Journey to the Center of Symfony: The Dependency Injection Container

With <3 from SymfonyCasts

Chapter 1: Creating a Container in the Wild

A whole mini-series on Symfony's Dependency Injection Container? Yes! Do you want to *really* understand how Symfony works - and also Drupal 8? Then you're in the right place.

That's because Symfony is 2 parts. The first is the request/routing/controller/response and event listener flow we talked about in the first Symfony Journey part. The second half is all about the container. Understand it, and you'll unlock everything.

Setting up the Playground

Symfony normally gives you the built container. Instead of that, let's do a DIY project and create that by hand. Actually, let's get out of Symfony entirely. Inside the directory of our project, create a new folder called dino_container. We're going to create a PHP file in here where we can mess around - how about roar.php.

This file is *all* alone - it has *nothing* to do with the framework or our project at all. We're flying solo.

I'll add a namespace Dino\Play - but only because it makes PhpStorm auto-complete my use statements nicely.

Let's require the project's autoloader - so go up one directory, then get vendor/autoload.php:

11 lines | dino_container/roar.php

```
<?php

namespace Dino\Play;

use Monolog\Logger;

require __DIR__.'/../vendor/autoload.php';
... lines 8 - 11
```

Great, now we can access Symfony's DependencyInjection classes and a few other libraries we'll use, like Monolog.

Using Monolog Straight-Up

In fact, forget about containers and Symfony and all that weird stuff. Let's *just* use the Monolog library to log some stuff. That's simple, just $logger = new Logger(). The first argument is the channel - it's like a category - you'll see that word main in the logs. Now log something: $logger->info(), then ROOOAR:

11 lines | dino_container/roar.php

```
... lines 1 - 4
use Monolog\Logger;

require __DIR__.'/../vendor/autoload.php';

$logger = new Logger('main');
$logger->info('ROOOOAR');
```

Ok, let's see if we can get this script to yell back at us. Run it with:

```
$ php dino_container/roar.php
```

Fantastic! If you don't do anything, Monolog spits your messages out into stderr.

To pretend like this little file is an application, I'll create a runApp() function that does the yelling. Pass it a $logger argument and move our info() call inside:

16 lines | dino_container/roar.php

```
... lines 1 - 8
$logger = new Logger('main');
runApp($logger);

function runApp(Logger $logger)
{
    $logger->info('ROOOOAR');
}
```

I'm just doing this to separate my setup code - the part where I configure objects like the Logger - from my *real* application, which in this case, roars at us. It still works like before.

Create a Container

Now, to the container? First, the basics:

1. A service is just a fancy name a computer science major made up to describe a useful object. A logger is a useful object, so it's a service. A mailer object, a database connection object and an object that talks to your coffee maker's API: all useful objects, all services.

2. A container is an object, but it's really just an associative array that holds all your service objects. You ask it for a service by some nickname, and it gives you back that object. And it has some other super-powers that we'll see later.

Got it? Great, create a $container variable and set it to a new ContainerBuilder object.

19 lines | dino_container/roar.php

```
... lines 1 - 5
use Symfony\Component\DependencyInjection\ContainerBuilder;
... lines 7 - 9
$container = new ContainerBuilder();
... lines 11 - 19
```

Hello Mr Container! Later, we'll see why Mr Container is called a builder.

Working with it is simple: use set to put a service into it, and get to fetch that back later. Call set and pass it the string logger. That's the key for the service - it's like a nickname, and we could use anything we want.

TIP The standard is to use lowercase characters, numbers, underscores and periods. Some other characters are illegal and while service ids are case *insensitive*, using lower-cased characters is faster. Want details? See github.com/knpuniversity/symfony-journey/issues/5.

Then pass the $logger object:

19 lines | dino_container/roar.php

```
... lines 1 - 9
$container = new ContainerBuilder();
$logger = new Logger('main');
$container->set('logger', $logger);
... lines 13 - 19
```

Now, pass $container to runApp instead of the logger and update its argument. To fetch the logger from the container, I'll say $container->get() then the key - logger:

19 lines | dino_container/roar.php

```
... lines 1 - 11
$container->set('logger', $logger);
runApp($container);

function runApp(ContainerBuilder $container)
{
    $container->get('logger')->info('ROOOOAR');
}
```

The logger service goes into the container with set, and it comes back out with get. No magic.

Test it out:

```
$ php dino_container/roar.php
```

Yep, still roaring.

Adding a Second Service

A real project will have *a lot* of services - maybe hundreds. Let's add a second one. When you log something, monolog passes that to handlers, and they actually do the work, like adding it to a log file or a database.

Create a new StreamHandler object - we can use it to save things to a file. We'll stream logs into a dino.log file:

22 lines | dino_container/roar.php

```
... lines 1 - 3

... lines 5 - 12
$handler = new StreamHandler(__DIR__.'/dino.log');
... lines 14 - 22
```

Next, pass an array as the second argument to our Logger with this inside:

22 lines | dino_container/roar.php

```
... lines 1 - 12
$handler = new StreamHandler(__DIR__.'/dino.log');
$logger = new Logger('main', array($handler));
... lines 15 - 22
```

Cool, so try it out. Oh, no more message! It's OK. As soon as you pass at least *one* handler, Monolog uses that instead of dumping things out to the terminal. But we *do* now have a dino.log.

With things working, let's also put the stream handler into the container. So, $container->set() - and here we can make up any name, so how about logger.stream_handler. Then pass it the $streamHandler variable:

23 lines | dino_container/roar.php

```
... lines 1 - 12
$handler = new StreamHandler(__DIR__.'/dino.log');
$container->set('logger.stream_handler', $handler);
... lines 15 - 23
```

Down in the $logger, just fetch it out with $container->get('logger.stream_handler'):

23 lines | dino_container/roar.php

```
... lines 1 - 13
$container->set('logger.stream_handler', $handler);
$logger = new Logger('main', array($container->get('logger.stream_handler')));
... lines 16 - 23
```

PhpStorm is highlighting that line - don't let it boss you around. It gets a little confused when I create a Container from scratch inside a Symfony project.

Try it out:

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Good, no errors, and when we tail the log, 2 messages - awesome!

Up to now, the container isn't much more than a simple associative array. We put 2 things in, we get 2 things out. But we're not really exercising the true power of the container, yet.

Chapter 2: Definitions: Teach the Container

We've got two problems. First, our services are *always* created. What if we had a mailer service? You only need to mail something on a very small percentage of requests. With this setup, we'll spend time and memory on *every* request to create the mailer object, even though we don't need it. That's bananas! Especially if you have a big system.

Second, the services need to be created in order: we *have* to create the $streamHandler first so that it's available when we create the logger. If we reorder things, it'll blow up. With a big system where services are created in many places, this will get tricky fast.

Here's the answer: don't create the services. Instead, let's *teach* the container *how* to create them. Then *it* will create the objects when and *if* we ask for them.

This means that instead of creating a Logger, create an $loggerDefinition variable and set it to a new Definition object. For the first argument, pass it the class name - Monolog\Logger.

30 lines | dino_container/roar.php

```
... lines 1 - 16
$loggerDefinition = new Definition('Monolog\Logger');
... lines 18 - 30
```

This Definition object knows everything about *how* to instantiate an object. We'll use it to *teach* the container that when I ask for the logger service, this is *how* you should create it. So naturally, if the class has constructor arguments, we need to configure those. Do that with $loggerDefinition->setArguments(), and this takes an array of the arguments. The first is just a string: main:

30 lines | dino_container/roar.php

```
... lines 1 - 16
$loggerDefinition = new Definition('Monolog\Logger');
$loggerDefinition->setArguments(array(
    'main',
... line 20
));
... lines 22 - 30
```

The second argument is an array of handler objects. So you might expect me to just pass $container->get('logger.stream_handler'). But no! That would mean that I'd *still* have to worry about creating the stream handler first.

Instead, we can *refer* to the service by its id. Create a new Reference object and pass logger.stream_handler. This tells Symfony: "Hey, this argument isn't the string logger.stream_handler, it's a service with this id.":

30 lines | dino_container/roar.php

```
... lines 1 - 16
$loggerDefinition = new Definition('Monolog\Logger');
$loggerDefinition->setArguments(array(
    'main',
    array(new Reference('logger.stream_handler'))
));
... lines 22 - 30
```

To put this into the container, instead of calling set, use setDefinition with the nickname logger and the $loggerDefinition. Now get rid of the old lines that set the logger service directly:

30 lines | dino_container/roar.php

```
... lines 1 - 16
$loggerDefinition = new Definition('Monolog\Logger');
$loggerDefinition->setArguments(array(
    'main',
    array(new Reference('logger.stream_handler'))
));
$container->setDefinition('logger', $loggerDefinition);
... lines 23 - 30
```

The container now knows *how* to create the logger service. So later, when and *if* we ask for it, the container will create it in the background using these instructions.

And our app doesn't know or care this is happening - it just happily asks for that service. So let's try it out:

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Boom! 3 log entries now.

The disadvantage is that adding service is now more abstract: instead of creating them directly, you describe them. But the up-side is *huge*. Services aren't created unless you need them, the container will give you clear error messages if you mess something up, *and* the way this is all cached will blow your mind.

Oh, and now order doesn't matter. Down near the bottom, create a new Definition and pass it the StreamHandler class. We can remove the Logger and StreamHandler use statements too, because in a second, we won't be referencing these directly anymore. Just like before, call setArguments, pass it an array, and put the *one* constructor argument - the log file path - inside of it. Finish it off with $container->setDefinition(), passing the service nickname as the first argument, and the Definition next:

30 lines | dino_container/roar.php

```
... lines 1 - 19
$handlerDefinition = new Definition('Monolog\Handler\StreamHandler');
$handlerDefinition->setArguments(array(__DIR__.'/dino.log'));
$container->setDefinition('logger.stream_handler', $handlerDefinition);
... lines 23 - 30
```

And now that the logger.stream_handler service is being set with a Definition, we can remove the original code that set it directly.

So even though the logger service *needs* the stream handler service, we can describe them in any order. When we eventually ask for the logger, Symfony will go make the logger.stream_handler service first, then pass it to logger. That's why it's called a dependency injector container: it helps you manage dependencies.

But like before, our "app code" has no idea any of this is happening. So when we hit the script again, there's another log message:

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Chapter 3: Definition Unlocked

This Definition object is massively important to Symfony's container, and in the framework, they're built behind-the-scenes all over the place. I'll show you how in a bit.

Beyond using the class name and passing constructor arguments, there are a bunch of other things that you can train a Definition object to do. For example, what if you wanted the container to instantiate a service, but then also call a method on it *before* passing the service back?

Suppose that I want to log a message as soon as the logger is created - even before it's returned from the container. To do that, use addMethodCall(). The Logger class has a debug method on it - pass that as the first argument. This is equivalent to calling $logger->debug(). Then pass the array of arguments - we have just one, the message we want to log: "The logger just got started":

33 lines | dino_container/roar.php

```
... lines 1 - 12
$loggerDefinition = new Definition('Monolog\Logger');
... lines 14 - 17
$loggerDefinition->addMethodCall('debug', array(
    'Logger just got started'
));
... lines 21 - 33
```

With this, the container will create the logger, call the debug() method on it, and *then* pass it back. Test it!

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Brilliant! Now let's get harder.

There are actually *two* ways to add handlers to Logger: via the constructor, like we're doing now, OR by calling a pushHandler() method. Let's see if we can create a *second* handler and hook it up with a method call.

Start by creating the new Definition - it'll be a stream again, so re-use the StreamHandler class. This handler will dump the output to the console. To do that, call setArguments() like before, but this time, we'll pass a single argument: php://stdout. Whoops, and I'll fix my wrong variable name. Now put it into the container with setDefinition() - we'll call it logger.std_out_logger.

40 lines | dino_container/roar.php

```
... lines 1 - 29
$stdOutLoggerDefinition = new Definition('Monolog\Handler\StreamHandler');
$stdOutLoggerDefinition->setArguments(array('php://stdout'));
$container->setDefinition('logger.std_out_handler', $stdOutLoggerDefinition);
... lines 33 - 40
```

Yea, the file *is* getting kind long and ugly. That'll be fixed soon.

To register this handler with the logger service, we could just add it to the second constructor argument. But to prove we've got this mastered, let's use $loggerDefinition->addMethodCall() and tell it to call a pushHandler method. This takes *one* argument: the actual handler object. To pass in the service we just setup, create a new Reference and give it the service name: logger.std_out_logger:

40 lines | dino_container/roar.php

```
... lines 1 - 12
$loggerDefinition = new Definition('Monolog\Logger');
... lines 14 - 20
$loggerDefinition->addMethodCall('pushHandler', array(
    new Reference('logger.std_out_handler')
));
... lines 24 - 40
```

Open up the Logger class so we can see what's happening. Inside this class, there's a public pushHandler() method that has one argument:

```
// ...
// the Monolog\Logger class
class Logger implements LoggerInterface
{
    // ...

    public function pushHandler(HandlerInterface $handler)
    {
        // ...
    }
}
```

Our new definition code says: call pushHandler on the Logger object and pass it the service whose id is logger.std_out_logger. With any luck, log messages will dump into our file *and* get printed out to the screen.

Let's do it!

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Hey, there's one message! But inside dino.log, we see *both*. So why did only one message get printed to the screen? Actually, it's just a matter of order: we're calling the debug() method *before* pushing the second handler. If you want to fix things, just rearrange the two method calls. Now it prints both.

In the Symfony Framework, or Drupal or anything else that uses Symfony's container, *every* service starts as a humble Definition object. This object lets you tweak your future services in a bunch of other ways too, including things like adding tags, a thing called a configurator, creating objects through factories and a few other ways. To find out a lot more, head over to Symfony.com: there's a *massive* section in the Components area. You can basically earn you degree in DependencyInjection.

Now, in the Symfony framework, we work with Yaml files instead of Definition objects. So, how does all of this translate to Yaml?

Chapter 4: Yaml for Service Definitions

Creating service definitions in PHP is just *one* way to do things: you can configure this same stuff purely in Yaml or XML. And since Symfony uses Yaml files, let's use them too.

Up top, create a $loader object - set it to new YamlFileLoader() from the DependencyInjection component. This takes two arguments: the $container and a new FileLocator. That's a really simple object that says: "Yo, look for files inside of a config directory". To make it read a Yaml file, call load() and point it at a new file called services.yml.

44 lines | dino_container/roar.php

```
... lines 1 - 11
$container = new ContainerBuilder();
$loader = new YamlFileLoader($container, new FileLocator(__DIR__.'/config'));
$loader->load('services.yml');
... lines 15 - 44
```

This code tells the container to go find service definitions in this file.

Now create a config/ directory and put that services.yml in there. Our goal is to move all of this Definition stuff into that Yaml file. We'll start with the logger. I'll comment out *all* of the $loggerDefinition stuff, but keep it as reference:

44 lines | dino_container/roar.php

```
... lines 1 - 15
/*
$loggerDefinition = new Definition('Monolog\Logger');
... lines 18 - 27
*/
... lines 29 - 44
```

Definitions in services.yml

The *whole* purpose of this Yaml file is to build service Definition objects. So it should be no surprise why we start with a services key. Next, since the nickname of the service is logger, put that. Indented below this is *anything* needing to configure this Definition object: to train the container on how to create the logger service.

Almost every service will at least have two parts: the class - set it to Monolog\Logger and arguments. We know we have 2 arguments. The first is the string main and the second is an array with a reference to another service. To add the first, just say main:

6 lines | dino_container/config/services.yml

```
services:
    logger:
        class: Monolog\Logger
        arguments:
            - 'main'
```

Before we put the second argument, let's just make sure things are *not* exploding so far:

```
$ php dino_container/roar.php
```

No explosion! It's printing out to the screen, but if you look in the log, it's not adding anything there - the new ones should be from 8:46.

When we say "go load services.yml", it creates a new Definition object for logger. But that logger doesn't have any handlers yet, so it's reverting to that default mode where it just dumps to the screen.

To hook up the first handler, our constructor needs a second argument. Add another line with a dash, then paste logger.stream_handler. If we did *just* this, it'll pass this in as a string. In PHP code, this is where we passed in a Reference object. To make this create a Reference, we put an @ symbol in front. I'll surround this in quotes - but you don't technically need to:

8 lines | dino_container/config/services.yml

```
services:
    logger:
        class: Monolog\Logger
        arguments:
            - 'main'
            - '@logger.stream_handler'
... lines 7 - 8
```

Try it! Woh, explosion! Argument 2 should be an array, but I'm passing an object. I was sloppy. For my second argument, I'm passing literally *one* object. But we know the second argument is an array of objects. So in Yaml, we need to surround this with square brackets to make that an array:

8 lines | dino_container/config/services.yml

```
services:
    logger:
        class: Monolog\Logger
        arguments:
            - 'main'
            - ['@logger.stream_handler']
... lines 7 - 8
```

This time, no errors!

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

The console message is gone, but the log file gets it.

The big point is that you can create Definition objects by hand, OR use a config file to do that for you. When services.yml is loaded, it *is* creating those same Definition objects.

And as you'll see in a bit, if you want to get really advanced, you'll want to understand both ways.

addMethodCall in Yaml

Next, we need to move over the addMethodCall stuff. In Yaml, add a calls key. The bummer of the calls key is that it has a funny syntax. Add a new line with a dash like arguments. We know that the method is called debug and we need to pass that method a single string argument. In Yaml, it translates to this. Inside square brackets pass debug, then another set of square brackets for the arguments. If we wanted to pass three arguments, we'd just put a comma-separated list. We'll just paste the message in as the only argument:

10 lines | dino_container/config/services.yml

```
services:
    logger:
        class: Monolog\Logger
... lines 4 - 6
        calls:
            - ['debug', ['Logger just got started!!!']]
... lines 9 - 10
```

I know that's ugly. But under the hood, that's just calling addMethodCall on the Definition and passing it debug and this arguments array. Let's go back to the terminal and try it:

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

Tail the logs, and boom! Our extra "logger has started" message is back. Now let's do the same for the other method call. It's exactly the same, except the argument is a service. Copy that name, add a new line under calls, say pushHandler, @ then the paste our handler name:

11 lines | dino_container/config/services.yml

```
services:
    logger:
... lines 3 - 6
        calls:
            - ['pushHandler', ['@logger.std_out_handler']]
            - ['debug', ['Logger just got started!!!']]
... lines 10 - 11
```

Test it out.

```
$ php dino_container/roar.php
```

Yes! Both handlers are back! And congrats! Our entire logger definition is now in Yaml. And this is a pretty complicated example - most services are just a class and arguments. Celebrate by removing the commented-out $loggerDefinition stuff.

Chapter 5: Parameters

Let's finish this up by converting both handlers to Yaml. Do the "stdout" logger first - it's easier. Under the services key, add a new entry for logger.std_out_logger and give it the class name:

15 lines | dino_container/config/services.yml

```
services:
    logger:
... lines 3 - 10
    logger.std_out_handler:
        class: Monolog\Handler\StreamHandler
... lines 13 - 15
```

Peak back - this has one argument. So add the arguments key and give it the php://stdout. Those quotes are optional, and if you want, you can put the arguments up onto one line, inside square brackets:

15 lines | dino_container/config/services.yml

```
services:
... lines 2 - 10
    logger.std_out_handler:
        class: Monolog\Handler\StreamHandler
        arguments: ['php://stdout']
... lines 14 - 15
```

And as long as this still prints to the screen, life is good:

```
$ php dino_container/roar.php
```

Perfect!

Adding a Parameter in PHP

Now let's move the *other* handler. But this one is a little trickier: its argument has a PHP expression - __DIR__. That's trouble.

But hey, ignore it for now! Copy the service name and put it into services.yml. The order of services does *not* matter. Pass it the class and give it a single argument. This will *not* work, but I'll copy the __DIR__.'/dino.log in as the argument:

That's the basic idea, but since that __DIR__ stuff is PHP code, this won't work. But the solution is really nice.

The container holds *more* than services. It *also* has a simple key-value configuration system called parameters. In PHP, to add a parameter, just say $container->setParameter() and invent a name. How about root_dir? And we'll set its value to __DIR__:

23 lines | dino_container/roar.php

```
... lines 1 - 10
$container = new ContainerBuilder();
$container->setParameter('root_dir', __DIR__);
... lines 13 - 23
```

That doesn't *do* anything, but now we can use that root_dir parameter *anywhere* else when we're building the container.

To use a parameter in Yaml, say %root_dir%:

19 lines | dino_container/config/services.yml

```
services:
... lines 2 - 10
    logger.stream_handler:
        class: Monolog\Handler\StreamHandler
        arguments: ['%root_dir%/dino.log']
... lines 14 - 19
```

With everything in Yaml, we can clean up! We don't need any Definition code at all in roar.php - just create the container, set the parameter and load the yaml file:

23 lines | dino_container/roar.php

```
... lines 1 - 10
$container = new ContainerBuilder();
$container->setParameter('root_dir', __DIR__);

$loader = new YamlFileLoader($container, new FileLocator(__DIR__.'/config'));
$loader->load('services.yml');

runApp($container);
... lines 18 - 23
```

Ok, moment of truth!

```
$ php dino_container/roar.php
$ tail dino_container/dino.log
```

It still prints! And it's still adding to our log file. And now all that service Definition code is sitting in services.yml.

Parameters in Yaml

Of course, you can also add parameters in Yaml. Add a parameters root key somewhere - order doesn't matter - and invent one called logger_start_message. Copy the string from the debug call and paste it. Now that we have a second parameter, we can grab the key and use it inside two percents:

22 lines | dino_container/config/services.yml

```
parameters:
    logger_startup_message: 'Logger just got started!!!'
... line 3
services:
    logger:
... lines 6 - 9
        calls:
... line 11
            - ['debug', ['%logger_startup_message%']]
... lines 13 - 22
```

And this still works just like before.

This last point is actually really important. Yaml files that build the container only have *three* valid root keys: services, parameters and another called imports, which just loads other files. And that makes sense. After all, a container is nothing more than a collection of services and parameters. This point will be really important later. Because in Symfony, files like config.yml violate this rule with root keys like framework and twig.

With all this hard work behind us, we're about to see one of the coolest features of the container, and the reason why it's so fast.

Chapter 6: The Container Dumper

After a container is built, you should compile it:

24 lines | dino_container/roar.php

```
... lines 1 - 10
$container = new ContainerBuilder();
... lines 12 - 16
$container->compile();
runApp($container);
... lines 19 - 24
```

This starts one final layer to the build process, which anyone can hook into to make final adjustments. For now, it's not doing anything - but it's really important inside the framework.

In a big project - parsing Yaml files and collecting all this service Definition stuff can start to take a lot of time. Our container is nice, but it's coming at a performance cost.

Let's see how much by adding some *really* basic profiling code. Up top, add a $startTime variable. And down below, figure out how much time elapsed, multiply it by 1000 to get microseconds, and while we're here, round it. And hey, let's use our container to get out the logger and debug a message about this:

29 lines | dino_container/roar.php

```
... lines 1 - 8
$start = microtime(true);
... lines 10 - 19
runApp($container);

$elapsed = round((microtime(true) - $start) * 1000);
$container->get('logger')->debug('Elapsed Time: '.$elapsed.'ms');
... lines 24 - 29
```

So let's see how long this takes:

```
$ php dino_container/roar.php
```

37ms at first, but then it settles to about 19ms after running a few times. Not bad, but this is a tiny project. Just keep that 19ms number in mind.

Caching the Container

Here's the question: can we take all of this metadata about the container and cache it somehow? Absolutely - and the way it caches is incredible.

After compiling, create a new variable called $dumper and set it to a new PhpDumper object. Pass the $container to the dumper:

33 lines | dino_container/roar.php

```
... lines 1 - 19
$container->compile();
$dumper = new PhpDumper($container);
... lines 22 - 33
```

This guy is an expert at taking that metadata and caching it to a file. To do that, use the good ol' fashioned file_put_contents - pass it some new file path - how about cached_container.php and for the contents, call $dumper->dump():

33 lines | dino_container/roar.php

```
... lines 1 - 19
$container->compile();
$dumper = new PhpDumper($container);
file_put_contents(__DIR__.'/cached_container.php', $dumper->dump());
... lines 23 - 33
```

Let's see what this does! Run the script again:

```
$ php dino_container/roar.php
```

Now the cached_container.php file pops into existence. And it's *awesome*.

The Cached Container

Oh, so many good things to see. First, notice that this dumps a PHP class that extends Container:

142 lines | dino_container/cached_container.php

```
... lines 1 - 3
use Symfony\Component\DependencyInjection\Container;
... lines 5 - 16
class ProjectServiceContainer extends Container
{
... lines 19 - 140
}
```

That's actually the same base class as the ContainerBuilder we've been working with, and *it* houses the all-important get() function that fetches out services. In other words, this ProjectServiceContainer looks and acts *just* like the $container we're using now.

Next, this has our two parameter values sitting on top. And if you call getParameter() to fetch one, it just uses this array:

142 lines | dino_container/cached_container.php

```
... lines 1 - 16
class ProjectServiceContainer extends Container
{
    private static $parameters = array(
            'root_dir' => '/Users/weaverryan/Sites/knp/knpu-repos/symfony-journey-to-center/dino_container',
            'logger_startup_message' => 'Logger just got started!!!',
        );
... lines 23 - 100
    public function getParameter($name)
    {
        $name = strtolower($name);

        if (!(isset(self::$parameters[$name]) || array_key_exists($name, self::$parameters))) {
            throw new InvalidArgumentException(sprintf('The parameter "%s" must be defined.', $name));
        }

        return self::$parameters[$name];
    }
... lines 111 - 140
}
```

And now, the most *important* thing to notice: for each of our three services, there's a concrete method that's called when we ask for that service:

142 lines | dino_container/cached_container.php

```
... lines 1 - 53
54      /**
55       * Gets the 'logger' service.
... lines 56 - 60
61       */
62      protected function getLoggerService()
63      {
64          $this->services['logger'] = $instance = new \Monolog\Logger('main', array(0 => $this->get('logger.stream_handler')));
65
66          $instance->pushHandler($this->get('logger.std_out_handler'));
67          $instance->debug('Logger just got started!!!');
68
69          return $instance;
70      }
... line 71
72      /**
73       * Gets the 'logger.std_out_handler' service.
... lines 74 - 78
79       */
80      protected function getLogger_StdOutHandlerService()
81      {
82          return $this->services['logger.std_out_handler'] = new \Monolog\Handler\StreamHandler('php://stdout');
83      }
... line 84
85      /**
86       * Gets the 'logger.stream_handler' service.
... lines 87 - 91
92       */
93      protected function getLogger_StreamHandlerService()
94      {
95          return $this->services['logger.stream_handler'] = new \Monolog\Handler\StreamHandler('/Users/weaverryan/Sites/knp/knpu-repo
96      }
... lines 97 - 142
```

Seriously, if you look at the get() function in the parent class, you'll find that calling $container->get('logger.std_out_logger') will ultimately execute this getLogger_StdOutLoggerService() method.

And these methods use the *exact* PHP code we would write to instantiate these objects directly. We pass the container Definition objects, and it dumps the raw PHP code that those represent.

This is even more incredible when you look at the getLoggerService() method:

142 lines | dino_container/cached_container.php

```
... lines 1 - 61
62      protected function getLoggerService()
63      {
64          $this->services['logger'] = $instance = new \Monolog\Logger('main', array(0 => $this->get('logger.stream_handler')));
65
66          $instance->pushHandler($this->get('logger.std_out_handler'));
67          $instance->debug('Logger just got started!!!');
68
69          return $instance;
70      }
... lines 71 - 142
```

Look closely: it creates the new Logger object, passes main and then passes an array, with a call to $this->get('logger.stream_handler') to fetch *that* service from itself - the container. The second arguments key in the Yaml file causes this.

Next, it has our two method calls: pushHandler() with $this->get('logger.std_out_logger') and then a call to debug(). Everything we put into those Definitions are dumped into a *real* PHP file that contains the raw code we would've written anyways.

So, if we use this container class directly, then fetching objects out of it could not be faster. Let's do it!

Using the Cached Container

Copy the path to the file and create a new $cachedContainer variable *way* up top before we even start with the ContainerBuilder. Our app now has two options: we can create the ContainerBuilder, load it up with the Definition config and then use it, OR, if that cached container is available, we can skip everything and just use it. After all. if we call get('logger') on it, it'll give us the exact same Logger.

So, if (!file_exists($cachedContainer)), then we *do* need to do all the building work to dump the container:

39 lines | dino_container/roar.php

```
... lines 1 - 12
require __DIR__.'/../vendor/autoload.php';

$cachedContainer = __DIR__.'/cached_container.php';
if (!file_exists($cachedContainer)) {
    $container = new ContainerBuilder();
    $container->setParameter('root_dir', __DIR__);

    $loader = new YamlFileLoader($container, new FileLocator(__DIR__ . '/config'));
    $loader->load('services.yml');

    $container->compile();
    $dumper = new PhpDumper($container);
    file_put_contents(__DIR__ . '/cached_container.php', $dumper->dump());
}
... lines 27 - 39
```

But one way or another, that file eventually exists. So if we require it, we can say $container = new \ProjectServiceContainer(), which is the class name used in the cache file:

39 lines | dino_container/roar.php

```
... lines 1 - 14
$cachedContainer = __DIR__.'/cached_container.php';
if (!file_exists($cachedContainer)) {
... lines 17 - 25
}
require $cachedContainer;
$container = new \ProjectServiceContainer();

runApp($container);
... lines 31 - 39
```

We're still passing this $container into runApp(), and even though it's technically a different object, it's not going to make *any* difference. The only thing we need to change is that runApp() is type-hinted with ContainerBuilder. Well, it turns out that what we really need is Container, which is the base class for the builder and our cached class.

So I'll change the type-hint to Container. And we can go a step further: the Container class implements an interface called ContainerInterface:

39 lines | dino_container/roar.php

```
... lines 1 - 6
use Symfony\Component\DependencyInjection\ContainerInterface;
... lines 8 - 34
function runApp(ContainerInterface $container)
{
    $container->get('logger')->info('ROOOOAR');
}
```

Ok, try out the brand new cached container!

```
$ php dino_container/roar.php
```

It works! And woh - check out that elapsed time: **4ms**, down from 19. If you delete the cached_container.php file, the next run takes 22ms because it needs to rebuild it. Then we're right back down to 4ms. This is one reason why Symfony is able to be so fast, even in big systems.

Now that you've got the *real* story of how container building works, let's see how things look inside Symfony.

Chapter 7: How Symfony Builds the Container

We rock at building containers. So now let's see how it's build inside of Symfony.

Setting up app_dev.php for Debugging

To figure things out, let's jump straight to the code, starting with the app_dev.php front controller. We're going to add some var_dump statements to core classes, and for that to actually work, we need to make a few changes here. First, instead of loading bootstrap.php.cache, require autoload.php. Second, make sure this $kernel->loadClassCache() line is commented out:

32 lines | web/app_dev.php

```
... lines 1 - 19
//$loader = require_once __DIR__.'/../app/bootstrap.php.cache';
$loader = require_once __DIR__.'/../app/autoload.php';
... lines 22 - 25
$kernel = new AppKernel('dev', true);
//$kernel->loadClassCache();
... lines 28 - 32
```

A copy of some *really* core classes in Symfony are stored in the cache directory for a little performance boost. These two changes turn that off so that if we var_dump somewhere, it'll definitely work.

Booting the Kernel

In the first journey episode, we followed this $kernel->handle() method to find out what happens between the request and response. But this method does something *else* too. Click to open it up: it lives in a core Kernel class. Inside handle(), it calls boot() on itself:

810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 11
namespace Symfony\Component\HttpKernel;
... lines 13 - 44
abstract class Kernel implements KernelInterface, TerminableInterface
{
... lines 47 - 178
    public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, $catch = true)
    {
        if (false === $this->booted) {
            $this->boot();
        }

        return $this->getHttpKernel()->handle($request, $type, $catch);
    }
... lines 187 - 808
}
```

But first, let me back up a second. Remember that the $kernel here is an instance of *our* AppKernel, and that extends this core Kernel.

The boot() method has one job: build the container. And most of the real work happens inside the initializeContainer() function:

```
810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php
... lines 1 - 551
552     protected function initializeContainer()
553     {
554         $class = $this->getContainerClass();
555         $cache = new ConfigCache($this->getCacheDir().'/'.$class.'.php', $this->debug);
... line 556
557         if (!$cache->isFresh()) {
558             $container = $this->buildContainer();
559             $container->compile();
560             $this->dumpContainer($cache, $container, $class, $this->getContainerBaseClass());
... lines 561 - 562
563         }
564
565         require_once $cache;
566
567         $this->container = new $class();
... lines 568 - 572
573     }
... lines 574 - 810
```

Hey, this looks really familiar. The container is built on line 558, and we'll look more at that function. Then its compiled and dumpContainer() writes the cached PHP container class. I'll show you - jump into the dumpContainer() function:

```
810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php
... lines 1 - 703
704     protected function dumpContainer(ConfigCache $cache, ContainerBuilder $container, $class, $baseClass)
705     {
706         // cache the container
707         $dumper = new PhpDumper($container);
... lines 708 - 712
713         $content = $dumper->dump(array('class' => $class, 'base_class' => $baseClass));
... lines 714 - 717
718         $cache->write($content, $container->getResources());
719     }
... lines 720 - 810
```

Hey, there's our PhpDumper class - it does the same thing we did by hand before.

Back in initializeContainer(), it finishes off by requiring the cached container file and creating a new instance:

```
810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php
... lines 1 - 551
552     protected function initializeContainer()
553     {
554         $class = $this->getContainerClass();
555         $cache = new ConfigCache($this->getCacheDir().'/'.$class.'.php', $this->debug);
... lines 556 - 564
565         require_once $cache;
566
567         $this->container = new $class();
568         $this->container->set('kernel', $this);
... lines 569 - 572
573     }
... lines 574 - 810
```

So Symfony creates and dumps the container just like we did.

kernel. and Environment Parameters

There are a lot of little steps that go into building the container, so I'll jump us to the important parts. Go into buildContainer() and look at the line that calls $this->getContainerBuilder():

```
810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php
... lines 1 - 628
629     protected function buildContainer()
630     {
... lines 631 - 640
641         $container = $this->getContainerBuilder();
... line 642
643         $this->prepareContainer($container);
644
645         if (null !== $cont = $this->registerContainerConfiguration($this->getContainerLoader($container))) {
646             $container->merge($cont);
647         }
... lines 648 - 650
651         return $container;
... lines 652 - 810
```

If we jump to that function, we can see the line that actually creates the new ContainerBuilder object - just like we did before:

```
810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php
... lines 1 - 684
685     protected function getContainerBuilder()
686     {
687         $container = new ContainerBuilder(new ParameterBag($this->getKernelParameters()));
... lines 688 - 692
693         return $container;
694     }
... lines 695 - 810
```

The only addition is that it passes it some parameters to start out. These are in getKernelParameters():

810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 579
    protected function getKernelParameters()
    {
        $bundles = array();
        foreach ($this->bundles as $name => $bundle) {
            $bundles[$name] = get_class($bundle);
        }

        return array_merge(
            array(
                'kernel.root_dir' => $this->rootDir,
                'kernel.environment' => $this->environment,
                'kernel.debug' => $this->debug,
                'kernel.name' => $this->name,
                'kernel.cache_dir' => $this->getCacheDir(),
                'kernel.logs_dir' => $this->getLogDir(),
                'kernel.bundles' => $bundles,
                'kernel.charset' => $this->getCharset(),
                'kernel.container_class' => $this->getContainerClass(),
            ),
            $this->getEnvParameters()
        );
    }
... lines 602 - 810
```

You probably recognize some of these - like kernel.root_dir, and now you know where they come from. It also calls getEnvParameters():

810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 609
    protected function getEnvParameters()
    {
        $parameters = array();
        foreach ($_SERVER as $key => $value) {
            if (0 === strpos($key, 'SYMFONY__')) {
                $parameters[strtolower(str_replace('__', '.', substr($key, 9)))] = $value;
            }
        }

        return $parameters;
    }
... lines 621 - 810
```

You may not know about this feature: if you set an environment variable that starts with SYMFONY__, that prefix is stripped and its added as a parameter automatically. That magic comes from right here

The Cached Container

Back in buildContainer(), let's var_dump() the $container so far to see what we've got:

811 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 628
    protected function buildContainer()
    {
... lines 631 - 640
        $container = $this->getContainerBuilder();
        var_dump($container);die;
... lines 643 - 652
    }
... lines 654 - 811
```

Ok, refresh! Hmm, it didn't hit my code. Why? Well, the container might already be cached, so it's not going through the building process. To force a build, you can delete the cached container file. But before you do that, I'll look inside - it's located at app/cache/dev/appDevDebugProjectContainer.php:

4243 lines | app/cache/dev/appDevDebugProjectContainer.php

```
... lines 1 - 16
class appDevDebugProjectContainer extends Container
{
    private static $parameters = array(
            'kernel.root_dir' => '/Users/weaverryan/Sites/knp/knpu-repos/symfony-journey-to-center/app',
            'kernel.environment' => 'dev',
... lines 22 - 640
        );
... lines 642 - 3820
    /**
     * Gets the 'user_agent_subscriber' service.
... lines 3823 - 3828
    protected function getUserAgentSubscriberService()
    {
        return $this->services['user_agent_subscriber'] = new \AppBundle\EventListener\UserAgentSubscriber($this->get('logger'));
    }
... lines 3833 - 4241
}
```

It's a lot bigger and has a different class name, but this is just like our cached container: it has all the parameters on top, then a bunch of methods to create the services. Now go delete that file and refresh.

```
$ rm app/cache/dev/appDevDebugProjectContainer.php
```

Great: *now* we see the dumped container. I want you to notice a few things. First, there are *no* service definitions at all. But we do have the 9 parameters. And that's it - the container is basically empty so far.

Loading the Yaml Files

To fill it with services, we'll load a Yaml file that'll supply some service definitions. Back in buildContainer(), this happens when the registerContainerConfiguration() method is called:

810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 628
    protected function buildContainer()
    {
... lines 631 - 640
        $container = $this->getContainerBuilder();
... line 642
        $this->prepareContainer($container);

        if (null !== $cont = $this->registerContainerConfiguration($this->getContainerLoader($container))) {
            $container->merge($cont);
        }
... lines 648 - 650
        return $container;
    }
... lines 653 - 810
```

I did skip a few things - but no worries, we'll cover them in a minute. This function actually lives in our AppKernel:

40 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
... lines 8 - 34
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
    }
}
```

The LoaderInterface argument is an object that's a lot like the YamlFileLoader that we created manually in roar.php. This loader can *also* read other formats, like XML. But beyond that, it's the same: you create a loader and then pass it a file full of services.

When Symfony boots, it only loads *one* configuration file - config_dev.yml if you're in the dev environment:

49 lines | app/config/config_dev.yml

```
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
... lines 13 - 49
```

I know you've looked at that file before, but two really important things are hiding here. I mentioned earlier that these configuration files have only three valid root keys: services (of course), parameters (of course) and imports - to load other files. But in this file - and almost every file in this directory - you see mostly other stuff, like framework, webprofiler and monolog. Having these root keys *should* be illegal. But in fact, they're the secret to how almost every service is added to the container. We'll explore those next - so ignore them for now.

The other important thing is that config_dev.yml imports config.yml:

74 lines | app/config/config.yml

```
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: services.yml }
... lines 5 - 74
```

And config.yml loads parameters.yml, security.yml and services.yml. Every file in the app/config directory - except the routing files - are being loaded by the container in order to provide services. In other words, all of these files have the exact same purpose as the services.yml file we played with before inside of dino_container.

The weird part is that none of these files have any services in them, except for one: services.yml:

12 lines | app/config/services.yml

```
... lines 1 - 5
services:
    user_agent_subscriber:
        class: AppBundle\EventListener\UserAgentSubscriber
        arguments: ["@logger"]
        tags:
            - { name: kernel.event_subscriber }
```

It holds our user_agent_subscriber service from episode 1. This gives us one service definition and parameters.yml adds a few parameters.

So after the registerContainerConfiguration() line is done, we've gone from zero services to only 1. Let's dump to prove it - $container->getDefinitions().

811 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 628
    protected function buildContainer()
    {
... lines 631 - 644
        if (null !== $cont = $this->registerContainerConfiguration($this->getContainerLoader($container))) {
            $container->merge($cont);
        }
        var_dump($container->getDefinitions());die;
... lines 649 - 652
    }
... lines 654 - 811
```

Refresh! Yep, there's just our *one* user_agent_subscriber service. We can dump the parameters too - $container->getParameterBag()->all():

811 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 628
    protected function buildContainer()
    {
... lines 631 - 644
        if (null !== $cont = $this->registerContainerConfiguration($this->getContainerLoader($container))) {
            $container->merge($cont);
        }
        var_dump($container->getParameterBag()->all());die;
... lines 649 - 652
    }
... lines 654 - 811
```

This dumps out the kernel parameters from earlier plus the stuff from parameters.yml.

So even though the container is still almost empty, we've nearly reached the end. This empty-ish container is returned to initializeContainer() where it's compiled and then dumped:

810 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 551
    protected function initializeContainer()
    {
... lines 554 - 557
            $container = $this->buildContainer();
            $container->compile();
            $this->dumpContainer($cache, $container, $class, $this->getContainerBaseClass());
... lines 561 - 572
    }
... lines 574 - 810
```

Before compiling, we only have 1 service. But we know from running container:debug that there are a *lot* of services when things finish. The secret is in the compile() function, which does two special things: process dependency injection extensions and run compiler passes. Those are up next.

Chapter 8: Dependency Injection Extensions

These Yaml files *should* only have keys for services, parameters and imports. What if I just make something up, like journey and put a dino_count of 10 under it:

77 lines | app/config/config.yml

```
... lines 1 - 5
journey:
    dino_count: 10
... lines 8 - 77
```

When we refresh, we get a *huge* error!

```
There is no extension able to load the configuration for "journey".
```

And it says it found valid namespaces for framework, security, twig, monolog, blah blah blah. Hey, *those* are the root keys that we have in our config files. So what makes journey invalid but framework valid? And what does framework do anyways?

Take out that journey code.

Registering of Extension Classes

The answer lives in the bundle classes. Open up AppBundle:

10 lines | src/AppBundle/AppBundle.php

```
... lines 1 - 4
use Symfony\Component\HttpKernel\Bundle\Bundle;

class AppBundle extends Bundle
{

}
```

This is empty, but it extends Symfony's base Bundle class. The key method is getContainerExtension():

212 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Bundle/Bundle.php

```
... lines 1 - 28
abstract class Bundle extends ContainerAware implements BundleInterface
{
... lines 31 - 71
    public function getContainerExtension()
    {
        if (null === $this->extension) {
            $class = $this->getContainerExtensionClass();
            if (class_exists($class)) {
                $extension = new $class();
... lines 78 - 88
                $this->extension = $extension;
            } else {
                $this->extension = false;
            }
        }

        if ($this->extension) {
            return $this->extension;
        }
    }
... lines 99 - 212
```

When Symfony boots, it calls this method on each bundle looking for something called an Extension. This calls getContainerExtensionClass() and checks to see if that class exists. Move down to that method:

212 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Bundle/Bundle.php

```
... lines 1 - 204
    protected function getContainerExtensionClass()
    {
        $basename = preg_replace('/Bundle$/', '', $this->getName());

        return $this->getNamespace().'\\DependencyInjection\\'.$basename.'Extension';
    }
... lines 211 - 212
```

Ah, and *here's* the magic. To find this "extension" class, it looks for a DependencyInjection directory and a class with the same name as the bundle, except replacing Bundle with Extension. For example, for AppBundle, it's looking for a DependencyInjection\AppExtension class. We don't have that.

Open up the TwigBundle class and double-click the directory tree at the top to move PhpStorm here. TwigBundle *does* have a DependencyInjection directory and a TwigExtension inside:

157 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 25
class TwigExtension extends Extension
{
... lines 28 - 33
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 36 - 130
    }
... lines 132 - 155
}
```

So because this is here, it's automatically registered with the container. We may not know what an extension does yet, but

we know how it's all setup.

Registering Twig Globals

Forget about extensions for a second and let me tell you about a totally unrelated feature. If you want to add a global variable to Twig, one way to do that is under the twig config. Just add globals, then set something up. I'll say twitter_username: weaverryan:

76 lines | app/config/config.yml

```
... lines 1 - 28
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
    globals:
        twitter_username: weaverryan
... lines 34 - 76
```

And just by doing that, we could open up any Twig template and have access to a twitter_username variable. My question is: how does that work?

The Extension load() Method

To answer that, look back at TwigExtension. The first secret is that when we call compile() on the container, this load() method is called. In fact the load() method is called on *every* extension that's registered with Symfony: so every class that follows the DependencyInjection\Extension naming-convention.

Let's dump the $configs variable, because I don't know what that is yet:

158 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 33
    public function load(array $configs, ContainerBuilder $container)
    {
        var_dump($configs);die;
... lines 37 - 131
    }
... lines 133 - 158
```

Go back and refresh! Ok: it dumps an array with the twig configuration. Whatever we have in config.yml under twig is getting passed to TwigExtension:

In fact, *that's* the rule. The fact that we have a key called framework means that this config will be passed to a class called FrameworkExtension. If you want to see how this config is used, look there. With the assetic key, that's passed to AsseticExtension. These extension classes have a getAlias() method in them, and that returns a lower-cased version of the class name without the word Extension.

Extensions Load Services

These extensions have two jobs. First, they add service definitions to the container. Because after all, the main reason for adding a bundle is to add services to your container.

The way it does this is just like our roar.php file, except it loads an XML file instead of Yaml:

158 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 33
    public function load(array $configs, ContainerBuilder $container)
    {
... line 36
        $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('twig.xml');
... lines 39 - 131
    }
... lines 133 - 158
```

Let's open up that Resources/config/twig.xml file:

144 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/Resources/config/twig.xml

```
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd">

    <parameters>
        <parameter key="twig.class">Twig_Environment</parameter>
... lines 9 - 28
    </parameters>

    <services>
        <service id="twig" class="%twig.class%">
            <argument type="service" id="twig.loader" />
            <argument>%twig.options%</argument>
            <call method="addGlobal">
                <argument>app</argument>
                <argument type="service" id="templating.globals" />
            </call>
        </service>

        <service id="twig.cache_warmer" class="%twig.cache_warmer.class%" public="false">
            <tag name="kernel.cache_warmer" />
            <argument type="service" id="service_container" />
            <argument type="service" id="templating.finder" />
        </service>
... lines 46 - 141
    </services>
</container>
```

If you ever wondered where the twig service comes from, it's right here! You can see it in container:debug:

```
$ php app/console container:debug twig
```

So the first job of an extension class is to add services, which it always does by loading one or more XML files.

Extensions Configuration

The second job is to read our configuration array and use that information to mutate the service definitions. We'll see code that does this shortly.

Most extensions will have two lines near the top that call getConfiguration() and processConfiguration:

158 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 33
34      public function load(array $configs, ContainerBuilder $container)
35      {
... lines 36 - 52
53          $configuration = $this->getConfiguration($configs, $container);
54
55          $config = $this->processConfiguration($configuration, $configs);
... lines 56 - 131
132     }
... lines 133 - 158
```

Next to every extension class, you'll find a class called Configuration:

201 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/Configuration.php

```
... lines 1 - 22
23  class Configuration implements ConfigurationInterface
24  {
... lines 25 - 199
200 }
```

Watch out, a meteor! Oh, never mind, it's just the awesome fact that if I mess up some configuration - like globals as globalsss in Yaml, we'll get a really nice error. That doesn't happen by accident, that system *evolved* these Configuration classes to make that happen.

This is probably one of the more bizarre classes you'll see: it builds a tree of valid configuration that can be used under this key. It adds a globals section, which says that the children are an array. It even has some stuff to validate and normalize what we put here:

201 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/Configuration.php

```
... lines 1 - 104
    private function addGlobalsSection(ArrayNodeDefinition $rootNode)
    {
        $rootNode
            ->fixXmlConfig('global')
            ->children()
                ->arrayNode('globals')
                    ->normalizeKeys(false)
                    ->useAttributeAsKey('key')
                    ->example(array('foo' => '"@bar"', 'pi' => 3.14))
                    ->prototype('array')
... lines 115 - 124
                        ->beforeNormalization()
                            ->ifTrue(function ($v) {
                                if (is_array($v)) {
                                    $keys = array_keys($v);
                                    sort($keys);

                                    return $keys !== array('id', 'type') && $keys !== array('value');
                                }

                                return true;
                            })
                            ->then(function ($v) { return array('value' => $v); })
                        ->end()
... lines 138 - 147
                    ->end()
                ->end()
            ->end()
        ;
    }
... lines 153 - 201
```

These Configuration classes are tough to write, but pretty easy to read. And if you can't get something to configure correctly, opening up the right Configuration class might give you a hint.

Back in TwigExtension, let's dump $config after calling processConfiguration():

158 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 33
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 36 - 51
        $configuration = $this->getConfiguration($configs, $container);

        $config = $this->processConfiguration($configuration, $configs);
        var_dump($config);die;
... lines 56 - 131
    }
... lines 133 - 158
```

This dumps out a nice, normalized and validated version of our config, including keys we didn't have, with their default values.

Extensions Mutate Definitions

So finally, how is the globals key used? Scroll down to around line 90:

157 lines | vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/DependencyInjection/TwigExtension.php

```
... lines 1 - 33
34      public function load(array $configs, ContainerBuilder $container)
35      {
... lines 36 - 86
87          if (!empty($config['globals'])) {
88              $def = $container->getDefinition('twig');
89              foreach ($config['globals'] as $key => $global) {
... lines 90 - 92
93                  $def->addMethodCall('addGlobal', array($key, $global['value']));
... line 94
95              }
96          }
... lines 97 - 130
131     }
... lines 132 - 157
```

For most people, this code will look weird. But not us! If there are globals, it gets the twig Definition back *out* of the ContainerBuilder. This definition was added when it loaded twig.xml, and now we're going to tweak it. Just focus on the second part of the if: it calls $def->addMethodCall() and passes it addGlobal and two arguments: our key from the config, and the value - weaverryan in this case.

If you read the Twig documentation, it tells you that if you want to add a global variable, you can call addGlobal on the Twig_Environment object. And that's exactly what this does. This type of stuff is *super* typical for extensions.

If you refresh without any debug code, we'll get a working page again. Now open up the cached container - app/cache/dev/appDevDebugProjectContainer.php and find the method that creates the twig service - getTwigService(). Make sure you spell that correctly:

4244 lines | app/cache/dev/appDevDebugProjectContainer.php

```
... lines 1 - 3703
3704    protected function getTwigService()
3705    {
3706        $this->services['twig'] = $instance = new \Twig_Environment($this->get('twig.loader'), array('debug' => true, 'strict_variables' =>
... lines 3707 - 3725
3726        $instance->addGlobal('twitter_username', 'weaverryan');
3727
3728        return $instance;
3729    }
... lines 3730 - 4244
```

Near the bottom, we see it: $instance->addGlobal('twitter_username', 'weaverryan'). We passed in simple configuration, TwigExtension used that to mutate the twig Definition, and ultimately the dumped container is updated.

That's the power of the dependency injection extensions, and if it makes even a bit of sense, you're awesome.

Our Configuration Wins

Oh, and one more cool note. If I added a twig service to config.yml, would it override the one from TwigBundle? Actually yes: even though the extensions are called after loading these files, any parameters or services we add here win.

Chapter 9: Compiler Passes

By the time we got to this step in the Kernel, our configuration files have been loaded, but this gives us just *one* service definition:

811 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Kernel.php

```
... lines 1 - 551
552     protected function initializeContainer()
553     {
... lines 554 - 556
557         if (!$cache->isFresh()) {
558             $container = $this->buildContainer();
559             var_dump($container->getDefinitions());die;
560             $container->compile();
561             $this->dumpContainer($cache, $container, $class, $this->getContainerBaseClass());
562 
563             $fresh = false;
564         }
... lines 565 - 573
574     }
... lines 575 - 811
```

So every other service must be added inside compile(). And that's true!

Calling compile() executes a group of functions called compiler passes. In fact, there's one called MergeExtensionConfigurationPass, and it's responsible for the Extension system we just looked at:

60 lines | vendor/symfony/symfony/src/Symfony/Component/DependencyInjection/Compiler/MergeExtensionConfigurationPass.php

```
... lines 1 - 21
22 class MergeExtensionConfigurationPass implements CompilerPassInterface
23 {
... lines 24 - 26
27     public function process(ContainerBuilder $container)
28     {
... lines 29 - 38
39         foreach ($container->getExtensions() as $name => $extension) {
... lines 40 - 49
50             $extension->load($config, $tmpContainer);
... lines 51 - 52
53         }
... lines 54 - 57
58     }
59 }
```

It loops over the extension objects and calls load() on each one. This is where most of the services come from.

But there's a bunch of other compiler passes, and most do small things. They're usually registered inside your bundle class - FrameworkBundle is a great example:

100 lines | vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/FrameworkBundle.php

```
... lines 1 - 45
class FrameworkBundle extends Bundle
{
... lines 48 - 64
    public function build(ContainerBuilder $container)
    {
... lines 67 - 72
        $container->addCompilerPass(new RoutingResolverPass());
        $container->addCompilerPass(new ProfilerPass());
... lines 75 - 76
        $container->addCompilerPass(new RegisterListenersPass(), PassConfig::TYPE_BEFORE_REMOVING);
        $container->addCompilerPass(new TemplatingPass());
... lines 79 - 97
    }
}
```

The build() method of every bundle is called early, and is used almost entirely just to add compiler passes. So what's the point of compiler passes? Why not just do any container modifications in the extension class?

The special thing about a compiler pass is that when it's called, the entire container has been built. So it's perfect when you need to tweak the container, but only once *all* of the service definitions are loaded.

Compiler Pass and Tags

Let's see an example. In our services.yml, we have one service, and it's an event subscriber. To tell Symfony this is an event subscriber, we had to add the tag: kernel.event_subscriber:

12 lines | app/config/services.yml

```
... lines 1 - 5
services:
    user_agent_subscriber:
        class: AppBundle\EventListener\UserAgentSubscriber
        arguments: ["@logger"]
        tags:
            - { name: kernel.event_subscriber }
```

So how does that work?

It's a compiler pass! And you can see it registered in FrameworkBundle, it's RegisterListenerPass:

107 lines | vendor/symfony/symfony/src/Symfony/Component/EventDispatcher/DependencyInjection/RegisterListenersPass.php

```
... lines 1 - 19
class RegisterListenersPass implements CompilerPassInterface
{
... lines 22 - 43
    public function __construct($dispatcherService = 'event_dispatcher', $listenerTag = 'kernel.event_listener', $subscriberTag = 'kern
    {
        $this->dispatcherService = $dispatcherService;
        $this->listenerTag = $listenerTag;
        $this->subscriberTag = $subscriberTag;
    }
... lines 50 - 105
}
```

The subscriberTag property is: kernel.event_subscriber. Near the bottom, it calls $container->findTaggedServiceIds() and passes it that:

107 lines | vendor/symfony/symfony/src/Symfony/Component/EventDispatcher/DependencyInjection/RegisterListenersPass.php

```
... lines 1 - 87
88          foreach ($container->findTaggedServiceIds($this->subscriberTag) as $id => $attributes) {
89              $def = $container->getDefinition($id);
... lines 90 - 94
95              $class = $def->getClass();
... lines 96 - 102
103             $definition->addMethodCall('addSubscriberService', array($id, $class));
104         }
... lines 105 - 107
```

It's saying: give me *all* services tagged with kernel.event_subscriber. The $definition variable at the bottom is the Definition object for the event_dispatcher. And we use it to add a method call for addSubscriberService and pass it the service id and the class.

Let's go see this in the cached container. Refresh to get it back, then search for user_agent_subscriber:

4244 lines | app/cache/dev/appDevDebugProjectContainer.php

```
... lines 1 - 1126
1127    protected function getDebug_EventDispatcherService()
1128    {
1129        $this->services['debug.event_dispatcher'] = $instance = new \Symfony\Component\HttpKernel\Debug\TraceableEventDispatche
... lines 1130 - 1136
1137        $instance->addSubscriberService('user_agent_subscriber', 'AppBundle\\EventListener\\UserAgentSubscriber');
... lines 1138 - 1160
1161        return $instance;
1162    }
... lines 1163 - 4244
```

There it is! It's calling the addSubscriberService method and passing the service id and class.

This is one of the most common jobs for a compiler pass. For example, there's another tag called form.type and this FormPass looks for all services tagged with that and does some container tweaking.

And there's a bunch more: like the compiler pass that checks for circular references. If service A depends on service B, which depends on service C, which depends on service A, you'll get a really clear exception. Then there are other passes which make micro-optimizations to speed the container up even more.

Creating a Compiler Pass

Most of the time, you won't need to create a compiler pass - you just need to understand how they work. But, we're diving deep, so let's make one! In AppBundle create a new DependencyInjection directory and inside of there a Compiler directory. I don't have to put it here, but this follows the core standard.

In here, create a new class called EarlyLoggingMessagePass. Remember how we logged a message as soon as the logger was created? We're going to do that again.

Compiler classes are pretty easy - just implement CompilerPassInterface and add the one method: process():

23 lines | src/AppBundle/DependencyInjection/Compiler/EarlyLoggingMessagePass.php

```
... lines 1 - 2
namespace AppBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class EarlyLoggingMessagePass implements CompilerPassInterface
{
... lines 10 - 16
    public function process(ContainerBuilder $container)
    {
... lines 19 - 20
    }

}
```

Now we should feel really comfortable: that's a ContainerBuilder object and we know all about him. It also has every service already defined inside. So we can say: $definition = $container->findDefinition('logger'). Now just add $definition->addMethodCall() and pass it debug for the method, and an array with a single argument: Logger CREATED:

23 lines | src/AppBundle/DependencyInjection/Compiler/EarlyLoggingMessagePass.php

```
... lines 1 - 16
    public function process(ContainerBuilder $container)
    {
        $definition = $container->findDefinition('logger');
        $definition->addMethodCall('debug', array('Logger CREATED!'));
    }
... lines 22 - 23
```

And that's a functional compiler pass.

You can register this by overriding the build() method in AppBundle and adding it there. But that's too easy.

Instead, go to AppKernel and override buildContainer(). Call the parent method, then add $container->addCompilerPass() and pass it a new EarlyLoggingMessagePass. And don't forget to return the $container:

48 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
... lines 8 - 39
    protected function buildContainer()
    {
        $containerBuilder = parent::buildContainer();
        $containerBuilder->addCompilerPass(new \AppBundle\DependencyInjection\Compiler\EarlyLoggingMessagePass());

        return $containerBuilder;
    }
}
```

Ok, let's try it! Refresh! Click into the profiler then go to the logs tab. Under debug, there's the message! First on the list.

Phew! So you're now a master. The Container is all about Definition objects, which are populated from Yaml and XML files and then updated later in the dependency injection extension classes. If you're following this, go dive into the FrameworkBundle and see where the *real* core services come from And congrats, because now, you're a dependency-injection-asaurus!

Ok guys, seeya next time!