since you'll never try to autowire and *use* that class like a service, Symfony will realize it's not being used and remove it from the container. Dang, that is smart!

Custom Service Configuration

So everything in `src/` is automatically registered as a service without us needing to do anything or touch this file.

But... occasionally, you'll need to add *extra* config to a *specific* service. That's what happened with `MixRepository` thanks to its non-autowireable `$isDebug` argument.

To fix that, at the bottom of this file, we're registering a new service whose ID and class is `App\Service\MixRepository`. This will actually *override* the service that was created during Service Auto-Registration, since both IDs will match `App\Service\MixRepository`. So, we're defining a *brand new* service.

But thanks to `_defaults`, it automatically has `autowire: true` and `autoconfigure: true`. Then we add the additional `bind` option.

So the only thing we need to put at the bottom of this file are services that need *additional* configuration to work. And... there's actually a *cooler* way to fix non-autowireable arguments that I'll show you next.

All Configuration Files are Equals!

But before we get to that, I want to mention one more thing: this file, `services.yaml`, is loaded via the *same* system that loads all of the files in `config/packages/`. In fact, there's no technical difference between this file and say... `framework.yaml`. That's right! If we wanted to, we could copy and delete the contents of `services.yaml`, paste them into `framework.yaml`, and everything would work *exactly* the same.

Except that... we would need to, y'know, just correct these paths since we're one directory deeper. Watch! I'll move this around real quick and... this still works just fine! Cool! Let's put that back the way it was and... there we go.

The only reason we have a `service.yaml` file is for organization. It feels good to have *one* file to "configure your services". The truly important thing is that all of this config lives under the `services` key. In fact, near the top of this file, you'll notice there's an empty `parameters` key.

In `cache.yaml`, we created a `parameters` key *there* to register a new parameter. It's really up to us to decide *where* we want to define this parameter. We can do it in `cache.yaml` or, to keep all parameters in one spot, we could copy this and move it over to `services.yaml`.

In `cache.yaml`, I'll also grab the `when@dev`, delete that, and paste it into `services.yaml`:

```yaml
config/services.yaml
// ... lines 1 - 5
parameters:
    cache_adapter: 'cache.adapter.filesystem'

when@dev:
    parameters:
        cache_adapter: 'cache.adapter.array'
// ... lines 12 - 34
```

On a technical level, that makes no difference and our app *still* works. But I like this better. Services and parameters are a global idea in your app... so it's nice to organize them all in one file.

All right, the only reason we wrote any code at the bottom of `services.yaml` was to tell the container what to pass to the non-autowireable `$isDebug` argument. But what if I told you there's a more *automatic* way to solve these problematic arguments? That's *next*.

Chapter 16: Bind Arguments Globally

In practice, you rarely need to do anything inside of `services.yaml`. Most of the time, when you add an argument to the constructor of a service, it's autowireable. So you add the argument, give it a type-hint... and *keep* coding!

```
src/Service/MixRepository.php
// ... lines 1 - 8
class MixRepository
{
    public function __construct(
        private HttpClientInterface $httpClient,
        private CacheInterface $cache,
        private bool $isDebug
    ) {}
// ... lines 16 - 25
}
```

But the `$isDebug` argument is *not* autowireable... since it's not a service. And *that* forced us to *completely* override the service so we could specify that *one* argument with `bind`. It works but... that was... kind of a lot of typing to do such a small thing!

```
config/services.yaml
// ... lines 1 - 12
services:
// ... lines 14 - 30
    App\Service\MixRepository:
        bind:
            '$isDebug': '%kernel.debug%'
```

Moving bind to `_defaults`

So here's a *different* solution. Copy that `bind` key, delete the service entirely, and up, under `_defaults`, paste:

```
config/services.yaml
// ... lines 1 - 12
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
        bind:
            '$isDebug': '%kernel.debug%'
// ... lines 20 - 32
```

When we move over and try this... the page *still* works! How cool is that? And, it makes sense. This section will automatically register `MixRepository` as a service... and then anything under `_defaults` will be applied *to* that service. So the end result is exactly what we had before.

I *love* doing this! It allows me to set up project-wide conventions. Now that we have this, we could add an `$isDebug` argument to the constructor of *any* service and it will instantly work.

Binding with Type_hints

By the way, if you want, you can *also* include the *type* with the bind.

So this would now *only* work if we use the `bool` type-hint with the argument:

```
config/services.yaml
// ... lines 1 - 12
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
        bind:
            'bool $isDebug': '%kernel.debug%'
// ... lines 20 - 32
```

If we used `string`, for example, Symfony would *not* try to pass in that value.

The Autowire Attribute

So the global bind is *awesome*. But starting in Symfony 6.1, there's *another* way to specify a non-autowireable argument. Comment out the global `bind`. I *do* still like doing this... but let's try the new way:

```
config/services.yaml
// ... lines 1 - 12
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
#        bind:
#            'bool $isDebug': '%kernel.debug%'
// ... lines 20 - 32
```

If we refresh now, we get an error because Symfony doesn't know what to pass to the `$isDebug` argument. To fix that, go into `MixRepository` and, above the argument (or before the argument if you're not using multiple lines), add a PHP 8 attribute called `Autowire`. Normally, PHP 8 attributes will auto-complete, but this *isn't* auto-completing for me. That's actually due to a bug in PhpStorm. To get around this, I'm going to type out `Autowire`... then go to the top and start adding the `use` statement for this manually, which *does* give us an option to auto-complete. Hit "tab" and... *tah dah*! If you want to make them alphabetical, you can move it around.

You may also notice that it's underlined with a message:

> *“Attribute cannot be applied to a property [...]”*

Again, PhpStorm is a bit confused because this is both a property *and* an argument.

Anyway, go ahead and pass this an argument `%kernel.debug%`:

```
src/Service/MixRepository.php
// ... lines 1 - 6
use Symfony\Component\DependencyInjection\Attribute\Autowire;
// ... lines 8 - 9
class MixRepository
{
    public function __construct(
// ... lines 13 - 14
        #[Autowire('%kernel.debug%')]
        private bool $isDebug
    ) {}
// ... lines 18 - 27
}
```

Refresh now and... got it! Pretty cool, right?

Next: most of the time when you autowire an argument like `HttpClientInterface`, there's only *one* service in the container that implements that interface. But what if there were *multiple* HTTP clients in our container? How could we choose the one we want? It's time to talk about *named autowiring*.

Chapter 17: Named Autowiring & Scoped HTTP Clients

In `MixRepository`, it would be cool if we didn't need to specify the host name when we make the HTTP request. Like, it'd be great if that were preconfigured and we only needed to include the path. *Also*, pretty soon, we're going to configure an access token that will be used when we make requests to the GitHub API. We could pass that access token manually here in our service, but how cool would it be if the HttpClient service came preconfigured to always include the access token?

So, *does* Symfony have a way for us to, sort of, "preconfigure" the HttpClient service? It *does*! It's called "scoped clients": a feature of HttpClient where you can create multiple HttpClient services, each preconfigured *differently*.

Creating a Scoped Client

Here's how it works. Open up `config/packages/framework.yaml`. To create a scoped client, under the `framework` key, add `http_client` followed by `scoped_clients`. Now, give your scoped client a name, like `githubContentClient`... since we're using a part of their API that returns the content of files. Also add `base_uri`, go copy the host name over here... and paste:

config/packages/framework.yaml

```yaml
// ... line 1
framework:
// ... lines 3 - 19
    http_client:
        scoped_clients:
            githubContentClient:
                base_uri: https://raw.githubusercontent.com
// ... lines 24 - 30
```

Remember: the purpose of these config files is to *change* the services in the container. The end result of this new code is that a *second* HttpClient service will be added to the container. We'll see that in a minute. And, by the way, there's no way that you could just *guess* that you need `http_client` and `scoped_clients` keys to make this work. Configuration is the kind of thing where you really need to rely on the documentation.

Anyways, now that we've preconfigured this client, we should be able to go into `MixRepository` and make a request directly to the path:

```
src/Service/MixRepository.php
// ... lines 1 - 9
class MixRepository
{
// ... lines 12 - 18
    public function findAll(): array
    {
        return $this->cache->get('mixes_data', function(CacheItemInterface $cacheItem) {
// ... line 22
            $response = $this->httpClient->request('GET', '/SymfonyCasts/vinyl-mixes/main/mixes.json');
// ... lines 24 - 25
        });
    }
}
```

But if we head over and refresh... ah...

> *“Invalid URL: scheme is missing [...]. Did you forget to add "http(s)://"?”*

I didn't *think* we forgot... since we configured it via the `base_uri` option... but apparently that didn't work. And you may have guessed why. Find your terminal and run:

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

There are now *two* HttpClient services in the container: The normal, non-configured one and the one that *we* just configured. Apparently, in `MixRepository`, Symfony is still passing us the *unconfigured* HttpClient service.

How can I be sure? Well, think back to how autowiring works. Symfony looks at the type-hint of our argument, which is `Symfony\Contracts\HttpClient\HttpClientInterface`, and then looks in the container to find a service whose ID is an exact match. It's *that* simple

Fetching the Named Version of a Service

So... if there are multiple services with the same "type" in our container, is only the *main* one autowireable? Fortunately, no! We can use something called "named autowiring"... and it's already showing us how. If we type-hint an argument with `HttpClientInterface` *and name* the argument `$githubContentClient`, Symfony will pass us the second one.

Let's try it: change the argument from `$httpClient` to `$githubContentClient`:

```php
// src/Service/MixRepository.php
// ... lines 1 - 9
class MixRepository
{
    public function __construct(
        private HttpClientInterface $githubContentClient,
// ... lines 14 - 16
    ) {}
// ... lines 18 - 27
}
```

and now... it doesn't work. Whoops...

> *"Undefined property: `MixRepository::$httpClient`"*

That's... just me being careless. When I changed the argument name, it changed the property name. So... we need to adjust the code below:

```
src/Service/MixRepository.php
// ... lines 1 - 9
class MixRepository
{
    public function __construct(
        private HttpClientInterface $githubContentClient,
// ... lines 14 - 16
    ) {}
// ... line 18
    public function findAll(): array
    {
        return $this->cache->get('mixes_data', function(CacheItemInterface
$cacheItem) {
// ... line 22
            $response = $this->githubContentClient->request('GET',
'/SymfonyCasts/vinyl-mixes/main/mixes.json');
// ... lines 24 - 25
        });
    }
}
```

And now... it's alive! We just autowired a *specific* HttpClientInterface service!

Next, let's tackle another tricky problem with autowiring by learning how to fetch one of the *many* services in our container that is totally *not* available for autowiring.

Chapter 18: Non-Autowireable Services

Run:

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

And... I'll make this a bit smaller so that everything shows up on one line. As we know, this command shows *all* of the services in our container... but only a *small* number of these are autowireable. We know that because a service is autowireable *only* if its ID, which is this over here, is a class or interface name.

So at first, it might look like the Twig service is *not* autowireable. After all, its ID - `twig` - is definitely *not* a class or interface. But if you scroll up to the top... let's see... yep! There's another service in the container whose ID is `Twig\Environment`, which is an *alias* to the service `twig`. This is a little trick Symfony does to make services autowireable. If we type-hint an argument with `Twig\Environment`, we get the `twig` service.

However, most of services in this list do *not* have an alias like that. So they are *not* autowireable. And, that's *usually* fine. If a service *isn't* autowireable, it's probably because you'll never need to use it. *But* let's pretend that we *do* want to use one of these.

Check this one out! It's called `twig.command.debug`. Open up another tab. Earlier, we ran:

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

This shows us *all* of the functions and filters from Twig... which is nice! Well, surprise! This command *comes* from the `twig.command.debug` service! Because, "everything in Symfony is done by a service" - even console commands.

As a challenge, let's see if we can inject this service into `MixRepository`, execute it, and dump its output.

Dependency Injection: Adding the new Argument

First things first. In `MixRepository`, we just discovered that, in order to do our work, we need access to another service. What do we do? The answer: *Dependency injection*, which is that fancy word for adding another construct argument and setting it onto a property, which we can do all at once with `private $twigDebugCommand`:

```
src/Service/MixRepository.php
// ... lines 1 - 9
class MixRepository
{
    public function __construct(
// ... lines 13 - 15
        private bool $isDebug,
        private $twigDebugCommand,
    ) {}
// ... lines 19 - 28
}
```

If we stopped right now and refreshed... no surprise! We get an error. Symfony has no idea what to pass for that argument.

What if we added the *type* for this class? Back over in our terminal, we can see that this service is an instance of `DebugCommand`. Over here, let's add that type-hint: `DebugCommand`... we want the one from `Symfony\Bridge\Twig\Command`. Hit "tab" to autocomplete that:

```
src/Service/MixRepository.php
// ... lines 1 - 5
use Symfony\Bridge\Twig\Command\DebugCommand;
// ... lines 7 - 11
class MixRepository
{
    public function __construct(
// ... lines 15 - 18
        private DebugCommand $twigDebugCommand,
    ) {}
// ... lines 21 - 30
}
```

And then... refresh. *Still* an error! Okay, we *should* add the type-hint because we're good programmers. But... no matter how hard we try, this is *not* an autowireable service. So, how do we fix this?

Binding the Argument in YAML

There are two main ways. I'll show you the *old* way first, which I'm *mostly* doing because you'll see it in documentation and blog posts all over the place. In `config/services.yaml`, just like we did earlier for the `$isDebug` argument, override our service entirely. Say `App\Service\MixRepository`, and add a `bind` key. Then, we're going to hint what to pass to the `$twigDebugCommand` argument.

The *only* tricky thing is figuring out what *value* to set. For example, if I go and copy the service ID - `twig.command.debug` - and paste that here... that's *not* going to work! That's literally going to pass that *string*. If you refresh, yup!

> *"Argument 4 must be of type `DebugCommand`, string given."*

We need to tell Symfony to pass the *service* that has this ID. In these YAML files, there's a special syntax to do just that: *prefix* the service ID with the `@` symbol:

config/services.yaml

```
// ... lines 1 - 12
services:
// ... lines 14 - 32
    App\Service\MixRepository:
        bind:
            $twigDebugCommand: '@twig.command.debug'
```

As *soon* as we do that... the fact that this doesn't explode means it's working!

The Autowire Attribute

But... let's remove this. Because I want to show you the *new* way do this... which leverages that same fancy `Autowire` attribute.

Up here, say `#[Autowire()]`, but instead of just passing a string, say `service: 'twig.command.debug'`:

src/Service/MixRepository.php

```
// ... lines 1 - 11
class MixRepository
{
    public function __construct(
// ... lines 15 - 18
        #[Autowire(service: 'twig.command.debug')]
        private DebugCommand $twigDebugCommand,
    ) {}
// ... lines 22 - 31
}
```

Using the new Argument

I love that! Before we try this, let's actually *use* the service. Head down to `findAll()`. Executing a console command *manually* in your PHP code is totally possible. It's a little weird, but cool! We need to create an `$output = new BufferedOutput()` object... then we can execute the command by saying `$this->twigDebugCommand->run(new ArrayInput())` - this is, sort of, *faking* the command-line arguments - pass that an empty `[]` - then `$output`. Whatever the command *outputs* will be set onto that object.

To see if it's working, just `dd($output)`:

src/Service/MixRepository.php

```
// ... lines 1 - 6
use Symfony\Component\Console\Input\ArrayInput;
use Symfony\Component\Console\Output\BufferedOutput;
// ... lines 9 - 13
class MixRepository
{
// ... lines 16 - 24
    public function findAll(): array
    {
        $output = new BufferedOutput();
        $this->twigDebugCommand->run(new ArrayInput([]), $output);
        dd($output);
// ... lines 30 - 35
    }
}
```

Testing time! Refresh... and got it! How fun is that?

All right, now that this is working, let's comment out this silliness. I'll keep our `$twigDebugCommand` injected just for reference.

The *key* takeaway is this: *most* arguments to services will be autowireable. Yay! But when you hit an argument that is *not* autowireable, you can use the `Autowire` attribute to point to the value or service you need.

Next: Remember when I told you that `MixRepository` was the first service we ever created? Well... I lied. It turns out that our *controllers* have been services this whole time!

Chapter 19: Controllers are Services Too!

Open up `src/Controller/VinylController.php`. It *may* or may not be obvious, but our controller classes are *also* services in the container! Yep! They *feel* special because they're *controllers*... but they're really just good old, boring services like everything else. Well, except that they have one *superpower* that *nothing* else has: the ability to autowire arguments into their action *methods*. Normally, autowiring only works with the constructor.

Binding Action Arguments

And, the action methods really *do* work *just* like the constructors when it comes to autowiring. For example, add a `bool $isDebug` argument to the `browse()` action... then `dump($isDebug)` below:

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
// ... lines 13 - 31
    public function browse(MixRepository $mixRepository, bool $isDebug,
string $slug = null): Response
    {
        dump($isDebug);
// ... lines 35 - 42
    }
}
```

And that... doesn't work! So far, the only two things that we know we are allowed to have as arguments to our "actions" are (A), any wildcards in the route like `$slug` and (B) autowireable services, like `MixRepository`.

But now, go back to `config/services.yaml` and *uncomment* that global `bind` from earlier:

```
config/services.yaml
// ... lines 1 - 12
services:
    # default configuration for services in *this* file
    _defaults:
// ... lines 16 - 17
        bind:
            'bool $isDebug': '%kernel.debug%'
// ... lines 20 - 32
```

This time... it works!

Adding a Constructor

Going in the *other* direction, because controllers are services, you can *absolutely* have a constructor if you want. Let's move `MixRepository` and `$isDebug` up to a new constructor. Copy those, remove them... add `public function __construct()`, paste... then I'll put them on their own lines. To turn them into properties, add `private` in front of each:

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
    public function __construct(
        private bool $isDebug,
        private MixRepository $mixRepository
    )
    {}
// ... lines 18 - 36
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
// ... lines 40 - 48
    }
}
```

Back down below, we just need to make sure we change to `dump($this->isDebug)` and add `$this->` in front of `mixRepository`:

```
src/Controller/VinylController.php
// ... lines 1 - 10
class VinylController extends AbstractController
{
    public function __construct(
        private bool $isDebug,
        private MixRepository $mixRepository
    )
    {}
// ... lines 18 - 36
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
        dump($this->isDebug);
// ... lines 41 - 42
        $mixes = $this->mixRepository->findAll();
// ... lines 44 - 48
    }
}
```

Nice! If we try this now... that works just fine!

I don't *normally* follow this approach... mainly because adding arguments to the action *method* is just so darn easy. But if you need a service or other value in *every* action method of your class, you can definitely clean up your argument list by injecting it through the constructor. I'll go remove that `dump()`.

Next, let's talk about environment variables and the purpose of the `.env` file that we looked at earlier. This stuff will become increasingly important as we make our app more and more realistic.

Chapter 20: Environment Variables

Open `config/packages/framework.yaml`. We don't need to be authenticated to use this raw user content part of GitHub's API:

```yaml
config/packages/framework.yaml
// ... line 1
framework:
// ... lines 3 - 19
    http_client:
        scoped_clients:
            githubContentClient:
                base_uri: https://raw.githubusercontent.com
// ... lines 24 - 30
```

But if we hit this endpoint a lot, we *might* hit their rate-limiting, which is pretty low for anonymous users. So let's *authenticate* our request.

Adding an Authorization Header to the HTTP Request

First, if you're coding along with me, head to "github.com" and create your own personal access token. Once you've done that, open up `MixRepository` and go down to where we make the HTTP request. To attach the access token to the request pass a *third* argument, which is an array. Inside, add a `headers` key set to another array, with an `Authorization` header assigned to the word `Token` and then the access token. Start by using a fake token:

```php
// src/Service/MixRepository.php
// ... lines 1 - 13
class MixRepository
{
// ... lines 16 - 24
    public function findAll(): array
    {
// ... lines 27 - 32
        return $this->cache->get('mixes_data', function(CacheItemInterface $cacheItem) {
// ... line 34
            $response = $this->githubContentClient->request('GET', '/SymfonyCasts/vinyl-mixes/main/mixes.json', [
                'headers' => [
                    'Authorization' => 'Token ghp_foo_bar',
                ]
            ]);
// ... lines 40 - 41
        });
    }
}
```

You can tell this is working because, when we go back over to the page and refresh... it explodes! Our API call now *fails* with a 404 because it recognizes that we're *trying* to authenticate with a token... but the one we passed is *bogus*.

Now add your *real* token. Try it again and... it works!

Moving Authorization Header to framework.yaml

So this is cool! *But* it would be nicer if the service came preconfigured to *automatically* set this authorization header... especially if we want to use this HTTP Client service in multiple places. Can we do that? You bet!

Copy the `Token` line, head into `framework.yaml`, and after `base_uri`, pass a `headers` key with `Authorization` set to our long string. Actually, let me put a *fake* token in there temporarily:

```
config/packages/framework.yaml
// ... line 1
framework:
// ... lines 3 - 19
    http_client:
        scoped_clients:
            githubContentClient:
                base_uri: https://raw.githubusercontent.com
                headers:
                    Authorization: 'Token ghp_FAKE'
// ... lines 26 - 32
```

Back in `MixRepository`, remove that third argument:

```
src/Service/MixRepository.php
// ... lines 1 - 13
class MixRepository
{
// ... lines 16 - 24
    public function findAll(): array
    {
// ... lines 27 - 32
        return $this->cache->get('mixes_data', function(CacheItemInterface $cacheItem) {
// ... line 34
            $response = $this->githubContentClient->request('GET', '/SymfonyCasts/vinyl-mixes/main/mixes.json');
// ... lines 36 - 37
        });
    }
}
```

And now, when we try this... great! Things are broken, which proves we're sending that header... just with the wrong value. If we change to our *real* token... once again... it works! Awesome!

Hello Environment Variables

So far, this is just a nice feature of the HttpClient. But this also helps highlight a common problem. It's... not super great to have our sensitive GitHub API token hardcoded in this file. I mean, this file is going to be committed to our repository. I love my teammates... but I don't love them *so* much that I want to share my access token to with them... or the access token for our company.

This is where *environment variables* come in handy. If you're not familiar with environment variables, they're variables that you can set on any system (Windows, Linux, whatever.)... and then you can read them from inside of PHP. Many hosting platforms make it super easy to set these. How does that help us? Because, in theory, we could set our access token as an *environment* variable then simply *read* it in PHP. That would let us *avoid* putting that sensitive value *inside* our code.

Reading Environment Variables

But, before we talk about *setting* environment variables, how do we *read* environment variables in Symfony? Copy your access token so you don't lose it, put single quotes around `Token`, and then we're going to use a very special syntax to *read* an environment variable. It's actually going to look like a parameter. Start and end with `%`, and inside, say `env()` with the name of the environment variable. How about `GITHUB_TOKEN`. I just made that name up:

```yaml
config/packages/framework.yaml
// ... line 1
framework:
// ... lines 3 - 19
    http_client:
        scoped_clients:
            githubContentClient:
                base_uri: https://raw.githubusercontent.com
                headers:
                    Authorization: 'Token %env(GITHUB_TOKEN)%'
// ... lines 26 - 32
```

If we head back and refresh... we are now *reading* that `GITHUB_TOKEN` environment variable... but we haven't *set* it yet, so we get this "Environment variable not found" error.

Setting Environment Variables & .env

In the real world, setting environment variables is... actually kind of tricky. It's different on Windows versus Linux. And while many *hosting* platforms *do* make it super easy to set environment variables, it's not very simple to do *locally* on your computer.

That is why this `.env` file exists. Very simply, when Symfony boots up, it reads the `.env` file and turns all of these into environment variables. This means we can say `GITHUB_TOKEN=` and paste our token... and now... it works!

```
.env
// ... lines 1 - 18
###

GITHUB_TOKEN=
```

By the way, if there were a *real* `GITHUB_TOKEN` environment variable set on my system that real environment variable would win over what we have in this file.

The .env.local File

Okay... this is cool... but we *still* have the same problem! We have a sensitive value that's inside of a file... which *is* committed to our repository.

Ok, then, let's try something else. Copy the GitHub token, delete the value from this file, and then create a new file called `.env.local`. Set the environment variable *here*.

And now... things *still* work!

Here's the deal. When Symfony boots up, it first reads the `.env` file and turns all of these into environment variables. *Then* it reads `.env.local` and turns anything in *here* into environment variables... which *override* any values set in `.env`.

The result is that your `.env` file is meant to hold safe, default values that are ok to be committed to your repository. Then locally, (and maybe also on production, depending on how you deploy), you create a `.env.local` file and put the sensitive values there. The *key* thing is that `.env.local` is *ignored* by Git. You can see it's already in our `.gitignore` file. So while this file *will* contain sensitive values, it will *not* be committed to the repository.

There *are* a few other `.env` files that you can create... and you can see them mentioned here. They're not as important, but if you want to read about them, you can check out the documentation.

Visualizing Env Vars with debug:dotenv

Another cool thing about environment variables is that you can visualize them by running:

```
php bin/console debug:dotenv
```

Sweet! You can see the current value of `GITHUB_TOKEN`... and that this value is also set in `.env.local`. In contrast, `APP_ENV` and `APP_SECRET` have `n/a` here, meaning their values are *not* being overridden in `.env.local`. It also tells us which `.env` files it detected.

Env Var Processors

There are a few tricks you can use with environment variables. For example, there's something called a "processor system" where you could use `trim` to "trim" the white space on `GITHUB_TOKEN`. *Or* you could use `file` where the `GITHUB_TOKEN` variable is actually a path to a file that contains the true value. Anyways, these are called "env var processors" if you want to read more about them.

Next, let's talk quickly about deployment... but even more about how we can safely store these sensitive values when you deploy to production. One option is Symfony's secrets vault.

Chapter 21: The Secrets Vault

I don't want to get *too* far into deployment, but let's do a quick "How To Deploy Your Symfony App 101" course. Here's the idea.

Deployment 101

Step 1: You need to *somehow* get all of your committed code onto your production machine and then run

```
composer install
```

to populate the `vendor/` directory.

Step 2: Somehow create a `.env.local` file with all of your production environment variables, which will include `APP_ENV=prod`, so that you're in the prod environment.

And Step 3: run

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

which will clear the cache in the production environment, and then

```
php bin/console cache:warmup
```

to "warm up" the cache. There may be a few other commands, like running your database migrations... but this is the general idea. And the Symfony docs have more details.

By the way, in case you're wondering, we deploy via https://platform.sh, using Symfony's Cloud integration... which handles *a lot* of stuff for us. You can check it out by going to https://symfony.com/cloud. It also helps support the Symfony project, so it's a win-win.

Use Real Environment Variables When Possible

Anyway, the trickiest part of the process is Step 2 - creating the `.env.local` file with all of your production values, which will include things like API keys, your database connection details and more.

Now, *if* your hosting platform allows you to store *real* environment variables directly inside of it, problem solved! If you set *real* env vars, then there is no need to manage a `.env.local` file at all. As soon as you deploy, Symfony will instantly see and use the real env vars. That's what we do for Symfonycasts.

Creating .env.local During Deploy?

But if that's *not* an option for you, you'll need to somehow give your deployment system access to your sensitive values so that it can create the `.env.local` file. But... since we're not committing any of these values to our repository, where *should* we store them?

One option for handling sensitive values is Symfony's *secrets vault*. It's a set of files that contain environment variables in an *encrypted form*. These files are *safe* to commit to your repository... because they're encrypted!

Creating the dev Vault

If you want to store secrets in a vault, you'll need two of them: one for the `dev` environment and one for the `prod` environment. We're going to *create* these two vaults first... *then* I'll explain how to read values out of them.

Start by creating one for the `dev` environment. Run:

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

Pass this `GITHUB_TOKEN`, which is the secret we want to set. It then asks for our "secret value". Since this is the vault for the `dev` environment, we want to put something that's safe for everyone to see. I'll explain why in a moment. I'll say `CHANGEME`. You can't see me type that... only because Symfony hides it for security reasons.

Since this is the *first* secret we've created, Symfony automatically created the secrets vault behind the scenes... which is literally a set of files that live in `config/secrets/dev/`. For the *dev* vault, we're going to commit *all* of these files to the repository. Let's do that. Add the entire secrets directory:

```
git add config/secrets/dev
```

Then commit with:

```
git commit -m "adding dev secrets vault"
```

The Secrets Vault Files

Here's a quick explanation of the files. `dev.list.php` stores a list of *which* values live inside the vault, `dev.GITHUB_TOKEN.28bd2f.php` stores the actual encrypted value, and `dev.encrypt.public.php` is the cryptographic key that allows developers on your team to add *more* secrets. So if another developer pulled down the project, they'll have this file... so they can add more secrets. Finally, `dev.decrypt.private.php` is the secret key that allows us to *decrypt* and *read* the values in the vault.

As *soon* as the vault files are present, Symfony will automatically open them, decrypt the secrets, and expose them as environment variables! But, more on that in a few minutes.

Storing the dev Decrypt Key?

But wait: did we really just *commit* the `decrypt` key to the repository? Yes! That would *normally* be a no-no! Why would you go to the trouble of encrypting values... just to store the decryption key right next to them?

The reason we're doing *exactly* that is that this is our *dev* vault, which means we're only going to store values that are safe for *all* developers to look at. The `dev` vault will only be used local development... and we want our teammates to be able to pull down the code and read those without any trouble.

Ok, at this point we have a `dev` vault that Symfony will automatically use in the `dev` environment. Next: let's create the *prod* vault, which will hold the *truly* secret values. We'll then learn relationship between vault secrets and environment variables... as well as an easy way to visualize all of this.

Chapter 22: Reading Secrets vs Env Vars

We *just* created a secrets vault for our `dev` environment... which will contain a default "safe" version of any sensitive environment variables. For example, we set the `GITHUB_TOKEN` value to `CHANGEME`.

Now let's create the `prod` environment vault. Do that by saying:

```
./bin/console secrets:set GITHUB_TOKEN --env=prod
```

This time, grab the *real* secret value from `.env.local` and paste it here. Just like before, since there wasn't a `prod` vault already, Symfony *created* it. And it's got the *same* four files as before. Though, there *is* one subtle, but important difference.

Add that new directory to git:

```
git add config/secrets/prod
```

Then run:

```
git status
```

Woh! Only *three* of the four files were added. The *fourth* file - the `decrypt` key - is *ignored* by Git. We already have a line inside in `.gitignore` for that. We do *not* want to commit the `prod` decrypt key to the repository... because anyone that has this key will be able to read *all* of our secrets.

So, if another developer pulls down the project now, they *will* have the `dev` decrypt key, so they'll have no problems reading values from the `dev` vault. They won't have the `prod` decrypt key... but no big deal! The only place where you need the `prod` decrypt key is on production!

So with this setup, when you deploy, instead of needing to create an entire `.env.local` file containing *all* of your secrets, you just need to worry about getting this *one* `prod.decrypt.private.php` file up into your code. Or, alternatively, you can read this key and set it on an environment variable: you can check the docs for details on how.

Using The Secrets Vault

But... wait a second. I haven't really explained *how* the vault is used! We know that the `dev` environment will use the `dev` vault... and `prod` will use `prod`... but how do we *read* secrets out of the vault?

The answer is... we already *are*! Secrets *become* environment variables. It's as simple as that! So in `config/packages/framework.yaml`, by using this `env` syntax, this `GITHUB_TOKEN` could be a *real* environment variable, or it could be a *secret* in our vault.

To see if this is working, head to `MixRepository` and `dd($this->githubContentClient)`:

src/Service/MixRepository.php

```php
// ... lines 1 - 13
class MixRepository
{
// ... lines 16 - 24
    public function findAll(): array
    {
// ... lines 27 - 31
        dd($this->githubContentClient);
// ... lines 33 - 39
    }
}
```

Move over, refresh, and... let's see if we can find the Authorization header in this. Actually, there's a really cool trick with dump. Click on this area and hold "command" or "control" + "F" to search inside of it. Search for the word "token" and... oh, that's not right! That's our *real* token. But... since we're in the *dev* environment, shouldn't it be reading our *dev* vault where we set the fake `CHANGEME` value? What's going on?

Secrets Must Fully Be Converted Away from Env Vars

As I mentioned, secrets become environment variables. *But* environment variables take *precedence* over secrets: even environment variables defined in the `.env` files. Yup, because we have a `GITHUB_TOKEN` env var set in `.env` and `.env.local`, *that* is taking precedence over the value in the vault!

Here's the point. As soon as you choose to convert a value from an environment variable into a secret, you need to *stop* setting it as an environment variable *completely*. In other words, delete `GITHUB_TOKEN` in `.env` and *also* in `.env.local`.

Go refresh, click on this again, use "command" + "F", search for "token", and... got it! We see "CHANGEME"! If we were in the `prod` environment, it would read the value from the prod vault... assuming the prod decrypt key was available.

The secrets:list Command

Ok, remove that `dd()` and refresh to discover that... locally, everything is broken! Dang! But... of course! It's now using that *fake* token from the *dev* vault. It *would* work ok on production... but how can I fix my local setup so I can keep working?

We *could* temporarily override the `GITHUB_TOKEN` secret value in the `dev` vault by running the `secrets:set` command. But... that's lame! We would need to be extra careful to not commit the modified, encrypted file.

Before we fix this, I want to show you a really handy command for the vault:

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

Yup, this shows you all of the secrets in our vault. Pretty cool! And you can even pass `--reveal` to *reveal* the value... as long as you have the `decrypt` key.

You may have noticed that it *gives* us the value right here... but then says "Local Value" with a blank space. Hmm...

Re-run the command, but this time add `--env=prod`.

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

And... same thing! This shows us the *real* `prod` value... but there's still this "Local Value" spot with nothing.

This "Local Value" is the key to fixing our broken dev setup: it's a way to override a secret, but only *locally* on our one machine.

How do you *set* this local override value? Copy the real `GITHUB_TOKEN` value, then move over, find `.env.local` - the same file we've been working in - and say `GITHUB_TOKEN=` and paste the value we just copied.

Yup! Locally, we're going to take advantage of the fact that environment variables "win" over secrets! Back at your terminal, run

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

again. Yes! The *official* value in the vault is "CHANGEME"... but the *local* value is our *real* token which, as we know, will *override* the secret and be used. If we try the page again... it works!

Okay, team! We're... well... *basically* done! So as a reward for your hard work on these *super* important topics, let's celebrate by using Symfony's code generator library: MakerBundle.

Chapter 23: MakerBundle & Autoconfiguration

Congrats, team! We are *done* with the heavy stuff in this tutorial! So it's time for a victory lap. Let's install one of my *favorite* Symfony bundles: MakerBundle. Find your terminal and run:

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

In this case, I'm using the `--dev` flag because this is a code generation utility that we only need locally, *not* on production.

This bundle, of course, provides services. But these services aren't really meant for us to use *directly*. Instead, all of the services from this bundle power a bunch of new `bin/console` commands. Run

```
php bin/console
```

and look for the `make` section. Ooh. There's a ton of stuff here for setting up security, generating doctrine entities for the database (which we'll do in the next tutorial), making a CRUD, and much more.

Generating a new Command Class

Let's try one: how about we try to build our *own* new custom console command that will appear in this list. To do that, run:

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

This will interactively ask you for the name of the command. Let's say `app:talk-to-me`. You don't *have* to, but it's pretty common to prefix your custom commands with `app:`. And... done!

That created exactly *one* new file: `src/Command/TalkToMeCommand.php`. Let's go open that up:

src/Command/TalkToMeCommand.php

```php
<?php

namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

#[AsCommand(
    name: 'app:talk-to-me',
    description: 'Add a short description for your command',
)]
class TalkToMeCommand extends Command
{
    protected function configure(): void
    {
        $this
            ->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
            ->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $arg1 = $input->getArgument('arg1');

        if ($arg1) {
            $io->note(sprintf('You passed an argument: %s', $arg1));
        }

        if ($input->getOption('option1')) {
            // ...
        }

        $io->success('You have a new command! Now make it your own! Pass --help to see your options.');

        return Command::SUCCESS;
    }
}
```

Cool! On top, you can see that the name and description of the command are done in a PHP attribute! Then, down in this `configure()` method, which we'll talk about more in a minute, we can configure arguments and options that can be passed from the command line.

When we *run* the command, `execute()` will be called... where we can print things out to the screen or read options and arguments.

Perhaps the *best* thing about this class is that... it *already* works. Check it out! Back at your terminal, run;

```
php bin/console app:talk-to-me
```

And... it's *alive*! It doesn't *do* much, but this output is coming from down here. Woo!

Autoconfiguration: Auto Discovering "Plugins"

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

The way this works is *much* cooler. Open up `config/services.yaml` and look at the `_defaults` section:

config/services.yaml

```yaml
// ... lines 1 - 12
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
// ... lines 18 - 32
```

We talked about what `autowire: true` means, but I didn't explain the purpose of `autoconfigure: true`. Because this is below `_defaults`, autoconfiguration *is* active on all of our services, including our new `TalkToMeCommand` service. When `autoconfiguration` is enabled, it basically tells Symfony:

> *"Hey, please look at the base class or interface of each service, and if it looks like a class should be a console command... or an event subscriber... or any other class that hooks into a part of Symfony, please automatically integrate the service into that system. Okay, thanks. Bye!"*

Yep! Symfony sees that our class extends `Command` and thinks:

> *"Hmm, I may not be a self-aware AI... but I bet this is a command. I better notify the console system about it!"*

I *love* autoconfiguration. It means that we can create a PHP class, extend whatever base class or implement whatever interface needed for the "thing" that we're building, and... it will just *work*.

Internally, if you want all the nerdy details, autoconfiguration adds a *tag* to your service, like `console.command`, which is what ultimately helps it get noticed by the console system.

All right, now that our command is working, let's have some fun and customize it *next*.

Chapter 24: Customizing a Command

We have a new console command! *But*... it doesn't do much yet, aside from printing out a message. Let's make it *fancier*.

Scroll to the top. This is where we have the name of our command, and there's also a description... which shows up next to the command. Let me change ours to

> *"A self-aware command that can do... only one thing."*

src/Command/TalkToMeCommand.php

```
// ... lines 1 - 12
#[AsCommand(
// ... line 14
    description: 'A self-aware command that can do... only one thing.',
)]
class TalkToMeCommand extends Command
{
// ... lines 19 - 43
}
```

Configuring Arguments and Options

Our command is called `app:talk-to-me` because, when we run this, I want to make it possible to pass a name to the command - like Ryan - and then it'll reply with "Hey Ryan!". So, literally, we'll type `bin/console app:talk-to-me ryan` and it'll reply back.

When you want to pass a value to a command, that's known as an *argument*... and those are configured down in... the `configure()` method. There's already an argument called `arg1`... so let's change that to `name`.

This key is completely *internal*: you'll never see the word `name` when you're *using* this command. But we *will* use this key to *read* the argument value in a minute. We can also give the argument a description and, if you want, you can make it *required*. I'll keep it as optional.

The next thing we have are *options*. These are like arguments... except that they start with a `--` when you use them. I want to have an optional flag where we can say `--yell` to make the command *yell* our name back.

In this case, the name of the option, `yell`, *is* important: we *will* use this name when passing the option at the command line to use it. The `InputOption::VALUE_NONE` means that our flag will just be `--yell` and not `--yell=` some value. If your option accepts a value, you would change this to `VALUE_REQUIRED`. Finally, give this a description.

```
src/Command/TalkToMeCommand.php
// ... lines 1 - 16
class TalkToMeCommand extends Command
{
    protected function configure(): void
    {
        $this
            ->addArgument('name', InputArgument::OPTIONAL, 'Your name')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'Shall I yell?')
        ;
    }
// ... lines 26 - 43
}
```

Beautiful! We're not *using* this argument and option yet... but we can already re-run our command with a `--help` option:

```
php bin/console app:talk-to-me --help
```

And... awesome! We see the description up here... along with some details about how to use the argument and the `--yell` option.

Filling in execute()

When we call our command, very simply, Symfony will call `execute()`... which is where the fun starts. Inside, we can do *whatever* we want. It passes us two arguments: `$input` and `$output`. If you want to read some input - like the `name` argument or the `yell` option, use `$input`. And if you want to *output* something, use `$output`.

But in Symfony, we normally pop these two things into *another* object called `SymfonyStyle`. This is helper class makes reading and outputing easier... and fancier.

Ok: let's start by saying `$name = $input->getArgument('name')`. If we don't have a name, I'll default this to `whoever you are`. Below, read the option: `$shouldYell = $input->getOption('yell')`:

src/Command/TalkToMeCommand.php

```php
// ... lines 1 - 16
class TalkToMeCommand extends Command
{
// ... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $name = $input->getArgument('name') ?: 'whoever you are';
        $shouldYell = $input->getOption('yell');
// ... lines 32 - 40
    }
}
```

Cool. Let's clear out this stuff down here and start our message: `$message = sprintf('Hey %s!', $name)`. Then if we want to yell, you know what to do: `$message = strtoupper($message)`. Below, use `$io->success()` and put the message there.

```
src/Command/TalkToMeCommand.php
// ... lines 1 - 16
class TalkToMeCommand extends Command
{
// ... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $name = $input->getArgument('name') ?: 'whoever you are';
        $shouldYell = $input->getOption('yell');

        $message = sprintf('Hey %s!', $name);
        if ($shouldYell) {
            $message = strtoupper($message);
        }

        $io->success($message);

        return Command::SUCCESS;
    }
}
```

This is one of the many helper methods on the `SymfonyStyle` class that help format your output. There's also `$io->warning()`, `$io->note()`, and several others.

Let's try it. Spin over and run:

```
php bin/console app:talk-to-me ryan
```

And... oh hello there! If we yell:

```
php bin/console app:talk-to-me ryan --yell
```

THAT WORKS TOO! We can even yell at 'whoever I am':

```
php bin/console app:talk-to-me --yell
```

Awesome! But let's get crazier... by autowiring a service and asking a question *interactively* on the command line. That's next... and it's the last chapter!

Chapter 25: Command: Autowiring & Interactive Questions

Last chapter team! Let's do this!

Ok, what if we need a *service* from inside our command? For example, let's say that we want to use `MixRepository` to print out a vinyl mix recommendation. How can we do that?

Well, we're inside of a service and we need access to *another* service, which means we need... the dreaded *dependency injection*. Kidding - not dreaded, easy with autowiring!

Add `public function __construct()` with `private MixRepository $mixRepository` to create and set that property all at once.

```
src/Command/TalkToMeCommand.php
// ... lines 1 - 4
use App\Service\MixRepository;
// ... lines 6 - 17
class TalkToMeCommand extends Command
{
    public function __construct(
        private MixRepository $mixRepository
    )
    {
// ... line 24
    }
// ... lines 26 - 55
}
```

Though, if you hover over `__construct()`, it says:

> *“Missing parent constructor call.”*

To fix this, call `parent::__construct()`:

```
src/Command/TalkToMeCommand.php
// ... lines 1 - 4
use App\Service\MixRepository;
// ... lines 6 - 17
class TalkToMeCommand extends Command
{
    public function __construct(
        private MixRepository $mixRepository
    )
    {
        parent::__construct();
    }
// ... lines 26 - 55
}
```

This is a *super* rare situation where the base class has a constructor that we need to call. In fact, this is the *only* situation I can think of in Symfony like this... so not *normally* something you need to worry about.

Interactive Questions

Down here, let's output a mix recommendation... but make it even *cooler* by first asking the user *if* they want this recommendation.

We can ask interactive questions by leveraging the `$io` object. I'll say `if ($io->confirm('Do you want a mix recommendation?'))`:

```
src/Command/TalkToMeCommand.php
// ... lines 1 - 17
class TalkToMeCommand extends Command
{
// ... lines 20 - 34
    protected function execute(InputInterface $input, OutputInterface
$output): int
    {
// ... lines 37 - 45
        $io->success($message);

        if ($io->confirm('Do you want a mix recommendation?')) {
// ... lines 49 - 51
        }
// ... lines 53 - 54
    }
}
```

This will ask that question, and if the user answers "yes", return true. The `$io` object is *full* of cool stuff like this, including asking multiple choice questions, and auto-completing answers. Heck, we can even build a progress bar!

Inside the if, get all of the mixes with `$mixes = $this->mixRepository->findAll()`. Then... we need just a bit of ugly code - `$mix = $mixes[array_rand($mixes)]` - to get a random mix.

Print the mix with one more `$io` method `$io->note()` passing `I recommend the mix` and then pop in `$mix['title']`:

```php
// src/Command/TalkToMeCommand.php
// ... lines 1 - 17
class TalkToMeCommand extends Command
{
// ... lines 20 - 34
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
// ... lines 37 - 45
        $io->success($message);

        if ($io->confirm('Do you want a mix recommendation?')) {
            $mixes = $this->mixRepository->findAll();
            $mix = $mixes[array_rand($mixes)];
            $io->note('I recommend the mix: ' . $mix['title']);
        }
// ... lines 53 - 54
    }
}
```

And... done! By the way, notice this `return Command::SUCCESS`? That controls the exit code of your command, so you'll always want to have `Command::SUCCESS` at the bottom of your command. If there was an error, you could `return Command::ERROR`.

Tip

Whoops, the correct constant name if the command fails is `Command::FAILURE`!

Okay, let's try this! Head over to your terminal and run:

```
php bin/console app:talk-to-me --yell
```

We get the output... and then we get:

> *“Do you want a mix recommendation?”*

Why, yes we *do*! And what an *excellent* recommendation!

All right, team! We did it! We finished - what I think is - the most important Symfony tutorial of all time! No matter what you need to build in Symfony, the concepts we've just learned will be the *foundation* of doing it.

For example, if you need to add a custom function or filter to Twig, no problem! You do this by creating a Twig *extension* class... and you can use MakerBundle to generate this for you or build it by hand. It's very similar to creating a custom console command: in both cases, you're building something to "hook into" part of Symfony.

So, to create a Twig *extension*, you would create a new PHP class, make it implement whatever interface or base class that Twig extensions need (the documentation will tell you that)... and then you just fill in the logic... which I won't show here.

That's it! Behind the scenes, your Twig extension would *automatically* be seen as a service, and autoconfiguration would make sure it's integrated into Twig... *exactly* like the console command.

In the next course, we'll put our new superpowers to work by adding a database to our app so that we can load real, dynamic data. And if you have any *real*, *dynamic* questions, we are here for you, as always, down in the comment section.

All right, friends. Thanks so much for coding with me and we'll see you next time.