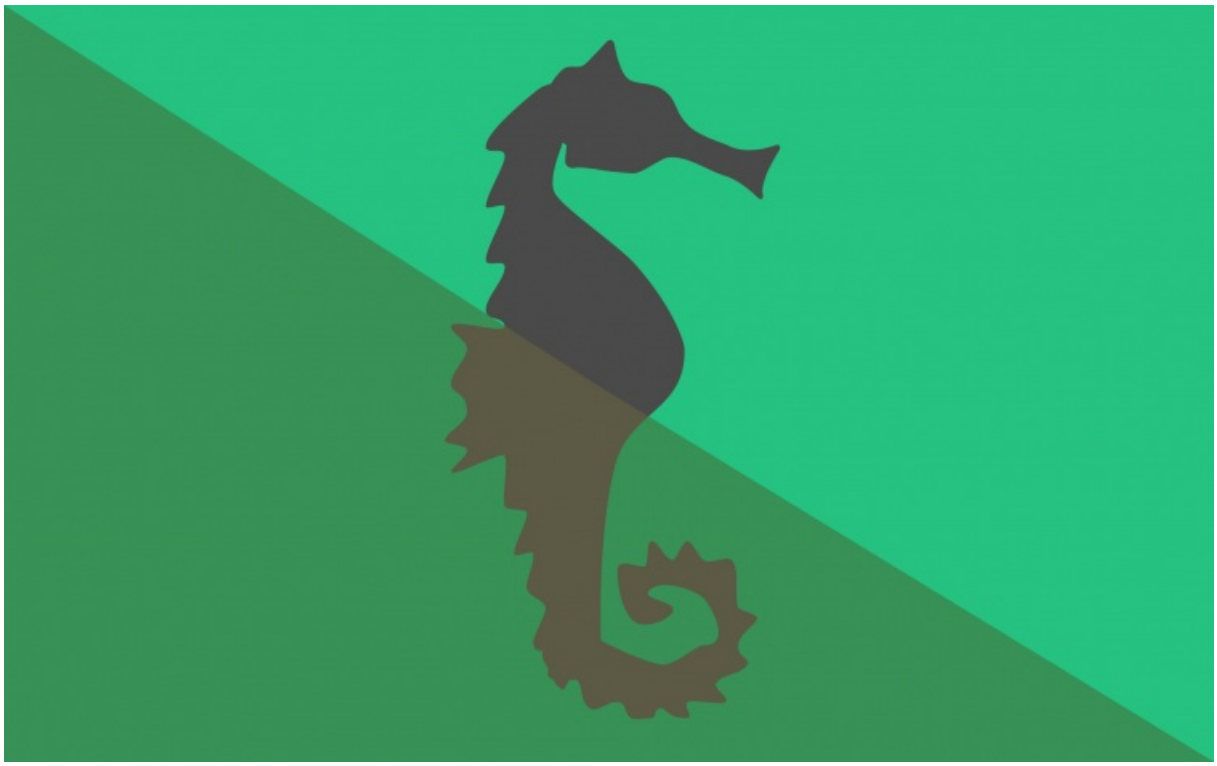
Symfony 3: Level up with Services and the Container



With <3 from SymfonyCasts

Chapter 1: Create Service

Hey, hey, friends! Back for more!? You should be feeling *pretty* good about yourself already - but you're about to feel like a kid in a very nerdy, borderline embarrassing candy store. In this course, we *fully* uncover services: those little useful objects that literally do *everything* in Symfony.

Now as always, you *should* code along with me - otherwise we'll find out and it's kinda company policy to mail you a big package full of glitter. But if you code along - and we meet in person, well, there's a cookie with your name on it.

So find the "Download" button on any of the tutorial pages, download the code, unzip it and move into the start directory. As always, I already have the start directory code here, so I'll skip to the last step: open up a new terminal and launch the built-in PHP web server with:

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

In your case, open up the README.md file in the start directory - it has a few extra instructions.

Services: Doers of Good

In the last courses, if I repeated anything too many times, it was this: Refresh! But second would be that Symfony is basically just a big container of useful objects called services... and *everything* that happens is *actually* done by one of these.

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 43
    public function listAction()
    {
... lines 46 - 50
        return $this->render('genus/list.html.twig', [
            'genuses' => $genuses
        ]);
    }
... lines 55 - 123
}
```

For example, the render() function - it's the key to rendering templates, right? No - it's a fraud! Open up Symfony's base Controller:

398 lines | vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

```
... lines 1 - 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);
        }

        if (!$this->container->has('twig')) {
            throw new \LogicException('You can not use the "render" method if the Templating Component or the Twig Bundle are not av
        }

        if (null === $response) {
            $response = new Response();
        }

        $response->setContent($this->container->get('twig')->render($view, $parameters));

        return $response;
    }
... lines 204 - 396
}
```

it doesn't do *any* work: it just finds the templating service and tells *it* to do all the work. Ah, management.

This means that Symfony doesn't render templates: The *templating* service renders templates. To get a big list of birthday wishes, I mean services, run:

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

That's a lot of firepower at our fingertips.

We *also* found out that we can control these services in app/config/config.yml:

72 lines | app/config/config.yml

```
... lines 1 - 36
# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
    number_format:
        thousands_separator: ','
... lines 43 - 72
```

New Goal: Service Architecture

But now we have a new, daring goal: adding our *own* services to the container. It turns out, learning to do this will unlock almost *everything* else in Symfony.

Open up GenusController and look at showAction():

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 58
    public function showAction($genusName)
    {
... lines 61 - 72
        // todo - add the caching back later
        /*
        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
        $key = md5($funFact);
        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 86 - 97
    }
... lines 99 - 123
}
```

We *used* to have about 15 lines of code that parsed the $funFact through Markdown and then cached it. I want that back. But, but but! I don't want to have these 15 lines of code live here, in my controller.

Why not? Three reasons:

1. I can't re-use this. If I need to do parse some markdown somewhere else... well, I could copy-and-paste? But then, how would I sleep at night?

2. It's not instantly clear *what* these 15 lines do: I have to actually take time and read them to find out. If you have a lot of chunks like this, suddenly *nobody* knows what's going on in your app. You know what I'm talking about?

3. If you want to unit test this code... well, you can't. To unit test something, it needs to live in its own, isolated, focused class.

Well hey, that's a great idea - let's move this *outside* of our controller and solve all three problems at once... *plus* impress our developer friends with our sweet code organizational skills.

Chapter 2: Creating a Service Class

Ready to move a chunk of code *out* of the controller? Well good for you.

Step 1: create a new PHP class. In AppBundle, I'll create a new directory called Service - but that could be called *anything*. Inside, add a new PHP class called MarkdownTransformer:

12 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 2
namespace AppBundle\Service;

class MarkdownTransformer
{
... lines 7 - 10
}
```

If you're keeping score at home, that could *also* be called anything.

Start this with one public function parse() with a $str argument:

12 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
    public function parse($str)
    {
        return strtoupper($str);
    }
}
```

Eventually, this will do *all* the dirty work of markdown parsing and caching. But for now... keep it simple and return strtoupper($str). But use your imagination - pretend like it totally *is* awesome and is parsing our markdown. In fact, it's so awesome that we want to use it in our controller. How?

Find GenusController. First, create a new object with $transformer = new MarkdownTransformer():

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

```
... lines 1 - 13
14  class GenusController extends Controller
15  {
... lines 16 - 58
59      public function showAction($genusName)
60      {
61          $em = $this->getDoctrine()->getManager();
62
63          $genus = $em->getRepository('AppBundle:Genus')
64              ->findOneBy(['name' => $genusName]);
65
66          if (!$genus) {
67              throw $this->createNotFoundException('genus not found');
68          }
69
70          $markdownParser = new MarkdownTransformer();
... lines 71 - 97
98      }
... lines 99 - 123
124 }
```

Noooothing special here: the new method is *purposefully* not static, and this means we need to instantiate the object first. Next, add $funFact = $transformer->parse() and pass $genus->getFunFact():

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

```
... lines 1 - 69
70          $markdownParser = new MarkdownTransformer();
71          $funFact = $markdownParser->parse($genus->getFunFact());
... lines 72 - 125
```

And that's it! If you're feeling massively underwhelmed... you're right where I want you! I want this to be boring and easy - there are fireworks and exciting stuff later.

Finish this by passing $funFact into the template so we can render the parsed version:

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

```
... lines 1 - 13
14  class GenusController extends Controller
15  {
... lines 16 - 58
59      public function showAction($genusName)
60      {
... lines 61 - 69
70          $markdownParser = new MarkdownTransformer();
71          $funFact = $markdownParser->parse($genus->getFunFact());
... lines 72 - 92
93          return $this->render('genus/show.html.twig', array(
... line 94
95              'funFact' => $funFact,
... line 96
97          ));
98      }
... lines 99 - 123
124 }
```

Then, open the template and replace genus.funFact with just funFact:

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

```
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>

    <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>
... lines 18 - 19
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 25 - 42
```

Try it out: open up localhost:8000/genus - then click one of the genuses. Yes! The fun fact is *screaming* at us in upper case.

So believe it or not: you just saw one of the most important and commonly-confusing object-oriented strategies that exist anywhere... in any language! And it's this: you should take chunks of code that do things and move them into an outside function in an outside class. That's it.

Oh, and guess what? MarkdownTransformer is a *service*. Because remember, a service is just a class that does work for us. And when you isolate a *lot* of your code into these service classes, you start to build what's called a "service-oriented architecture". OooOOoooOOOo. That basically means that instead of having *all* of your code in big controllers, you organize them into nice little services that each do one job.

Of course, the MarkdownTransformer service isn't *actually* transforming... any... markdown - so let's fix that.

Chapter 3: The Dreaded Dependency Injection

The MarkdownTransformer will do two things: parse markdown and eventually cache it. Let's start with the first.

Open up GenusController and copy the code that originally parsed the text through markdown. Now... paste this into the parse function, return that and update the variable to $str. Wow, that was easy:

13 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
    public function parse($str)
    {
        return $this->get('markdown.parser')
            ->transform($str);
    }
}
```

Go back and refresh. It *explodes*!

> Attempted to call an undefined method named "get" on MarkdownTransformer

Forget about Symfony: this makes sense. The class we just created does *not* have a get() function and it doesn't *extend* anything that would give this to us.

In a controller, we *do* have this function. But more importantly, we have access to the container, either via that shortcut method or by saying $this->container. From there we can fetch *any* service by calling ->get() on it.

But that's special to the controller: as soon as you're *not* in a controller - like MarkdownTransformer - you *don't* have access to the container, its services... or *anything*.

The Dependency Injection Flow

So here's the quadrillion bitcoin question: how can we *get* access to the container inside MarkdownTransformer? But wait, we don't *really* need the *whole* container: all we *really* need is the markdown parser object. So a better question is: how can we get access to the *markdown parser* object inside MarkdownTransformer?

The answer is probably the scariest word that was ever created for such a simple idea: dependency injection. Ah, ah, ah. I think someone invented that word *just* to be a jerk... especially because it's *so* simple...

Here's how it goes: whenever you're inside of a class and you need access to an object that you don't have - like the markdown parser - add a public function __construct() and add the object you need as an argument:

20 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
... lines 7 - 8
    public function __construct($markdownParser)
    {
... line 11
    }
... lines 13 - 18
}
```

Next create a private property and in the constructor, assign that to the object: $this->markdownParser = $markdownParser:

20 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
    private $markdownParser;

    public function __construct($markdownParser)
    {
        $this->markdownParser = $markdownParser;
    }
... lines 13 - 18
}
```

Now that the markdown parser is set on the property, use it in parse(): get rid of $this->get() and just use $this->markdownParser:

20 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
... lines 7 - 13
    public function parse($str)
    {
        return $this->markdownParser
            ->transform($str);
    }
}
```

We're done! Well, done with *this* class. You see: *whoever* instantiates our MarkdownTransformer will now be *forced* to pass in a markdown parser object.

Of course now we broke the code in our controller. Yep, in GenusController PhpStorm is *angry*: we're missing the required $markdownParser argument in the new MarkdownTransformer() call. That's cool - because now that we're in the controller, we *do* have access to that object. Pass in $this->get('markdown.parser'):

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 58
    public function showAction($genusName)
    {
... lines 61 - 69
        $markdownTransformer = new MarkdownTransformer(
            $this->get('markdown.parser')
        );
... lines 73 - 99
    }
... lines 101 - 125
}
```

Try it out!

It's alive! Twig is escaping the <p> tag - but that proves that markdown parsing *is* happening. The process we just did is dependency injection. It basically says: if an object needs something, you should pass it to that object. It's really programming 101. But if it still feels weird, you'll see a lot more of it.

Chapter 4: Being Awesome with Type-Hints

20 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
class MarkdownTransformer
{
... lines 7 - 8
    public function __construct($markdownParser)
... lines 10 - 18
}
```

What type of object is this $markdownParser argument? Oh, you can't tell? Well, neither can I. With no type-hint, this could be anything! A MarkdownParser object, a string, an octopus!

We need to add a *typehint* to make our code clearer... and avoid weird errors in case we accidentally pass in something else... like an octopus.

Run:

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

And select markdown.parser - that's the service we're passing into MarkdownTransformer. Ok, it's an instance of Knp\Bundle\MarkdownBundle\Parser\Preset\Max. We can use that as the type-hint.

But hold on - I'm going to complicate things... but then we'll all learn something cool and celebrate. Press shift+shift, type "max" and open that class:

14 lines | vendor/knplabs/knp-markdown-bundle/Parser/Preset/Max.php

```
... lines 1 - 6
/**
 * Full featured Markdown Parser
 */
class Max extends MarkdownParser
{

}
```

Ah, this extends MarkdownParser and *that* does all the work:

245 lines | vendor/knplabs/knp-markdown-bundle/Parser/MarkdownParser.php

```
... lines 1 - 8
/**
 * MarkdownParser
 *
 * This class extends the original Markdown parser.
 * It allows to disable unwanted features to increase performances.
 */
class MarkdownParser extends MarkdownExtra implements MarkdownParserInterface
{
... lines 17 - 243
}
```

And *this* implements a MarkdownParserInterface. We could type-hint with Max, MarkdownParser *or* MarkdownParserInterface: they will all work. BUT, when possible, it's best to find a base class - or better - and *interface* that

has the methods on it you need, and use that.

Type-hint the argument with MarkdownParserInterface:

22 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 4
use Knp\Bundle\MarkdownBundle\MarkdownParserInterface;

class MarkdownTransformer
{
... lines 9 - 10
    public function __construct(MarkdownParserInterface $markdownParser)
... lines 12 - 20
}
```

Why is this the best option? Two small reasons. First, in theory, we could swap out the $markdownParser for a different object, as long as it implemented this interface. Second, it's *really* clear *what* methods we can call on the $markdownParser property: only those on that interface.

But hold on a second, PhpStorm is angry about calling transform() on $this->markdownParser:

> Method "transform" not found in class MarkdownParserInterface

Weird! Open that interface. Oh, it has only one method: transformMarkdown():

16 lines | vendor/knplabs/knp-markdown-bundle/MarkdownParserInterface.php

```
... lines 1 - 4
interface MarkdownParserInterface
{
    /**
     * Converts text to html using markdown rules
     *
     * @param string $text plain text
     *
     * @return string rendered html
     */
    function transformMarkdown($text);
}
```

Hold on: to be clear: everything will work right now. Refresh to prove it.

The weirdness is just that we are *forcing* an object that implements MarkdownParserInterface to be passed in... but then we're calling a method that's *not* on that interface.

Change our call to transformMarkdown():

22 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 6
class MarkdownTransformer
{
... lines 9 - 15
    public function parse($str)
    {
        return $this->markdownParser
            ->transformMarkdown($str);
    }
}
```

Inside MarkdownParser, you can see that transformMarkdown() and transform() do the same thing anyways:

245 lines | vendor/knplabs/knp-markdown-bundle/Parser/MarkdownParser.php

```
... lines 1 - 14
class MarkdownParser extends MarkdownExtra implements MarkdownParserInterface
{
... lines 17 - 114
    public function transformMarkdown($text)
    {
        return parent::transform($text);
    }
... lines 119 - 243
}
```

This didn't change any behavior: it just made our code more portable: our class will work with *any* object that implements MarkdownParserInterface.

And if this doesn't *completely* make sense, do not worry. Just focus on this takeaway: when you need an object from inside a class, use dependency injection. And when you add the __construct() argument, type-hint it with either the class you see in debug:container *or* an interface if you can find one. Both totally work.

Chapter 5: Register your Service in the Container

The MarkdownTransformer class is *not* an Autobot, nor a Decepticon, but it *is* a service because it does work for us. It's no different from any other service, like the markdown.parser, logger or *anything* else we see in debug:container... except for one thing: it sees dead people. I mean, it does *not* live in the container.

Nope, we need to instantiate it *manually*: we can't just say something like $this->get('app.markdown_transformer') and expect the container to create it for us.

Time to change that... and I'll tell you why it's awesome once we're done.

Open up app/config/services.yml:

```
10 lines | app/config/services.yml
# Learn more about services, parameters and containers at
# http://symfony.com/doc/current/book/service_container.html
parameters:
#    parameter_name: value

services:
#    service_name:
#        class: AppBundle\Directory\ClassName
#        arguments: ["@another_service_name", "plain_value", "%parameter_name%"]
```

Tip

If you're using Symfony 3.3, your app/config/services.yml contains some extra code that may break things when following this tutorial! To keep things working - and learn about what this code does - see https://knpuniversity.com/symfony-3.3-changes

To add a new service to the container, you basically need to *teach* the container *how* to instantiate your object.

This file already has some example code to do this... kill it! Under the services key, give your new service a nickname: how about app.markdown_transformer:

```
10 lines | app/config/services.yml
... lines 1 - 5
services:
    app.markdown_transformer:
... lines 8 - 10
```

This can be anything: we'll use it later to *fetch* the service. Next, in order for the container to be able to instantiate this, it needs to know two things: the class name and what arguments to pass to the constructor. For the first, add class: then the full AppBundle\Service\MarkdownTransformer:

```
10 lines | app/config/services.yml
... lines 1 - 5
services:
    app.markdown_transformer:
        class: AppBundle\Service\MarkdownTransformer
... lines 9 - 10
```

For the second, add arguments: then make a YAML array: []. These are the constructor arguments, and it's pretty simple. If we used [sunshine, rainbows], it would pass the string sunshine as the first argument to MarkdownTransformer and rainbow as the second. And that would be a very pleasant service.

In reality, MarkdowTransformer requires *one* argument: the markdown.parser service. To tell the container to pass that, add @markdown.parser:

```
10 lines | app/config/services.yml
... lines 1 - 5
services:
    app.markdown_transformer:
        class: AppBundle\Service\MarkdownTransformer
        arguments: ['@markdown.parser']
```

That's it. The @ is special: it says:

> Woh woh woh, don't pass the *string* markdown.parser, pass the *service* markdown.parser.

And with 4 lines of code, a *new* service has been born in the container. I'm such a proud parent.

Go look for it:

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

There is its! And it's so cute. Idea! Let's use it! Instead of new MarkdownTransformer(), be lazier: $transformer = $this->get('app.markdown_transformer):

```
125 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 58
    public function showAction($genusName)
    {
... lines 61 - 69
        $markdownTransformer = $this->get('app.markdown_transformer');
... lines 71 - 97
    }
... lines 99 - 123
}
```

When this line runs, the container will create that object for us behind the scenes.

Why add a Service to the Container

Believe it or not, this was a *huge* step. When you add your service to the container, you get two great thing. First, using the service is *so* much easier: $this->get('app.markdown_transformer). We don't need to worry about passing constructor arguments: heck it could have *ten* constructor arguments and this simple line would stay the same.

Second: if we ask for the app.markdown_transformer service more than once during a request, the container *only* creates one of them: it returns that same *one* object each time. That's nice for performance.

Oh, and by the way: the container doesn't create the MarkdownTransformer object until and *unless* somebody asks for it. That means that adding more services to your container does *not* slow things down.

The Dumped Container

Ok, I *have* to show you something cool. Open up the var/cache directory. If you don't see it - you may have it excluded: switch to the "Project" mode in PhpStorm.

Open var/cache/dev/appDevDebugProjectContainer.php:

3709 lines | var/cache/dev/appDevDebugProjectContainer.php

```
... lines 1 - 9
10  /**
11   * appDevDebugProjectContainer.
12   *
13   * This class has been auto-generated
14   * by the Symfony Dependency Injection Component.
15   */
16  class appDevDebugProjectContainer extends Container
17  {
... lines 18 - 3707
3708 }
```

This is the container: it's a class that's dynamically built from our configuration. Search for "MarkdownTransformer" and find the getApp_MarkdownTransformerService() method:

3709 lines | var/cache/dev/appDevDebugProjectContainer.php

```
... lines 1 - 15
16  class appDevDebugProjectContainer extends Container
17  {
... lines 18 - 23
24      public function __construct()
25      {
... lines 26 - 32
33          $this->methodMap = array(
... line 34
35              'app.markdown_transformer' => 'getApp_MarkdownTransformerService',
... lines 36 - 238
239         );
... lines 240 - 249
250     }
... lines 251 - 272
273     /**
274      * Gets the 'app.markdown_transformer' service.
275      *
276      * This service is shared.
277      * This method always returns the same instance of the service.
278      *
279      * @return \AppBundle\Service\MarkdownTransformer A AppBundle\Service\MarkdownTransformer instance.
280      */
281     protected function getApp_MarkdownTransformerService()
282     {
283         return $this->services['app.markdown_transformer'] = new \AppBundle\Service\MarkdownTransformer($this->get('markdown.p
284     }
... lines 285 - 3707
3708 }
```

Ultimately, when we ask for the app.markdown_transformer service, *this* method is called. And look! It runs the same PHP code that we had before in our controller: new MarkdownTransformer() and then $this->get('markdown.parser') - since $this *is* the container.

You don't need to understand how this works - but it's important to see this. The configuration we wrote in services.yml may *feel* like magic, but it's not: it causes Symfony to write plain PHP code that creates our service objects. There's no magic: we *describe* how to instantiate the object, and Symfony writes the PHP code to do that. This makes the container blazingly fast.

Chapter 6: Injecting the Cache Service

Phew! Dependency injection, check! Registering new services, check! Delicious snack, check! Well, I *hope* you just had a delicious snack.

This tutorial is the start to our victory lap. We need to add caching to MarkdownTransformer: it should be pretty easy. Copy part of the *old* caching code and paste that into the parse() function. Remove the else part of the if and just return $cache->fetch():

32 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 6
class MarkdownTransformer
{
... lines 9 - 15
    public function parse($str)
    {
        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
        $key = md5($str);
        if ($cache->contains($key)) {
            return $cache->fetch($key);
        }
... lines 23 - 29
    }
}
```

Below, assign the method call to the $str variable and go copy the old $cache->save() line. Return $str and re-add the sleep() call so that things are *really* slow - that keeps it interesting:

32 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 6
class MarkdownTransformer
{
... lines 9 - 15
    public function parse($str)
    {
... lines 18 - 23
        sleep(1);
        $str = $this->markdownParser
            ->transformMarkdown($str);
        $cache->save($key, $str);

        return $str;
    }
}
```

On top, change the $funFact variables to $str. Perfect!

We know this won't work: there is no get() function in this class. And more importantly, we don't have access to the doctrine_cache.provider.my_markdown_cache service. How can we *get* access? Dependency injection.

Dependency Inject!

This time, add a *second* argument to the constructor called $cache. And hmm, we should give this a type-hint. Copy the service name and run:

```
$ ./bin/console debug:container doctrine_cache.providers.my_markdown_cache
```

This service is an instance of ArrayCache. But wait! Do *not* type-hint that. In our earlier course on environments, we setup a cool system that uses ArrayCache in the dev environment and FilesystemCache in prod:

72 lines | app/config/config.yml

```
... lines 1 - 7
parameters:
    locale: en
    cache_type: file_system
... lines 11 - 65
doctrine_cache:
    providers:
        my_markdown_cache:
            type: %cache_type%
            file_system:
                directory: %kernel.cache_dir%/markdown_cache
```

If we type-hint with ArrayCache, this will explode in prod because this service will be a different class.

Let's do some digging: open up ArrayCache:

95 lines | vendor/doctrine/cache/lib/Doctrine/Common/Cache/ArrayCache.php

```
... lines 1 - 32
class ArrayCache extends CacheProvider
{
... lines 35 - 93
}
```

This extends CacheProvider:

278 lines | vendor/doctrine/cache/lib/Doctrine/Common/Cache/CacheProvider.php

```
... lines 1 - 31
abstract class CacheProvider implements Cache, FlushableCache, ClearableCache, MultiGetCache
{
... lines 34 - 276
}
```

That might work. But *it* implements several interface - one of them is just called Cache. Let's try that. If this isn't the right interface - meaning it doesn't contain the methods we're using - PhpStorm will keep highlighting those after we add the type-hint:

35 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 7
class MarkdownTransformer
{
... lines 10 - 12
    public function __construct(MarkdownParserInterface $markdownParser, Cache $cache)
    {
... lines 15 - 16
    }
... lines 18 - 33
}
```

I'll use a keyboard shortcut - option+enter on a Mac - and select initialize fields:

35 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 7
class MarkdownTransformer
{
... line 10
    private $cache;

    public function __construct(MarkdownParserInterface $markdownParser, Cache $cache)
    {
... line 15
        $this->cache = $cache;
    }
... lines 18 - 33
}
```

All this did was add the private $cache property and set it in __construct(). You can also do that by hand.

Cool! Update parse() with $cache = $this->cache:

35 lines | src/AppBundle/Service/MarkdownTransformer.php

```
... lines 1 - 7
class MarkdownTransformer
{
... lines 10 - 18
    public function parse($str)
    {
        $cache = $this->cache;
... lines 22 - 32
    }
}
```

And look! All of the warnings went away. That *was* the right interface to use. Yay!

Because we added a new constructor argument, we need to update any code that instantiates the MarkdownTransformer. But now, that's not done by us: it's done by Symfony, and we help it in services.yml. Under arguments, add a comma and quotes. Copy the service name - @doctrine_cache.providers.my_markdown_cache and paste it here:

10 lines | app/config/services.yml

```
... lines 1 - 5
services:
    app.markdown_transformer:
        class: AppBundle\Service\MarkdownTransformer
        arguments: ['@markdown.parser', '@doctrine_cache.providers.my_markdown_cache']
```

That's it! That's the dependency injection pattern.

Go back to refresh. The sleep() should make it really slow. And it *is* slow. Refresh again: still slow because we setup caching to really only work in the prod environment.

Clear the prod cache:

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

And now add /app.php/ in front of the URI to use this environment. This should be slow the first time... but then fast after. Super fast! Caching is working. And dependency injection is behind us.

Chapter 7: Adding a Twig Extension + DI Tags

Because the $funFact needs to be parsed through markdown, we have to pass it as an independent variable into the template. You know what would be *way* cooler? If we could just say genus.funFact|markdown. Ok, actually we *can* do this already: the KnpMarkdownBundle comes with a filter called markdown:

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

```
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>

    <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>{{ genus.funFact|markdown }}</dd>
... lines 18 - 19
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 25 - 42
```

Remove the app.php from the URL and refresh. Voilà! This parses the string to markdown. Well, it doesn't look like much - but if you view the HTML, there's a p tag from the process.

Let's make this a little more obvious while we're working. Open up the Genus entity and find getFunFact(). Temporarily hack in some bold markdown code:

115 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 11
class Genus
{
... lines 14 - 86
    public function getFunFact()
    {
        return '**TEST** '.$this->funFact;
    }
... lines 91 - 113
}
```

Ok, try it again.

Nice! The bold **TEST** tells us that the markdown filter from KnpMarkdownBundle is working.

Ready for me to complicate things? I always do. The markdown filter uses the markdown.parser service from KnpMarkdownBundle - it does *not* use our app.markdown_transformer. And this means that it's *not* using our caching system. Instead, let's create our *own* Twig filter.

Creating a Twig Extension

To do that, you need a Twig "extension" - that's basically Twig's plugin system. Create a new directory called Twig - and nope, that name is *not* important. Inside, create a new php class - MarkdownExtension:

24 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... line 1

namespace AppBundle\Twig;

class MarkdownExtension extends \Twig_Extension
{
... lines 7 - 22
}
```

Remember: Twig is its own, independent library. If you read Twig's documentation about creating a Twig extension, it will tell you to create a class, make it extend \Twig_Extension and then fill in some methods.

Use the "Code"->"Generate" menu - or cmd+n - and select "Implement Methods". The *one* method you *must* have is called getName(). It's also the most boring: just make it return any unique string - like app_markdown:

24 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 4
class MarkdownExtension extends \Twig_Extension
{
... lines 7 - 18
    public function getName()
    {
        return 'app_markdown';
    }
}
```

To add a new filter, go back to the "Code"->"Generate" menu and select "Override Methods". Choose getFilters():

24 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 4
class MarkdownExtension extends \Twig_Extension
{
    public function getFilters()
    {
... lines 9 - 11
    }
... lines 13 - 22
}
```

Here, you'll return an array of new filters: each is described by a new \Twig_SimpleFilter object. The first argument will be the filter name - how about markdownify - that sounds fun. Then, point to a function in *this* class that should be called when that filter is used: parseMarkdown:

24 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 4
class MarkdownExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return [
            new \Twig_SimpleFilter('markdownify', array($this, 'parseMarkdown'))
        ];
    }
... lines 13 - 22
}
```

Create the new public function parseMarkdown() with a $str argument. For now, just strtoupper() that guy to start:

24 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 4
class MarkdownExtension extends \Twig_Extension
{
... lines 7 - 13
    public function parseMarkdown($str)
    {
        return strtoupper($str);
    }
... lines 18 - 22
}
```

Cool? Update the Twig template to use this:

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

```
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>

    <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>{{ genus.funFact|markdownify }}</dd>
... lines 18 - 19
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 25 - 42
```

Our Twig extension is perfect... but it will *not* work yet. Refresh. Huge error:

> Unknown markdownify filter.

Twig doesn't automatically find and load our extension. Somehow, we need to say:

> Hey Symfony? How's it going? Oh, I'm good. Listen, when you load the twig service, can you add my MarkdownExtension to it?

How are we going to do this? With *tags*.

Chapter 8: Tagging Services (and having Fun!)

We need to somehow tell Twig about our fun new Twig extension. To do that, first, register it as a service. The name doesn't matter, so how about app.markdown_extension. Set the class, but skip arguments: we don't have any yet, so this is optional:

```
15 lines | app/config/services.yml
... lines 1 - 5
6   services:
... lines 7 - 10
11      app.markdown_extension:
12          class: AppBundle\Twig\MarkdownExtension
... lines 13 - 15
```

Now, this service is a bit different: it's *not* something that we intend to use directly in our controller, like app.markdown_transformer. Instead, we simply want Twig to *know* about our service. We somehow need to raise our hand and say:

> Oh, oh oh! This service is special - this service is a *Twig Extension*!

We do that by adding a *tag*. The syntax is weird, so stay with me: Add tags:, then under that a dash and a set of curly-braces. Inside, set name to twig.extension:

```
15 lines | app/config/services.yml
... lines 1 - 5
6   services:
... lines 7 - 10
11      app.markdown_extension:
12          class: AppBundle\Twig\MarkdownExtension
13          tags:
14              - { name: twig.extension }
```

And that's it.

Real quick - make sure it works. Refresh! Boom! We see a pretty awesome-looking upper-case string.

What are these Tags???

Tags are *the* way to hook your services into different parts of the core system. When Symfony creates the twig service, it looks for *all* services in the container that are *tagged* with twig.extension. It then configures these as extensions on twig.

Google for "Symfony dependency injection tags": there's an awesome reference section on Symfony.com called The Dependency Injection Tags. It lists *every* tag that can be used to hook into core Symfony. And if you're tagging a service, well, you're probably doing something really cool.

For example, if you want to register an event listener and actually hook into Symfony's boot process, you create a service and then tag it with kernel.event_listener. You won't memorize all of these: you just need to understand their purpose. Someday, you'll read some docs or watch a tutorial here that will tell you to *tag* a service. I want you to understand what that tag is actually doing.

Finish the Extension

Let's finish our extension: we need to parse this through markdown. We find ourselves in a familiar position: we're inside a service and need access to some *other* service: MarkdownTransformer. Dependency injection!

Add public function __construct() with a MarkdownTransformer argument. I'll hold option+enter and select "Initialize fields" as a shortcut. Again, this just added the property for me and assigned it in __construct():

33 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 6
class MarkdownExtension extends \Twig_Extension
{
    private $markdownTransformer;

    public function __construct(MarkdownTransformer $markdownTransformer)
    {
        $this->markdownTransformer = $markdownTransformer;
    }
... lines 15 - 31
}
```

Go Deeper!

Watch our PhpStorm Course to learn about these great shortcuts.

In parseMarkdown(), return $this->markdownTransformer->parse() and pass it $str:

33 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 6
class MarkdownExtension extends \Twig_Extension
{
... lines 9 - 22
    public function parseMarkdown($str)
    {
        return $this->markdownTransformer->parse($str);
    }
... lines 27 - 31
}
```

The last step is to update our service in services.yml. Add arguments: ['@app.markdown_transformer']:

16 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 10
    app.markdown_extension:
... lines 12 - 14
        arguments: ['@app.markdown_transformer']
```

Refresh! And it's working. Now, let me show you a shortcut.

Chapter 9: Autowiring Madness

Ooh, bonus feature! In services.yml, remove arguments and instead just say autowire: true:

17 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 10
    app.markdown_extension:
... lines 12 - 14
        #arguments: ['@app.markdown_transformer']
        autowire: true
```

Refresh again. It still works! But how? We didn't tell Symfony what arguments to pass to our constructor? What madness is this!? With autowire: true, Symfony reads the *type-hints* for each constructor argument:

33 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 6
class MarkdownExtension extends \Twig_Extension
{
... lines 9 - 10
    public function __construct(MarkdownTransformer $markdownTransformer)
... lines 12 - 31
}
```

And tries to automatically find the correct service to pass to you. In this case, it saw the MarkdownTransformer type-hint and knew to use the app.markdown_transformer service: since that is an instance of this class. You can also type-hint interfaces.

Tip

The autowiring logic has changed in Symfony 3.3 and higher. For more info, we have a tutorial! https://knpuniversity.com/screencast/symfony-3.3/autowiring-logic

This doesn't *always* work, but Symfony will give you a big clear exception if it can't figure out what to do. But when it *does* work, it's a great time saver.

Auto-Escaping a Twig Filter

The HTML is *still* being escaped - I don't want to finish before we fix that! We *could* add the |raw filter... but let's do something cooler. Add a third argument to Twig_SimpleFilter: an options array. Add is_safe set to an array containing html:

35 lines | src/AppBundle/Twig/MarkdownExtension.php

```
... lines 1 - 6
class MarkdownExtension extends \Twig_Extension
{
... lines 9 - 15
    public function getFilters()
    {
        return [
            new \Twig_SimpleFilter('markdownify', array($this, 'parseMarkdown'), [
                'is_safe' => ['html']
            ])
        ];
    }
... lines 24 - 33
}
```

This means it's *always* safe to output contents of this filter in HTML. Refresh one last time. Beautiful.

Where now!?

Oh my gosh guys! I think you just leveled up: Symfony offense increased by five points. Besides the fact that a *lot* more things will start making sense in Symfony, you also know everything you need to start organizing your code into service classes - that whole *service-oriented architecture* thing I was talking about earlier. This will lead you to *wonderful* applications.

There's really nothing that we can't do now in Symfony. In the next courses, we'll use all this to master new tools like forms and security. Seeya next time!