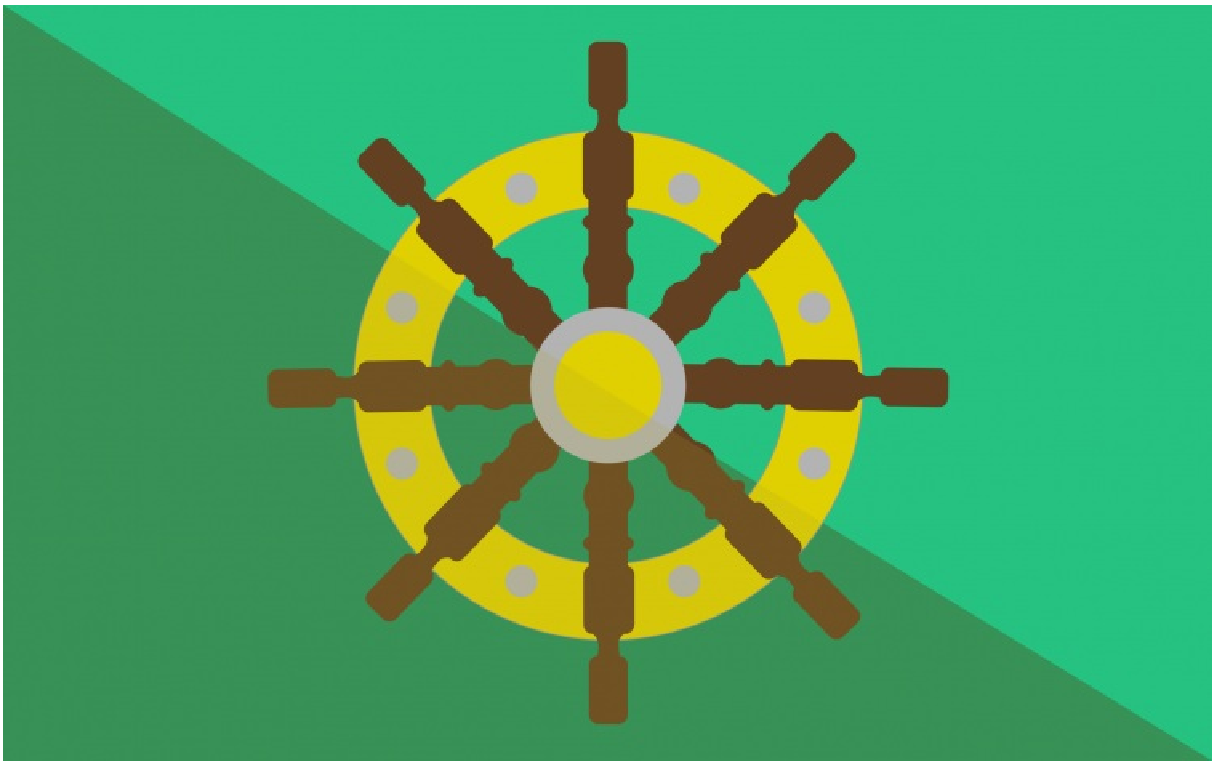
Symfony 3 Fundamentals: Bundles, Configuration & Environments



With <3 from SymfonyCasts

Chapter 1: Bundles

Woh! You're back! Hey friend! Ok, I'm glad you're here: this is a *big* episode for us. We're about to learn some of the most critical concepts that will *really* help you to master Symfony... and of course, impress all your friends.

Like always, you should totally code along with me. Download the code from the course page and move into the start directory. I already have that ready, so let's start the built-in web server. Open a new terminal tab and run the ./bin/console server:run command:

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

Ding! Now dust off your browser and try to load http://localhost:8000/genus/octopus, which is the page we made in the last tutorial. Awesome!

The 2 Parts of Symfony

After episode 1, we already know a lot. We know that Symfony is pretty simple: create a **route**, create a **controller** function, make sure that function returns a Response object and then go eat a sandwich. So, Route -> Controller -> Response -> Sandwich. And suddenly, you know half of Symfony... *and* you're not hungry anymore.

The second half of Symfony is a all about the huge number of optional useful objects that can help you get your work done. For example, there's a *logger* object, a *mailer* object, and a *templating* object that... uh... renders templates. In fact, the $this->render() shortcut we've been using in the controller is just a shortcut to go out to the templating object and call a method on it:

```
398 lines | vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
... lines 1 - 11
namespace Symfony\Bundle\FrameworkBundle\Controller;
... lines 13 - 38
abstract class Controller implements ContainerAwareInterface
{
... lines 41 - 185
    protected function render($view, array $parameters = array(), Response $response = null)
    {
        if ($this->container->has('templating')) {
            return $this->container->get('templating')->renderResponse($view, $parameters, $response);
        }
... lines 191 - 202
    }
... lines 204 - 396
}
```

All of these useful objects - or services - are put into one big *beautiful* object called the container. If I give you the container, then you're *incredibly* dangerous: you can fetch any object you want and do anything.

How do we know what handy services are inside of the container? Just use the debug:container command:

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

You can even search for services - like log:

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

But Where do Services Come From?

But where do these come from? What magical, mystical creatures are providing us with all of these free tools? The answer is:

the bundle fairies.

In your IDE, open up app/AppKernel.php:

54 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new Symfony\Bundle\MonologBundle\MonologBundle(),
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
            new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
            new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
... lines 18 - 20
            new AppBundle\AppBundle(),
        );

        if (in_array($this->getEnvironment(), array('dev', 'test'), true)) {
            $bundles[] = new Symfony\Bundle\DebugBundle\DebugBundle();
            $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
            $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
            $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
        }

        return $bundles;
    }
... lines 33 - 52
}
```

The kernel is the *heart* of your Symfony application... but it really doesn't do much. Its main job is to initialize all the *bundles* we need. A bundle is basically just a Symfony plugin, and *its* main job is to add *services* to your container.

Remember that giant list from a minute ago? Yep, *every single* service in that list is provided to us from one of these bundles.

But at its simplest: a bundle is basically just a directory full of PHP classes, configuration and other goodies. And hey, we have our own: AppBundle:

54 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 20
            new AppBundle\AppBundle(),
        );
... lines 23 - 31
    }
... lines 33 - 52
}
```

Install a Bundle: Get more Services

I have challenge for us: I want to render some of this octopus information through a markdown parser. So the question is, does Symfony already have a markdown parsing service?

I don't know! Let's find out via debug:container:

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

Hmm, nothing: there's no built-in tool to help us.

Symfony community to the rescue! If you're missing a tool, there *might* be a Symfony bundle that provides it. In this case, there is: it's called KnpMarkdownBundle.

Copy its composer require line. You don't need to include the version constraint: Composer will figure that out for us. Run that in your terminal:

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

Let's keep busy while that's working. To enable the bundle, grab the new statement from the docs and paste that into AppKernel: the order of these doesn't matter:

```
54 lines | app/AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 18
            new Knp\Bundle\MarkdownBundle\KnpMarkdownBundle(),
... lines 20 - 21
        );
... lines 23 - 31
    }
... lines 33 - 52
}
```

That's it! Just wait for Composer to finish its job... and maybe send a nice tweet to Jordi - he's the creator and maintainer of Composer. There we go!

Ok, before we do *anything* else, let's run an experiment. Try running debug:container again with a search for markdown.

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

Boom! Suddenly, there are *two* services matching. These are coming from the bundle we just installed. The one we're really interested in is markdown.parser.

Chapter 2: Using a Service

Ok! Let's see if we can use this new service. First, some setup: in the template, remove the fun fact text and move it into GenusController by creating a new $funFact variable:

```
44 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 10
class GenusController extends Controller
{
    /**
     * @Route("/genus/{genusName}")
     */
    public function showAction($genusName)
    {
        $funFact = 'Octopuses can change the color of their body in just *three-tenths* of a second!';
... lines 19 - 23
    }
... lines 25 - 42
}
```

Let's make things interesting by adding some asterisks around three-tenths - Markdown should eventually turn that into italics.

Pass funFact into the template:

```
44 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 10
class GenusController extends Controller
{
... lines 13 - 15
    public function showAction($genusName)
    {
        $funFact = 'Octopuses can change the color of their body in just *three-tenths* of a second!';

        return $this->render('genus/show.html.twig', array(
            'name' => $genusName,
            'funFact' => $funFact,
        ));
    }
... lines 25 - 42
}
```

And render it with the normal {{ funFact }}:

40 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 15
                <dt>Fun Fact:</dt>
                <dd>{{ funFact }}</dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 23 - 40
```

When we refresh the browser, we have the exact same text, but with the unparsed asterisks.

Now, how the heck can we use the new markdown.parser service to turn those asterisks into italics?

Fetch the Service and Use it!

Remember: we have access to the container from inside a controller. Start with $funFact = $this->container->get('markdown.parser'). Now we have that parser object and can call a method on it. The one we want is ->transform() - pass that the string to parse:

46 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 10
class GenusController extends Controller
{
... lines 13 - 15
    public function showAction($genusName)
    {
        $funFact = 'Octopuses can change the color of their body in just *three-tenths* of a second!';
        $funFact = $this->container->get('markdown.parser')
            ->transform($funFact);
... lines 21 - 25
    }
... lines 27 - 44
}
```

So.... how did I know this object has a transform() method? Well, a few ways. First, PhpStorm knows what the markdown.parser object is, so it gives me autocompletion. But you can always read the documentation of the bundle: it'll tell you *how* to use any services it gives us.

Ok team - time to try this out. Refresh! And hey, it's working! Ok, it's not *exactly* working. Open the page source: it looks like the parser is working its magic, but the HTML tags are being escaped into HTML entities.

This is Twig at work! One of the *best* features of Twig is that it automatically escapes any HTML that you render. That gives you *free* security from XSS attacks. And for those few times when you *do* want to print HTML, just add the |raw filter:

40 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 15
                <dt>Fun Fact:</dt>
                <dd>{{ funFact|raw }}</dd>
... lines 18 - 40
```

Refresh again: it's rending in some lovely italics.

Fetching Services the Lazy Way

One more thing! We can actually do all of this with *less* code. In the controller, replace $this->container->get() with just $this->get():

```
46 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 10
class GenusController extends Controller
{
... lines 13 - 15
    public function showAction($genusName)
    {
        $funFact = 'Octopuses can change the color of their body in just *three-tenths* of a second!';
        $funFact = $this->get('markdown.parser')
            ->transform($funFact);
... lines 21 - 25
    }
... lines 27 - 44
}
```

This does the same thing as before.

Ok: here's the *big*, giant important takeaway so far: why do we add bundles to our app? Because bundles put more services in our container. And services are tools.

So my question now is: how can we configure these services so that they do *exactly* what we need them to?

Chapter 3: Config.yml: Control Center for Services

Ok, I get it: I bring in a bundle and it gives me more useful objects. But, there *must* be a way for us to configure and *control* how these services behave, right? Otherwise, how could we control what server the mailer uses for sending emails? Or what if we want to change how the logger works: making it log to a database instead of the default var/logs/dev.log file?

The answer to *all* of this lies in just *one* file: app/config/config.yml. That's right: *one* file is responsible for controlling everything from the log file to the database password. That's pretty powerful: so let's find out how it works!

Other than imports - which loads other files - and parameters - which we'll talk about soon - *every* root key in this file - like framework, twig and doctrine - corresponds to a *bundle* that is being configured:

72 lines | app/config/config.yml

```yaml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: services.yml }

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

framework:
... lines 12 - 35
# Twig Configuration
twig:
... lines 38 - 42
# Doctrine Configuration
doctrine:
... lines 45 - 72
```

All of this stuff under framework is configuration for the FrameworkBundle:

72 lines | app/config/config.yml

```yaml
... lines 1 - 10
framework:
    #esi:             ~
    #translator:      { fallbacks: ["%locale%"] }
    secret:          "%secret%"
    router:
        resource: "%kernel.root_dir%/config/routing.yml"
        strict_requirements: ~
    form:            ~
    csrf_protection: ~
    validation:      { enable_annotations: true }
    #serializer:      { enable_annotations: true }
    templating:
        engines: ['twig']
        #assets_version: SomeVersionScheme
    default_locale:  "%locale%"
    trusted_hosts:   ~
    trusted_proxies: ~
    session:
        # handler_id set to null will use default session handler from php.ini
        handler_id:  ~
        save_path:   "%kernel.root_dir%/../var/sessions/%kernel.environment%"
    fragments:       ~
    http_method_override: true
    assets: ~
... lines 35 - 72
```

Everything under twig is used to *control* the behavior of the services from TwigBundle:

72 lines | app/config/config.yml

```yaml
... lines 1 - 35
# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
... lines 40 - 72
```

The job of a bundle is to give us services. And this is *our* chance to *tweak* how those services behave.

Get a big List of all Configuration

That's amazing! Except... how are we supposed to know *what* keys can live under each of these sections? Documentation of course! There's a great reference section on symfony.com that shows you *everything* you can control for each bundle.

But I'll show you an even cooler way.

Head back to terminal and use our favorite ./bin/console to run config:dump-reference:

```
$ ./bin/console config:dump-reference
```

Actually, there's a shorter version of this called debug:config. This shows us a map with the bundle name on the left and the "extension alias" on the right... that's a fancy way of saying the root config key.

Tip

You can also use the shorter: ./bin/console debug:config command.

That's not really that useful. But re-run it with an argument: twig:

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

Woh! It dumped a giant yml example of *everything* you can configure under the twig key. Ok, it's not all documented... but honestly, this is usually enough to find what you need.

Playing with Configuration

Ok, lets's experiment! Obviously, the render() function we use in the controller leverages a twig service behind the scenes. Pretend that the number of known species is this big 99999 number and send that through a built-in filter called number_format:

```
40 lines | app/Resources/views/genus/show.html.twig
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ name }}</h2>

    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 14
                <dd>{{ '99999'|number_format }}</dd>
... lines 16 - 17
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 23 - 40
```

Refresh! That filter gives us a nice, 99,999, formatted-string. But what if we lived in a country that formats using a . instead? Time to panic!!?? Of course not: the bundle that gives us the twig service *probably* gives us a way to control this behavior.

How? In the config:dump-reference dump, there's a number_format:, thousands_separator key. In config.yml, add number_format: then thousands_separator: '.':

```
72 lines | app/config/config.yml
... lines 1 - 35
# Twig Configuration
twig:
... lines 38 - 39
    number_format:
        thousands_separator: '.'
... lines 42 - 72
```

Behind the scenes, this changes how the service behaves. And when we refresh, that filter gives us 99.999.

If this makes sense, you'll be able to control virtually *every* behavior of *any* service in Symfony. And since *everything* is done with a service... well, that makes you pretty dangerous.

Now, what if you make a typo in this file? Does it just ignore your config? Hmm, try it out: rename this key to thousand_separators with an extra s. Refresh! Boom! A *huge* error! All of the configuration is *validated*: if you make a typo, Symfony has your back.

Chapter 4: Adding a Cache Service

I've got a pretty important new challenge. We're going to be rendering a lot of markdown... and we *don't* want to do this on every request - it's just too slow. We really need a way to cache the parsed markdown.

Hmm, so caching is yet *another* tool that we need. If we had a service that was really good at caching a string and letting us fetch it out later, that would be perfect! Fortunately, Symfony comes with a bundle called DoctrineCacheBundle that can give us *exactly* this.

Enabling DoctrineCacheBundle

First, double-check that you have the bundle in your composer.json file:

```
65 lines | composer.json
{
... lines 2 - 17
    "require": {
... lines 19 - 22
        "doctrine/doctrine-cache-bundle": "^1.2",
... lines 24 - 62
    }
}
```

If for some reason you don't, use Composer to download it.

The bundle lives in the vendor/ directory, but it isn't enabled. Do that in the AppKernel class with new DoctrineCacheBundle():

```
55 lines | app/AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 19
            new Doctrine\Bundle\DoctrineCacheBundle\DoctrineCacheBundle(),
... lines 21 - 22
        );
... lines 24 - 32
    }
... lines 34 - 53
}
```

That added a use statement on top of the class... which is great - but I'll move it down to be consistent with everything else. Awesome!

Once you've enabled a bundle, there are usually two more steps: configure it, then use it. And of course, this has its own documentation that'll explain all of this. But guys, we're already getting really *good* at Symfony. I bet we can figure out how to use it entirely on our own.

1) Configure the Bundle

First, we need to configure the bundle. To see what keys it has, find the terminal and run:

```
$ ./bin/console config:dump-reference
```

The list has a new entry: doctrine_cache. Re-run the command with this:

```
$ ./bin/console config:dump-reference doctrine_cache
```

Nice! There's our huge configuration example! Ok, ok: I don't expect you to just look at this and instantly know how to use the bundle. You really *should* read its documentation. But before long, you really *will* be able to configure new bundles really quickly - and maybe without needing their docs.

Configuring a Cache Service

Now, remember the goal: to get a cache service we can use to avoid processing markdown on each request. When we added KnpMarkdownBundle, we magically had a new service. But with this bundle, we need to *configure* each service we want.

Open up config.yml and add doctrine_cache. Below that, add a providers key:

```
77 lines | app/config/config.yml
... lines 1 - 72
doctrine_cache:
    providers:
... lines 75 - 77
```

Next, the config dump has a name key. This Prototype comment above that is a confusing term that means that we can call this name *anything* we want. Let's make it my_markdown_cache:

```
77 lines | app/config/config.yml
... lines 1 - 72
doctrine_cache:
    providers:
        my_markdown_cache:
... lines 76 - 77
```

You'll see how that's important in a second.

Finally, tell Doctrine what *type* of cache this is by setting type to file_system:

```
77 lines | app/config/config.yml
... lines 1 - 72
doctrine_cache:
    providers:
        my_markdown_cache:
            type: file_system
```

This is just *one* of the built-in types this bundle offers: its docs would tell you the others.

And that's it! In the terminal run ./bin/console debug:container and search for markdown_cache:

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

Et voila! We have a new service called doctrine_cache.providers.my_markdown_cache. That's the *whole* point of this bundle: we describe a cache system we want, and it configures a service for that. Now we're dangerous.

Chapter 5: Configuring DoctrineCacheBundle

We have a new toy, I mean service! In GenusController, let's use it: $cache = $this->get('') and start typing markdown. Ah, there's our service!

```
55 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 10
class GenusController extends Controller
{
... lines 13 - 15
    public function showAction($genusName)
    {
        $funFact = 'Octopuses can change the color of their body in just *three-tenths* of a second!';

        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
... lines 21 - 34
    }
... lines 36 - 53
}
```

Here's the goal: make sure the same string doesn't get parsed twice through markdown. To do that, create a $key = md5($funFact);:

```
55 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 19
        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
        $key = md5($funFact);
... lines 22 - 55
```

To use the cache service, add if ($cache->contains($key)). In this case, just set $funFact = $cache->fetch($key);:

```
55 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 19
        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
        $key = md5($funFact);
        if ($cache->contains($key)) {
            $funFact = $cache->fetch($key);
... lines 24 - 28
        }
... lines 30 - 55
```

Else, we're going to need to parse through Markdown.

Let's live *dangerously*: add a sleep(1); to pretend like our markdown transformation is *really* taking a long time. Next, parse the fun fact and finish with $cache->save($key, $funFact):

```
55 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 21
        if ($cache->contains($key)) {
            $funFact = $cache->fetch($key);
        } else {
            sleep(1); // fake how slow this could be
            $funFact = $this->get('markdown.parser')
                ->transform($funFact);
            $cache->save($key, $funFact);
        }
... lines 30 - 55
```

Let's do this! Go back to the browser... and watch closely my friends. Refresh! Things are moving a bit slow. Yep: the web debug toolbar shows us 1048 ms. Refresh again. super fast! Just 41 ms: the cache is working! But... uhh... where is this being cached exactly?

Out of the box, the answer is in var/cache. In the terminal, ls var/cache, then ls var/cache/dev:

```
$ ls var/cache/dev/
```

Hey! There's a doctrine directory with cache/file_system inside. *There* is our cached markdown.

This bundle *assumed* that this is where we want to cache things. That was really convenient, but clearly, there needs to be a way to control that as well.

Configuring the Cache Path

Rerun config:dump-reference doctrine_cache:

```
$ ./bin/console config:dump-reference doctrine_cache
```

Ah, there's a directory key we can use to control things. In the editor, add this new directory key and set it to /tmp/doctrine_cache for now:

```
78 lines | app/config/config.yml
... lines 1 - 72
doctrine_cache:
    providers:
        my_markdown_cache:
            type: file_system
            directory: /tmp/doctrine_cache
```

Ok, back to the browser team. Refresh! Poseidon's beard, check out that HUGE error! "Unrecognized option 'directory'". Remember, configuration is validated... which means we just messed something up. Oh yea: I see the problem: I need to have file_system and *then* directory below that. Add file_system above directory and then indent it to make things match:

```
79 lines | app/config/config.yml
... lines 1 - 72
doctrine_cache:
    providers:
        my_markdown_cache:
            type: file_system
            file_system:
                directory: /tmp/doctrine_cache
```

Ok, try this out one more time.

It should be slow the first time... yes! And then super fast the second time. In the terminal, we can even see a /tmp/doctrine_cache/ directory:

```
$ ls /tmp/doctrine_cache/
```

Big picture time: bundles give you services, and those services can be controlled in config.yml. Every bundle works a little bit different - but if you understand this basic concept, you're on your way.

Chapter 6: Environments

Question: if config.yml is so important - then what the heck is the point of all of these other files - like config_dev.yml, config_test.yml, parameters.yml, security.yml and services.yml. What is their purpose?

What is an Environment?

The answer is *environments*. Now, I don't mean environments like dev, staging, or production on your servers. In Symfony, an environment is a set of configuration. Environments are also one of its most *powerful* features.

Think about it: an application is a big collection of code. But to get that code running it needs configuration. It needs to know what your database password is, what file your logger should write to, and at what priority of messages it should bother logging.

The dev and prod Environments

Symfony has *two* environments by default: dev and prod. In the dev environment your code is booted with a lot of logging and debugging tools. But in the prod environment, that same code is booted with minimal logging and other configuration that makes everything fast.

Tip

Actually, there's a third environment called test that you might use while writing automated tests.

So.... how do we choose which environment we're using? And what environment have we been using so far?

app.php versus app_dev.php

The answer to that lives in the web directory, which is the *document root*. This is the only directory whose files can be accessed publicly.

These two files - app.php and app_dev.php - are the keys. When you visit your app, you're always executing *one* of these files. Since we're using the server:run built-in web server: we're executing app_dev.php:

33 lines | web/app_dev.php

```
... lines 1 - 2
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Debug\Debug;

// If you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#checking-symfony-application-configuration-and-setup
// for more information
//umask(0000);

// This check prevents access to debug front controllers that are deployed by accident to production servers.
// Feel free to remove this, extend it, or make something more sophisticated.
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !(in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', 'fe80::1', '::1')) || php_sapi_name() === 'cli-server')
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).' for more information.');
}

/**
 * @var Composer\Autoload\ClassLoader $loader
 */
$loader = require __DIR__.'/../app/autoload.php';
Debug::enable();

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

The web server is preconfigured to hit this file.

That means that when we go to localhost:8000/genus/octopus that's equivalent to going to localhost:8000/app_dev.php/genus/octopus. With that URL, the page still loads *exactly* like before.

So how can we switch to the prod environment? Just copy that URL and change app_dev.php to app.php. Welcome to the prod environment: same app, but no web debug toolbar or other dev tools:

33 lines | web/app.php

```
... lines 1 - 2
use Symfony\Component\HttpFoundation\Request;

/**
 * @var Composer\Autoload\ClassLoader
 */
$loader = require __DIR__.'/../app/autoload.php';
include_once __DIR__.'/../var/bootstrap.php.cache';

// Enable APC for autoloading to improve performance.
// You should change the ApcClassLoader first argument to a unique prefix
// in order to prevent cache key conflicts with other applications
// also using APC.
/*
$apcLoader = new Symfony\Component\ClassLoader\ApcClassLoader(sha1(__FILE__), $loader);
$loader->unregister();
$apcLoader->register(true);
*/

//require_once __DIR__.'/../app/AppCache.php';

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
//$kernel = new AppCache($kernel);

// When using the HttpCache, you need to call the method in your front controller instead of relying on the configuration parameter
//Request::enableHttpMethodParameterOverride();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

This baby is optimized for speed.

But don't worry: in production you won't have this ugly app.php in your URL: you'll configure your web server to execute that file when nothing appears in the URL.

So this is how you "choose" your environment. And other than on your production server, you'll pretty much always want to be in the dev environment.

But the real fun starts next: when we learn how to bend and optimize each environment exactly to our needs.

Chapter 7: config_dev.yml & config_prod.yml

We have two environments! That's *super* flexible... or I think it will be, just as soon as we figure out how the heck we can *configure* each environment.

Compare app.php and app_dev.php. What are the differences? Ok, ignore that big if block in app_dev.php. The *important* difference is a single line: the one that starts with $kernel = new AppKernel(). Hey, that's the class that lives in the app/ directory!

The first argument to AppKernel is prod in app.php:

```
33 lines | web/app.php

... lines 1 - 22
$kernel = new AppKernel('prod', false);
... lines 24 - 33
```

And dev in app_dev.php:

```
33 lines | web/app_dev.php

... lines 1 - 26
$kernel = new AppKernel('dev', true);
... lines 28 - 33
```

This defines your environment. The second argument - true or false - is a debug flag and basically controls whether or not errors should be shown. That's less important.

config_dev.yml versus config_prod.yml

Now, what do these dev and prod strings do? Here's the secret: when Symfony boots, it loads only *one* configuration file for the entire system. And no, it's *not* config.yml - I was lying to you. Sorry about that. No, the dev environment loads *only* config_dev.yml:

46 lines | app/config/config_dev.yml

```yaml
imports:
    - { resource: config.yml }

framework:
    router:
        resource: "%kernel.root_dir%/config/routing_dev.yml"
        strict_requirements: true
    profiler: { only_exceptions: false }

web_profiler:
    toolbar: true
    intercept_redirects: false

monolog:
    handlers:
        main:
            type:   stream
            path:   "%kernel.logs_dir%/%kernel.environment%.log"
            level:  debug
        console:
            type:   console
            bubble: false
            verbosity_levels:
                VERBOSITY_VERBOSE: INFO
                VERBOSITY_VERY_VERBOSE: DEBUG
            channels: ["!doctrine"]
        console_very_verbose:
            type:   console
            bubble: false
            verbosity_levels:
                VERBOSITY_VERBOSE: NOTICE
                VERBOSITY_VERY_VERBOSE: NOTICE
                VERBOSITY_DEBUG: DEBUG
            channels: ["doctrine"]
        # uncomment to get logging in your browser
        # you may have to allow bigger header sizes in your Web server configuration
        #firephp:
        #    type:   firephp
        #    level:  info
        #chromephp:
        #    type:   chromephp
        #    level:  info

#swiftmailer:
#    delivery_address: me@example.com
```

And in prod, the only file it loads is config_prod.yml:

28 lines | app/config/config_prod.yml

```yaml
imports:
    - { resource: config.yml }

#framework:
#    validation:
#        cache: validator.mapping.cache.apc
#    serializer:
#        cache: serializer.mapping.cache.apc

#doctrine:
#    orm:
#        metadata_cache_driver: apc
#        result_cache_driver: apc
#        query_cache_driver: apc

monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: error
            handler:      nested
        nested:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
        console:
            type:  console
```

Ok fellow deep sea explorers, this is where things get cool! Look at the first line of config_dev.yml:

46 lines | app/config/config_dev.yml

```yaml
imports:
    - { resource: config.yml }
... lines 3 - 46
```

What does it do? It *imports* the main config.yml: the main *shared* configuration. Then, it overrides any configuration that's special for the dev environment.

Check this out! Under monolog - which is the bundle that gives us the logger service - it configures extra logging for the dev environment:

46 lines | app/config/config_dev.yml

```yaml
... lines 1 - 13
monolog:
    handlers:
        main:
... lines 17 - 18
            level:  debug
... lines 20 - 46
```

By setting level to debug, we're saying "log everything no matter its priority!"

So what about config_prod.yml? No surprise: it does the *exact* same thing: it loads the main config.yml file and then overrides things:

```
28 lines | app/config/config_prod.yml
imports:
    - { resource: config.yml }
... lines 3 - 28
```

This file has a similar setup for the logger, but now it says action_level: error:

```
28 lines | app/config/config_prod.yml
... lines 1 - 15
monolog:
    handlers:
        main:
... line 19
            action_level: error
... lines 21 - 28
```

This only logs messages that are at or above the error level. So only messages when things break.

Experimenting with config_dev.yml

Let's play around a bit with the dev environment! Under monolog, uncomment the firephp line:

```
46 lines | app/config/config_dev.yml
... lines 1 - 13
monolog:
    handlers:
... lines 16 - 34
        # uncomment to get logging in your browser
        # you may have to allow bigger header sizes in your Web server configuration
        firephp:
            type:   firephp
            level:  info
... lines 40 - 46
```

This is a cool handler that will show you log messages *right* in your browser.

Head over to it and run "Inspect Element". Make sure that the URL will access the dev environment and then refresh. And check *this* out: a bunch of messages telling us what route was matched. Heck, we can even see what route was matched for our ajax call. To see this working, you'll need a FirePHP extension installed in your browser. In your app, Monolog is attaching these messages to your response headers, and the extension is reading those. We don't want this to happen on production, so we only enabled this in the dev environment.

Environments are awesome! So how could we use them *only* cache our markdown string in the prod environment?

Chapter 8: Caching in the prod Environment Only

We absolutely need to cache our markdown processing. But what if we need to tweak how the markdown renders? In that case, we *don't* want caching. Could we somehow *disable* caching in the dev environment only?

Yes! Copy the doctrine_cache from config.yml and paste it into config_dev.yml. Next, change type from file_system to array:

```
51 lines | app/config/config_dev.yml
... lines 1 - 46
doctrine_cache:
    providers:
        my_markdown_cache:
            type: array
```

The array type is basically a "fake" cache: it won't ever store anything.

Yea, that's it! Head back to the browser and refresh. This is definitely *not* caching: it takes an entire second because of the sleep(). Try again. *Still* not caching! Now, change to the prod environment and refresh. Beautiful, this is still really, really fast. Well that was easy.

Clearing prod Cache

Ok, this *did* work, but there's a small gotcha we're not considering. In config.yml, change that thousands_separator back to a comma:

```
79 lines | app/config/config.yml
... lines 1 - 35
# Twig Configuration
twig:
... lines 38 - 39
    number_format:
        thousands_separator: ','
... lines 42 - 79
```

Try this first in the dev environment: Yep! No problems. Now refresh in the prod environment. Huh, it's *still* a period. What gives?

Here's the thing: the prod environment is primed for speed. And that means, when you change *any* configuration, you need to manually clear the cache before you'll see those changes.

How do you do that? Simple: in the terminal, run:

```
$ ./bin/console cache:clear --env=prod
```

You see, even the console script executes your app in a specific environment. By default it uses the dev environment. Normally, you don't really care. But for a few commands, you'll want to switch using this flag.

Ok, back to the browser. Refresh in prod. Boom! There's our comma!

The other Files: services.yml, security.yml, etc

Alright, what about all of these *other* configuration files. It turns out, there all part of the *exact* same configuration system we've just mastered. So where is parameters.yml loaded? Well, at the top of config.yml its imported, along with security.yml and services.yml:

```
79 lines | app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: services.yml }
... lines 5 - 79
```

The key point is that *all* of the files are just loading each other: it's all the same system.

In fact, I could copy all of security.yml, paste it into config.yml, completely delete security.yml and everything would be fine. In fact, the only reason security.yml even exists is because it *feels* good to keep that stuff in its own file. The same goes for services.yml - a *big* topic we'll talk about in the future.

Now, parameters.yml *is* a little special. Let's find out why.

Chapter 9: Parameters: The Variables of Configuration

Congrats! We've basically mastered Symfony configuration and environment system. But there's just *one* more trick it has up its sleeve.

Look closely inside config.yml file: one of the settings - default_locale - is set to a strange-looking value: %locale%. Huh.

79 lines | app/config/config.yml

```
... lines 1 - 10
framework:
... lines 12 - 24
    default_locale:  "%locale%"
... lines 26 - 79
```

Scrolling up a bit, there's another root key called parameters with locale: en:

79 lines | app/config/config.yml

```
... lines 1 - 5
# Put parameters here that don't need to change on each machine where the app is deployed
# http://symfony.com/doc/current/best_practices/configuration.html#application-related-configuration
parameters:
    locale: en
... lines 10 - 79
```

You're witnessing the power of a special "variable system" inside config files. Here's how it works: in any of these configuration files, you can have a parameters key. And below that you can create variables like locale and set that to a value. Why is this cool? Because you can then reuse that value in *any* other file by saying %locale%.

Look under the doctrine key:

79 lines | app/config/config.yml

```
... lines 1 - 42
# Doctrine Configuration
doctrine:
    dbal:
... line 46
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
... lines 52 - 79
```

Hey, a bunch more, like %database_host% and %database_port%. These are set just like locale, but in a different file: parameters.yml:

```
20 lines | app/config/parameters.yml.dist
# This file is a "template" of what your parameters.yml file should look like
# Set parameters here that may be different on each deployment target of the app, e.g. development, staging, production.
# http://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
parameters:
    database_host:     127.0.0.1
    database_port:     ~
    database_name:     symfony
    database_user:     root
    database_password: ~
... lines 10 - 20
```

So that's it! If you add a new key under parameters, you can use that in any other file by saying %parameter_name%.

And just like services, we can get a *list* of every parameter available. How? Ah, our friend the console of course. Run:

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

Woah, that's a *huge* list you can take advantage of or even override. Most of these you won't care about, but don't forget this little cheatsheet is there for you.

Creating a new Parameter

We can leverage parameters to do something *really* cool with our cache setup.

In the prod environment, we use the file_system cache. In dev, we use array. We can improve this. Create a new parameter called cache_type and set that to file_system. Scroll down and set type to %cache_type%:

```
80 lines | app/config/config.yml
... lines 1 - 7
parameters:
... line 9
    cache_type: file_system
... lines 11 - 73
doctrine_cache:
    providers:
        my_markdown_cache:
            type: %cache_type%
... lines 78 - 80
```

Run over in the terminal to see if the parameter showed up:

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

It's right on top. Cool! Clear the cache in the prod environment so we can double-check everything is still working:

```
$ ./bin/console cache:clear --env=prod
```

Ok good - now refresh using app.php. It's still loading fast - so we haven't broken anything... yet.

Here's where things get interesting. In config_dev.yml, it took a lot of code *just* to turn caching off. Parameters to the rescue! Copy the parameters key from config.yml and paste it into this file. But now, change its value to array and celebrate by completely removing the doctrine_cache key at the bottom:

```
49 lines | app/config/config_dev.yml
... lines 1 - 3
parameters:
    cache_type: array
... lines 6 - 49
```

That's it! Refresh the browser in the dev environment: *great* it's still slow, which means it's working.

Chapter 10: parameters.yml & %kernel.root_dir%

There are some really *special* parameters I need to tell you about. In this big list that debug:container gave us, find the group that starts with kernel.. You won't find these defined anywhere: they're baked right into Symfony and are some of the *most* useful parameters.

Notice kernel.debug - whether or not we're in debug mode - and kernel.environment. But the best ones to know about are kernel.cache_dir - where Symfony stores its cache - and kernel.root_dir - which is actually the app/ directory where the AppKernel class lives. Anytime you need to reference a path in your project, use kernel.root_dir and build the path from it.

Earlier, just to show off, we configured the DoctrineCacheBundle to store the markdown cache in /tmp/doctrine_cache:

```
80 lines | app/config/config.yml
... lines 1 - 73
doctrine_cache:
    providers:
        my_markdown_cache:
            type: %cache_type%
            file_system:
                directory: /tmp/doctrine_cache
```

Referencing absolute paths is a little weird: why not just store this stuff in Symfony's cache dir? Ok, ok, the bundle actually did this by default, before we started messing with the configuration. But we're learning people! So let's use one of these new kernel. parameters to fix this.

How? Just change the directory to %kernel.cache_dir% then /markdown_cache:

```
80 lines | app/config/config.yml
... lines 1 - 73
doctrine_cache:
    providers:
        my_markdown_cache:
            type: %cache_type%
            file_system:
                directory: %kernel.cache_dir%/markdown_cache
```

It's totally ok to mix the parameters inside larger strings.

Clear the cache in the prod environment:

```
$ ./bin/console cache:clear --env=prod
```

And switch to the prod tab to try this all out. Now, in the terminal:

```
$ ls var/cache/prod/
```

And there's our cached markdown.

Why is parameters.yml Special?

I have a question: if config.yml imports parameters.yml, then why bother having this file at all? Why not just put all the parameters at the top of config.yml?

Here's why: parameters.yml holds *any* configuration that will be different from one machine where the code is deployed to another.

For example, your database password is most likely *not* the same as *my* database password and hopefully not the same as the production database password. But if we put that password right in the middle of config.yml, that would be a nightmare! In that scenario *I* would probably commit my password to git and then you would have to change it to *your* password but then try to not commit that change. Gross.

Instead of that confusing mess of seaweed, we use parameters in config.yml. This allows us to isolate all the machine-specific configuration to parameters.yml. And here's the final key: parameters.yml is *not* committed to the repository - you can see there's an entry for it in .gitignore:

17 lines | .gitignore

```
/app/config/parameters.yml
... lines 2 - 17
```

Of course, if I just cloned this project, and I won't have a parameters.yml file: I have to create it manually. Actually, this is the *exact* reason for this other file: parameters.yml.dist:

20 lines | app/config/parameters.yml.dist

```
# This file is a "template" of what your parameters.yml file should look like
# Set parameters here that may be different on each deployment target of the app, e.g. development, staging, production.
# http://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
parameters:
    database_host:     127.0.0.1
    database_port:     ~
    database_name:     symfony
    database_user:     root
    database_password: ~
    # You should uncomment this if you want use pdo_sqlite
    # database_path: "%kernel.root_dir%/data.db3"

    mailer_transport:  smtp
    mailer_host:       127.0.0.1
    mailer_user:       ~
    mailer_password:   ~

    # A secret key that's used to generate certain security-related tokens
    secret:            ThisTokenIsNotSoSecretChangeIt
```

This is not read by Symfony, it's just a template of all of the parameters this project needs. If you add or remove things from the parameters.yml, be sure to add or remove them from parameters.yml.dist. You *do* commit this file to git.

Tip

Due to a post-install command in your composer.json, after running composer install, Symfony *will* read parameters.yml.dist and ask you to fill in any values that are missing from parameters.yml.

Let's put this into practice. What if our app does *not* need to send emails. That means we *don't* need SwiftmailerBundle. And *that* means we don't need any of these mailer_ parameters: these are used in config.yml under swiftmailer. We could keep this stuff, but why not get rid of the extra stuff?

In AppKernel, start by removing the SwiftmailerBundle line completely:

```
55 lines | app/AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 14
            //new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
... lines 16 - 22
        );
... lines 24 - 32
    }
... lines 34 - 53
}
```

Because that's gone, you'll need to remove the entire swiftmailer section in config.yml. And finally, we don't need the mailer_ parameters anymore, so delete them from parameters.yml *and* parameters.yml.dist so other devs won't worry about adding them. Awesome!

Head over to the terminal and run:

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

Cool - the app still runs, but there are *no* services that match mailer anymore.

Chapter 11: Mastering Route config Loading

Ok, you're a configuration pro: you know where services come from and how to configure them. Here's our question: where do routes come from? And can third party bundles give us more routes?

The answer is, no :(. Ah, I'm kidding - they totally can. In fact, find your terminal and run:

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

Surprise! There are a lot more than just the two routes that we've created. Where are these coming from?

Answer time: when Symfony loads the route list, it only loads *one* routing file. In the dev environment, it loads routing_dev.yml:

15 lines | app/config/routing_dev.yml

```
_wdt:
    resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
    prefix:   /_wdt

_profiler:
    resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
    prefix:   /_profiler

_errors:
    resource: "@TwigBundle/Resources/config/routing/errors.xml"
    prefix:   /_error

_main:
    resource: routing.yml
```

In *all* environments, it loads routing.yml:

9 lines | app/config/routing.yml

```
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
... lines 4 - 9
```

Inside routing_dev.yml, we're importing additional routes from WebProfilerBundle and the TwigBundle:

15 lines | app/config/routing_dev.yml

```
_wdt:
    resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
    prefix:   /_wdt

_profiler:
    resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
    prefix:   /_profiler

_errors:
    resource: "@TwigBundle/Resources/config/routing/errors.xml"
    prefix:   /_error
... lines 12 - 15
```

That's why there are *extra* routes in debug:router. This *also* imports routing.yml at the bottom to load all the main routes:

```
15 lines | app/config/routing_dev.yml
... lines 1 - 12
_main:
    resource: routing.yml
```

You can really import *anything* you want: you're in complete control of what routes you import from a bundle. These files are usually XML, but that doesn't make any difference.

In the main routing.yml, there's yet *another* import:

```
9 lines | app/config/routing.yml
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
... lines 4 - 9
```

We're out of control! This loads annotation routes from the controllers in our bundle.

Creating YAML Routes

But you can also create routes right inside YAML - a lot of people actually prefer this over annotations. The difference is *purely* preference: there's no performance impact to either.

Tip

In a *big* project, there is actually *some* performance impact in the dev environment for using annotations.

Let's create a homepage route: use the key homepage - that's its name. Then add path set to just /:

```
9 lines | app/config/routing.yml
... lines 1 - 4
homepage:
    path: /
... lines 7 - 9
```

Now, we need to point this route to a controller: the function that will render it. Doing this in yml is a bit more work. Add a defaults key and an _controller key below that. Set this to AppBundle:Main:homepage:

```
9 lines | app/config/routing.yml
... lines 1 - 4
homepage:
    path: /
    defaults:
        _controller: AppBundle:Main:homepage
```

Yes yes, this looks totally weird - this is a Symfony-specific shortcut. It means that we will have a controller in AppBundle called MainController with a method named homepageAction().

In the Controller directory, create that new MainController class:

```
13 lines | src/AppBundle/Controller/MainController.php
... lines 1 - 2
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class MainController extends Controller
{
... lines 9 - 12
}
```

Next add public function homepageAction(). Make this class extend Symfony's base Controller so that we can access the render() function. Set it to render main/homepage.html.twig without passing any variables:

```
13 lines | src/AppBundle/Controller/MainController.php
... lines 1 - 6
class MainController extends Controller
{
    public function homepageAction()
    {
        return $this->render('main/homepage.html.twig');
    }
}
```

Create this *real* quick in app/Resources/views: new file, main/homepage.html.twig:

```
6 lines | app/Resources/views/main/homepage.html.twig
{% extends 'base.html.twig' %}

{% block body %}
... line 4
{% endblock %}
```

Templates basically always look the same: extend the base template - base.html.twig - and then override one or more of its blocks. Override block body: the block that holds the main content in base.html.twig.

Add some content to greet the aquanauts:

```
6 lines | app/Resources/views/main/homepage.html.twig
... lines 1 - 2
{% block body %}
    <h1 class="page-header text-center">Welcome Aquanauts!</h1>
{% endblock %}
```

Done! Before trying this, head over to the terminal and run:

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

There it is! There are many ways to define a route, but the result is exactly the same. Refresh the browser. Thank you yml for that lovely brand new route.

Woh guys. This course was a *serious* step forward. In the next parts, we're going to start adding big features: like talking to a database, forms, security and more. And when we do, I've got some exciting news: because of your work here, it's all going to make sense. Let's keep going and get really productive with Symfony.