**Sobre bundles en symfony 5**

**Following** [**https://symfonycasts.com/screencast/symfony-bundle/bundle-services#play**](https://symfonycasts.com/screencast/symfony-bundle/bundle-services#play)

En este tutorial lo primero que se hace es crear el bundle dentro pero fuera del proyecto, es decir, fuera de su carpeta SRC aventurando que en un futuro será totalmente independiente, en pasos posteriores lo separará del todo, pero de momento, para probar compatibilidad y cohesión lo hace de este modo. En el capitulo 9 empieza su separación.

Crear un proyecto básico en symfony 5

composer create-project symfony/website-skeleton bundles52 5.2.\*

composer require apache-pack

Crear un directorio lib/nbrdelbundle/src

Y meter ahí la clase en cuestión

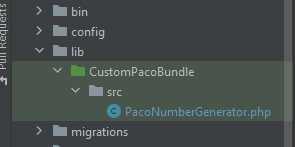
Como namespace ponerle el que quieras que sea el de tu bundle

namespace Paco\CustomPacoBundle;  
  
class PacoNumberGenerator  
{  
 public function getNumber(){  
 return 4564;  
 }  
}

Al intentar usar esta clase aunque en phpstorm te la linca y symfony te deja inyectarla obtendrás un class not found a la hora de ejecutarla =>

Add de añadir otra entrada en el psr-4 de composer.json

"autoload-dev": {  
 "psr-4": {  
 "Paco\\CustomPacoBundle\\": "lib/CustomPacoBundle/src/",  
 "App\\Tests\\": "tests/"  
 }  
},



Y ejecutar composer dump-autoload.

Ahora fallaría al intentar inyectarlo como servicio ya que el autowiring de symfony solo apunta a las clases dentro de src/ => voy al services.yml y añado mi clase disponible para autowiring añadiendo al final:

Paco\CustomPacoBundle\PacoNumberGenerator: ~

Ahora mismo tenemos una librería, que no es un bundle, la diferencia es que una librería exige que añadas configuración adicional para usarla como servicio en un contenedor de symfony mientras que un bundle nosotros añadimos esa configuración en el propio bundle y ya no es necesario añadir configuración extra fuera. Un bundle puede añadir automáticamente servicios al contenedor y para ello usaremos el directory DependencyInjection con su correspondiente clase, ver más abajo(2):

Creamos el archivo PacoNumberGeneratorBundle y lo añadimos al bundles.php

<?php  
  
  
namespace Paco\CustomPacoBundle;  
  
use Symfony\Component\HttpKernel\Bundle\Bundle;  
  
class PacoNumberGeneratorBundle extends Bundle  
{  
  
}

return [  
 Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],  
 Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],  
 Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],  
 Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],  
 Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],  
 Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true, 'test' => true],  
 Symfony\Bundle\MakerBundle\MakerBundle::class => ['dev' => true],  
 Doctrine\Bundle\DoctrineBundle\DoctrineBundle::class => ['all' => true],  
 Doctrine\Bundle\MigrationsBundle\DoctrineMigrationsBundle::class => ['all' => true],  
 Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],  
 Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],  
 Paco\CustomPacoBundle\PacoNumberGeneratorBundle::class => ['all' => true],  
];

(2) Creamos la carpeta DependencyInjection y dentro el archivo:

<?php  
  
  
namespace Paco\CustomPacoBundle\DependencyInjection;  
  
  
use Symfony\Component\DependencyInjection\ContainerBuilder;  
use Symfony\Component\DependencyInjection\Extension\Extension;  
  
class PacoNumberGeneratorExtension extends Extension  
{  
 public function load(array $configs, ContainerBuilder $container)  
 {  
 // *TODO: Implement load() method.* var\_dump("in the bundle");die;  
 }  
}

Cuando symfony intenta carga los bundles, itera sobre todos los bundles y busca por el directorio DependencyInjection, dentro por la clase con el mismo nombre del bundle terminada en “Extension”, luego si existe, la instancia y ejecuta el método “load” dentro de ella. Asi usaremos esto para añadir servicios… (..)

Ahora moveremos la notación que hicimos en el services.yml general a dentro de nuestro bunlde, asi borramos:

Paco\CustomPacoBundle\PacoNumberGenerator: ~

del services.yaml general y creamos el nuestro propio dentro de Resources/config/services.yaml:

*# config/services.yaml*services:  
 *# default configuration for services in \*this\* file* \_defaults:  
 autowire: true *# Automatically injects dependencies in your services.* autoconfigure: true *# Automatically registers your services as commands, event subscribers, etc.  
  
 # makes classes in src/ available to be used as services  
 # this creates a service per class whose id is the fully-qualified class name* Paco\CustomPacoBundle\:  
 resource: '../../../src/\*'  
 exclude: '../../../src/{DependencyInjection,Entity,Tests,Kernel.php}'  
  
 *# ...*

Y cargamos esa configuración dentro del método load del DependencyInjection:

<?php  
  
  
namespace Paco\CustomPacoBundle\DependencyInjection;  
  
  
use Symfony\Component\Config\FileLocator;  
use Symfony\Component\DependencyInjection\ContainerBuilder;  
use Symfony\Component\DependencyInjection\Extension\Extension;  
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;  
  
class PacoNumberGeneratorExtension extends Extension  
{  
 public function load(array $configs, ContainerBuilder $container)  
 {  
 // *TODO: Implement load() method.* $loader = new YamlFileLoader($container, new FileLocator(*\_\_DIR\_\_*.'/../Resources/config'));  
 $loader->load('services.yaml');  
 }  
}

Importante:

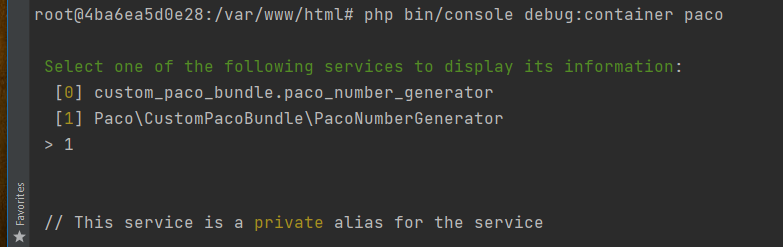
En nuestra aplicación usualmente llamaremos al id del servicio (alias) y no a su clase => vamos a añadir la configuración con el id del servicio a la declaración del mismo:

*# config/services.yaml*services:  
 *# default configuration for services in \*this\* file* \_defaults:  
 autowire: true *# Automatically injects dependencies in your services.* autoconfigure: true *# Automatically registers your services as commands, event subscribers, etc.  
  
 # makes classes in src/ available to be used as services  
 # this creates a service per class whose id is the fully-qualified class name  
# Paco\CustomPacoBundle\:  
# resource: '../../../src/\*'  
# exclude: '../../../src/{DependencyInjection,Entity,Tests,Kernel.php}'  
  
 # ...  
  
 # the id is not a class, so it won't be used for autowiring* custom\_paco\_bundle.paco\_number\_generator:  
 class: Paco\CustomPacoBundle\PacoNumberGenerator  
 public: true  
  
 *# but this fixes it!  
 # the ``app.rot13.transformer`` service will be injected when  
 # an ``App\Util\Rot13Transformer`` type-hint is detected* Paco\CustomPacoBundle\PacoNumberGenerator: '@custom\_paco\_bundle.paco\_number\_generator'

IMPOR:

*# the id is not a class, so it won't be used for autowiring*custom\_paco\_bundle.paco\_number\_generator:  
 class: Paco\CustomPacoBundle\PacoNumberGenerator  
 public: true  
  
*# but this fixes it!*Paco\CustomPacoBundle\PacoNumberGenerator: '@custom\_paco\_bundle.paco\_number\_generator'

Vemos como a pesar de figurar dos servicios uno es un alias del otro:



Los servicios pueden ser public o privados, por defecto son privados. Si se usan usando inyección de dependencias no es necesario que sean públicos, pero si alguien los quiere instanciar directamente si necesitará que estén seteados como public específicamente ( se supone…)

**Pasando argumentos a nuestro bundle desde la app principal**

Symfony procesa todos los yaml que hay en config packages y busca una Extension con la key a procesar a la que le pasará los datos, así creamos en config/packages:

Paco\_number\_generator.yaml:

paco\_number\_generator:  
 bar: true

y ese parámetro será pasado en el array de $configs a nuestro bundle

vale, ahora el tema está en pasarlo a la clase en cuestión de nuestro blunde, a su constructor

creamos la clase Configuration dentro de DependencyInjection y ahí checkeamos que los parámetros que nos pasan son del tipo valido:

class Configuration implements ConfigurationInterface  
{  
  
 public function getConfigTreeBuilder()  
 {  
 $treeBuilder = new TreeBuilder('custom\_paco\_bundle');  
 $rootNode = $treeBuilder->getRootNode();  
 $rootNode  
 ->children()  
 //->booleanNode('unicorns\_are\_real')->defaultTrue()->end()  
 ->integerNode('basenumber')->defaultValue(3)->end()  
 ->integerNode('topnumber')->defaultValue(3)->end()  
 ->end()  
 ;  
 return $treeBuilder;  
 }  
}

una vez hecho esto, dentro del PacoNumberGeneratorExtension en DependencyInjection recojo esa configuración y la paso a la clase:

public function load(array $configs, ContainerBuilder $container)  
{  
 //var\_dump($configs);die;  
 // *TODO: Implement load() method.* $loader = new YamlFileLoader($container, new FileLocator(*\_\_DIR\_\_*.'/../Resources/config'));  
 $loader->load('services.yaml');  
  
 //$loader = new XmlFileLoader($container, new FileLocator(\_\_DIR\_\_.'/../Resources/config'));  
 //$loader->load('services.xml');  
  
 //we pass the argument from the config/packages/paco\_number\_generator.yml to the class  
 $configuration = $this->getConfiguration($configs, $container);  
 $config = $this->processConfiguration($configuration, $configs);  
  
 $definition = $container->getDefinition('custom\_paco\_bundle.paco\_number\_generator');  
 $definition->setArgument(0, $config['basenumber']);  
 $definition->setArgument(1, $config['topnumber']);  
}

**Permitir inyectar o sobreescribir servicios de dentro del bundle desde fuera:**  
En el ejemplo haremos una clase provider que ponga una cadena aleatoria delante del numero y permitiremos desde fuera del bundle inyectar una clase de este tipo:

Añadimos una clase a nuestro bundle y la anexamos a nuestro constructor:

class MeetingMessageProvider  
{  
 public function getWordList(): array  
 {  
 return [  
 'Hi adorable ',  
 'Hi active ',  
 'Hi admire ',  
 'Hi adventurous ',  
 'Hi agreeable ',  
 ];  
 }  
}

class PacoNumberGenerator  
{  
 private $baseNumber;  
 private $topNumber;  
 private $meetingMessageProvider;  
  
 public function \_\_construct(MeetingMessageProvider $meetingMessageProvider,$baseNumber = 0, $topNumber = 100){  
 $this->baseNumber = $baseNumber;  
 $this->topNumber = $topNumber;  
 $this->meetingMessageProvider = $meetingMessageProvider;  
 }  
  
 public function getNumber(){  
 return random\_int($this->baseNumber, $this->topNumber);  
 }  
  
 private function getWordList(): array  
 {  
 return $this->meetingMessageProvider->getWordList();  
 }  
}

Tb la añadimos a nuestro services.yaml tal que :

If you're used to configuring services in YAML, the type="service" is equivalent to putting an @ symbol before the service id

*# config/services.yaml*services:  
 *# default configuration for services in \*this\* file* \_defaults:  
 autowire: true *# Automatically injects dependencies in your services.* autoconfigure: true *# Automatically registers your services as commands, event subscribers, etc.  
  
 # makes classes in src/ available to be used as services  
 # this creates a service per class whose id is the fully-qualified class name  
# Paco\CustomPacoBundle\:  
# resource: '../../../src/\*'  
# exclude: '../../../src/{DependencyInjection,Entity,Tests,Kernel.php}'* custom\_paco\_bundle.meeting\_message\_provider:  
 class: Paco\CustomPacoBundle\MeetingMessageProvider  
  
 *# the id is not a class, so it won't be used for autowiring* custom\_paco\_bundle.paco\_number\_generator:  
 class: Paco\CustomPacoBundle\PacoNumberGenerator  
 arguments: ['@custom\_paco\_bundle.meeting\_message\_provider']  
 public: true  
  
 *# but this fixes it!  
 # the ``app.rot13.transformer`` service will be injected when  
 # an ``App\Util\Rot13Transformer`` type-hint is detected* Paco\CustomPacoBundle\PacoNumberGenerator: '@custom\_paco\_bundle.paco\_number\_generator'

Y lo inyectamos via Configuration.php en DependencyInjection:

$rootNode  
 ->children()  
 //->booleanNode('unicorns\_are\_real')->defaultTrue()->end()  
 ->integerNode('basenumber')->defaultValue(3)->end()  
 ->integerNode('topnumber')->defaultValue(3)->end()  
 ->scalarNode('meeting\_message\_provider')->defaultNull()->end()  
 ->end()  
;

Y en PacoNumberGeneratorExtension dentro de DependencyInjection comprobamos que no nos pasan un servicio nulo y lo pasamos al constructor:

if (null !== $config['meeting\_message\_provider']) {  
 $definition->setArgument(0, $config['meeting\_message\_provider']);  
}  
  
$definition->setArgument(1, $config['basenumber']);  
$definition->setArgument(2, $config['topnumber']);

ok, hacemos algunas modificaciones en la clase general para que use ese provider y haga algo:

public function getNumber()  
{  
 $randomMeetingIndex = array\_rand($this->meetingMessageProvider->getWordList(), 1);  
 $randomNumber = random\_int($this->baseNumber, $this->topNumber);  
 $randomMeetingWord = $this->meetingMessageProvider->getWordList()[$randomMeetingIndex];  
 return $randomMeetingWord." ". $randomNumber;  
}

Ahora vamos a sobrecargar ese parámetro para que, desde la aplicación externa podamos pasar una clase custom al bundle:

Creamos un MeetingExternCustom tal que:

<?php  
  
  
namespace App\Services;  
  
  
use Paco\CustomPacoBundle\MeetingMessageProvider;  
  
class MeetingExternCustom extends MeetingMessageProvider  
{  
 public function getWordList(): array  
 {  
 //$words = parent::getWordList();  
 $words = ['Hi outside', 'Hi out'];  
 return $words;  
 }  
}

Y la inyectamos via paco\_number\_generator.yaml:

paco\_number\_generator:  
 basenumber: 200  
 topnumber: 205  
 meeting\_message\_provider: App\Services\MeetingExternCustom

Al refrescar nos da un petting:

**Argument 1 passed to Paco\CustomPacoBundle\PacoNumberGenerator::\_\_construct() must be an instance of Paco\CustomPacoBundle\MeetingMessageProvider, string given, called in /var/www/html/var/cache/dev/Container1GyvuK0/getCustomPacoBundle\_PacoNumberGeneratorService.php on line 22**

La razón del petting es que realmente se esta pasando una cadena y no esta construyendo la referencia al servicio externo como clase:

|  |
| --- |
| Go back to our extension class. Here's the fix: when we set the argument to $config['word\_provider'], this of course sets that argument to the string value! To fix this in YAML, we would prefix the service id with the @ symbol. In PHP, wrap the value in a new Reference() object. This tells Symfony that we're referring to a service. |

Para solventar eso hacemos esta modificación en el método load del PacoNumberGeneratorExtension.php dentro del dependencyInjection folder:

if (null !== $config['meeting\_message\_provider']) {  
 //$definition->setArgument(0, $config['meeting\_message\_provider']);  
 $definition->setArgument(0, new Reference($config['meeting\_message\_provider']));  
}

y todo ok, estamos cargado con un servicio externo un servicio usado dentro de nuestro bundle!

Paso de refinamiento, en este bloque lo que hacemos es dar un alias a ese servicio que hemos hecho al MeetingMessageProvider con lo que cambiaremos la forma en la que se setea como argumento y además crearemos un interface para que desde fuera el que quiera sobrecargar nuestra funcionalidad no tenga que extender de nuestra clase, solo tenga que implementar una interfaz.

Allá vamos!:

Pongo un “alias” al servicio y public =true

custom\_paco\_bundle.meeting\_message\_provider:  
 class: Paco\CustomPacoBundle\MeetingMessageProvider  
 public: true  
  
Paco\CustomPacoBundle\MeetingMessageProvider: '@custom\_paco\_bundle.meeting\_message\_provider'

Gracias a eso podemos eliminar la línea de NewReference cuando inyectamos las dependencias, quedaría asi:

if (null !== $config['meeting\_message\_provider']) {  
 //$definition->setArgument(0, $config['meeting\_message\_provider']);  
 //$definition->setArgument(0, new Reference($config['meeting\_message\_provider']));  
 $container->setAlias('custom\_paco\_bundle.meeting\_message\_provider', $config['meeting\_message\_provider']);  
}

Ahora creamos una interfaz para evitar que quien quiera implementar o personalizar un servicio dentro de nuestro bundle tenga que extender de una clase en lugar de implementar una interfaz:

namespace Paco\CustomPacoBundle\Interfaces;  
  
interface MeetingMessageProviderInterface  
{  
 */\*\*  
 \* Return an array of words to use for the fake text.  
 \*  
 \** ***@return*** *array  
 \*/* public function getWordList(): array;  
}

asi nuestra clase number generator recibirá esa clase como parámetro:

public function \_\_construct(MeetingMessageProviderInterface $meetingMessageProvider, $baseNumber = 0, $topNumber = 100)  
{

y tanto nuestro meetingMessageProvider como el servicio externo que hace lo mismo implementarán esa interfaz:

class MeetingMessageProvider implements MeetingMessageProviderInterface  
{

**9.- Separar el bundle, crear su propio composer.json**

1) Sacarlo de la carpeta lib a su propia carpeta CustomPacoBundle y añadir git

mv lib\CustomPacoBundle\ ../CustomPacoBundle

git init

git status

git add .

git commit -m "Unicorns"

2) Inicializar composer:

composer init

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [kueva/custom-paco-bundle]: paco/custom-paco-bundle

Description []: Example of bundle, just a random number with random string service

Author [jonytrifulca <jonytrifulca@hotmail.com>, n to skip]:

Minimum Stability []:

Package Type (e.g. library, project, metapackage, composer-plugin) []: symfony-bundle

License []: MIT

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no

Would you like to define your dev dependencies (require-dev) interactively [yes]? no

{

"name": "paco/custom-paco-bundle",

"description": "Example of bundle, just a random number with random string service",

"type": "symfony-bundle",

"license": "MIT",

"authors": [

{

"name": "jonytrifulca",

"email": "jonytrifulca@hotmail.com"

}

],

"require": {}

}

Do you confirm generation [yes]?

Would you like the vendor directory added to your .gitignore [yes]?

4) Añadimos los requires:

"require": {  
 "php": "^7.1.3"  
}

5) Añadimos el autoload:

"autoload": {  
 "psr-4": {  
 "Paco\\CustomPacoBundle\\": "src/"  
 }  
}

6) En este punto ya casi estaríamos lo que se puede hacer para probar este bundle sin subirlo a packagelist es setear en nuestro proyecto un repositorio disponible via path:

|  |
| --- |
| Click on the "Repositories" documentation and... all the way at the bottom... there's a path option! This allows us to point to any directory on our computer that contains a composer.json file. Then, suddenly, that library becomes available to composer require. |

Añado al proyecto principal en el composer.json esto:

},  
 "repositories": [  
 {  
 "type": "path",  
 "url": "../PacoCustomBundle/"  
 }  
 ]  
}

Y hago un :

composer require paco/custom-paco-bundle:\*@dev

vale, esto peta pk ese directorio no esta dentro del docker, hago lo siguiente:

mapear el directorio anterior a tmp de docker y queda asi:

volumes:  
 - ./:/var/www/html  
 - ../:/tmp/dir  
 - .docker/apache-php/vhost.conf:/etc/apache2/sites-available/000-default.conf

"repositories": [  
 {  
 "type": "path",  
 "url": "/tmp/dir/CustomPacoBundle"  
 }  
]

Y ejecuto dentro del docker:

composer require paco/custom-paco-bundle:\*@dev

Refresco la pagina y todo sigue funcionando sin problemas, pero dentro de vendor lo que hay es un enlace simbolico al bundle, no ha copiado los archivos… raro raro. Vale ok, ha de ser asi realmente XD.

|  |
| --- |
| And, cool! On my system, it installed with a symlink, which means we can keep hacking on the bundle and testing it live in the app.  Oh, and since Symfony flex noticed that our package has a symfony-bundle type, it actually tried to configure a recipe, which would normally enable the bundle for us in bundles.php. It didn't this time, only because we already have that code.  Now that everything is reconnected, it should work! Refresh the page. Yes! That bundle is properly living on its own. |

Si lo quisiese hacer con git:

|  |
| --- |
| "repositories": [  {  "type": "vcs",  "url": "git@github.com:username/name-to-bundle.git"  }  ] |

En este caso, probablemente vaya totalmente al vendor, evidentemente.

**10.- Testing the bundle ( todo dentro del código aislado del bundle)**

1) Crear la carpeta test dentro del bundle

2) Configuramos el namespace en composer.json del bundle:

"autoload-dev": {  
 "psr-4": {  
 "Paco\\CustomPacoBundle\\Tests\\": "tests/"  
 }  
}

|  |
| --- |
| By putting it in autoload-dev, the autoload rules for the tests/ directory will not be added to the autoload matrix of our users' applications, which will give them a slight performance boost. |

3) Añadimos un test dentro

4) Instalamos phpunit, solo para nosotros, en el desarrollo del bundle, de ahí el --dev

composer require symfony/phpunit-bridge --dev

Y metemos en el .gitignore el composer.lock, no debemos distribuirlo con nuestro bundle!!!

5) Ejecutar los test Al ejecutar dentro del docker ( en el directorio de nuestro bundle tmp si lo hemos hecho asi):

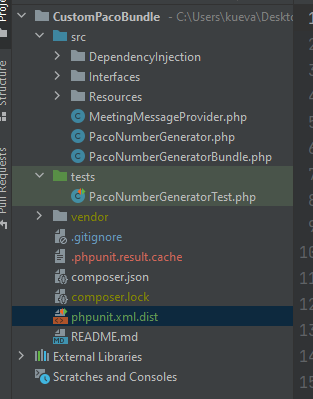
./vendor/bin/simple-phpunit

nos salen las opciones de phpunit, la razón es que no hemos creado un phpunit.xml.dist

lo creamos tal que :

<?xml version="1.0" encoding="UTF-8"?>  
  
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.1/phpunit.xsd"  
 backupGlobals="false"  
 colors="true"  
 bootstrap="./vendor/autoload.php"  
>  
 <php>  
 <ini name="error\_reporting" value="-1" />  
 <ini name="intl.default\_locale" value="en" />  
 <ini name="intl.error\_level" value="0" />  
 <ini name="memory\_limit" value="-1" />  
 </php>  
  
 <testsuites>  
 <testsuite name="Test suite">  
 <directory suffix="Test.php">./tests</directory>  
 </testsuite>  
 </testsuites>  
  
 <filter>  
 <whitelist>  
 <directory>./src</directory>  
 </whitelist>  
 </filter>  
</phpunit>

Y al ejecutar los test todo ok.



**11.- Service integration test**

|  |
| --- |
| And here is where things get interesting. We basically want to initialize our bundle into a real app, and check that the container has that service. But... we do not have a Symfony app lying around! So... let's make the smallest possible Symfony app ever.  To do this, we just need a Kernel class. And instead of creating a new file with a new class, we can hide the class right inside this file, because it's only needed here.  Add class KnpULoremIpsumTestingKernel extends Kernel from... wait... why is this not auto-completing the Kernel class? There should be one in Symfony's HttpKernel component! What's going on? |

Para hacer este test necesitamos añadir algunas dependencias a nuestro bundle, añadimos al composer.json: y hacemos composer install o hacemos uno por uno composer require con lo que ya te instalará la version que sea:

"require": {  
 "php": "^7.1.3",  
 "symfony/config": "^5.2",  
 "symfony/dependency-injection": "^5.2",  
 "symfony/http-kernel": "^5.2",  
 "symfony/yaml": "^5.2"

},

Y la clase que comprueba la conexión queda tal cual:

class FunctionalTest extends TestCase  
{  
 public function testServiceWiring()  
 {  
 //this 3 lines to boot a real symfony app  
 $kernel = new RealSymfonyAPPTestingKernel('test', true);  
 $kernel->boot();  
 $container = $kernel->getContainer();  
  
 //we test that there is a right wiring  
 $ipsum = $container->get('custom\_paco\_bundle.paco\_number\_generator');  
 $this->assertInstanceOf(PacoNumberGenerator::class, $ipsum);  
 $this->assertInternalType('string', $ipsum->getNumber());  
 }  
}  
  
class RealSymfonyAPPTestingKernel extends Kernel  
{  
 //we use to enable our bundle  
 public function registerBundles()  
 {  
 return [  
 new PacoNumberGeneratorBundle(),  
 ];  
 }  
 public function registerContainerConfiguration(LoaderInterface $loader)  
 {  
 }  
}

**12.- Complex test config**

Vamos a probar ahora nuestra configuración, es decir, que los parámetros que seteamos desde una aplicación externa que referencian a nuestro bundle se están pasando correctamente. Para ello vamos a probarlo de la siguiente forma:

|  |
| --- |
| We're going to use the third option: boot a real app with some config, and test the final services. Specifically, I want to test that the custom word\_provider config works. |

1. Creamos una clase fake que implemente el interfaz que la clase de nuestro bundle esta ready para recibir:

class ExampleOfMessageProvider implements MeetingMessageProviderInterface  
{  
 public function getWordList(): array  
 {  
 return ['fake1test', 'fake2test'];  
 }  
}

1. Creamos el método para testearlo:

public function testServiceWiringWithConfiguration()  
{  
 //init app with custom configuration  
 $kernel = new RealSymfonyAPPTestingKernel([  
 'meeting\_message\_provider' => 'example\_meeting\_provider'  
 ]);  
 $kernel->boot();  
 $container = $kernel->getContainer();  
  
 //test that the service injected is the correct service  
 $ipsum = $container->get('custom\_paco\_bundle.paco\_number\_generator');  
 $this->assertContains('fake', $ipsum->getNumber());  
}

1. Para para ello hemos de modificar el RealSymfonyAPPTestingKernel para emular que carga una configuración determinada:

Pero claro en lugar de que parsee un .yaml lo emulamos con un bloque de php con loader->load llamando a un callback function ContainerBuilder…(…) y le pasamos el servicio o clase

//this is the method that's responsible for parsing all the YAML files in the config/packages directory and the services.yaml file.  
public function registerContainerConfiguration(LoaderInterface $loader)  
{  
 //Instead of parsing YAML files, we can instead put all that logic into PHP with $loader->load()  
 //passing it a callback function with a ContainerBuilder argument.  
 // Inside of here, we can start registering services and passing bundle extension configuration  
 $loader->load(function(ContainerBuilder $container) {  
 $container->register('example\_meeting\_provider', ExampleOfMessageProvider::class);  
 $container->loadFromExtension('paco\_number\_generator', $this->bundleConfig);  
 });  
}

**13- Adding routes & controllers**

Añadiendo ejemplos de rutas y controladores a nuestro bundle.

Creamos un controller de ejemplo, creando la carpeta Controller correspondiente:

class CustomBundleApiController extends AbstractController  
{  
 private $pacoNumberGenerator;  
  
 public function \_\_construct(PacoNumberGenerator $pacoNumberGenerator)  
 {  
 $this->pacoNumberGenerator = $pacoNumberGenerator;  
 }  
   
 public function index()  
 {  
 return $this->json(  
 [  
 'number' => $this->pacoNumberGenerator->getNumber(),  
 ]  
 );  
 }  
}

|  |
| --- |
| When you create a controller in a reusable bundle, the best practice is to register your controller as a proper service and use dependency injection to get anything you need. |

Editamos el services.yml:

*#controller as service  
#When you create a controller in a reusable bundle, the best practice is to register your  
#controller as a proper service and use dependency injection to get anything you need*custom\_paco\_bundle.number\_generator\_api\_controller:  
 class: Paco\CustomPacoBundle\Controller\NumberGeneratorApiController  
 arguments: ['@custom\_paco\_bundle.paco\_number\_generator']  
 public: true  
 calls:  
 - [ setContainer,["@service\_container"] ]  
 tags:  
 - name: 'container.service\_subscriber'

Y añadimos las rutas pertinentes, creando un config/routes.yml(xml)

paco\_number\_generator\_api:  
 path: /  
 controller: custom\_paco\_bundle.number\_generator\_api\_controller:index

Vale ahora tenemos esto preparado para que el usuario final pueda usarlo desde su aplicación, para ello nos vamos a routes/ y creamos un archivo tal que paco\_number\_generator.yaml y dentro ponemos:

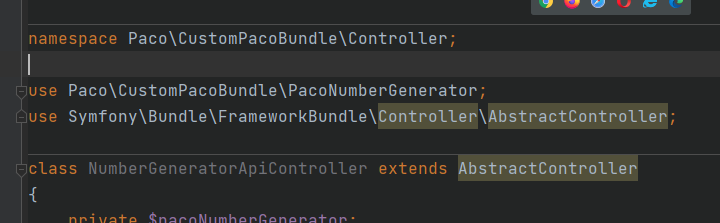
\_number\_generator:resource: '@PacoNumberGeneratorBundle/Resources/config/routes.yml'  
 prefix: /api/numbergenerator

La pregunta es pk funciona esto si en nuestro bundle estamos haciendo uso de AbstractController y realmente no lo tenemos instalado en el bundle?

("symfony/framework-bundle": "5.2.\*",

)

La razón es que si esta instalado en la app que lo usa desde fuera.



**14.- Controller functional test**

Haremos un test funcional que compruebe que nuestra ruta básica del bundle funciona.  
Dentro de tests creamos Controller y dentro

NumberGeneratorApiControllerTest

|  |
| --- |
| The setup for a functional test is pretty similar to an integration test: create a custom test kernel, but this time, import routes.xml inside. Then, we can use Symfony's BrowserKit to make requests into that kernel and check that we get a 200 status code back. |

Ahora si que necesitaremos en nuestro bundel el symfony/framework necesario para rutas y controladores:

composer require symfony/framework-bundle --dev

Tb necesitaremos:

composer require symfony/browser-kit –dev

|  |
| --- |
| For some reason, there is a missing parameter. It turns out that FrameworkBundle has one required piece of configuration. In your application, open config/packages/framework.yaml. Yep, right on top: the secret key. |

Configuramos el test tal que :

<?php  
  
  
namespace Paco\CustomPacoBundle\Tests\Controller;  
  
use Paco\CustomPacoBundle\PacoNumberGeneratorBundle;  
use PHPUnit\Framework\TestCase;  
use Symfony\Bundle\FrameworkBundle\FrameworkBundle;  
use Symfony\Component\Config\Loader\LoaderInterface;  
use Symfony\Component\DependencyInjection\ContainerBuilder;  
use Symfony\Component\HttpKernel\Kernel;  
use Symfony\Component\Routing\Loader\Configurator\RoutingConfigurator;  
use Symfony\Bundle\FrameworkBundle\KernelBrowser;  
use Symfony\Component\Routing\RouteCollectionBuilder;  
  
class NumberGeneratorApiControllerTest extends TestCase  
{  
  
 public function testIndex()  
 {  
 //instantiate the fake app:  
 $kernel = new NumberGeneratorApiControllerKernel();  
 $client = new KernelBrowser($kernel);  
 $client->request('GET', '/api');  
 //var\_dump($client->getResponse()->getContent());  
 $this->assertSame(200, $client->getResponse()->getStatusCode());  
 }  
  
}  
  
  
*/\*\*  
 \* Just a class to emulate testing kernel functional test  
 \* Class NumberGeneratorApiControllerKernel  
 \** ***@package*** *Paco\CustomPacoBundle\Tests\Controller  
 \*/*class NumberGeneratorApiControllerKernel extends Kernel  
{  
 public function \_\_construct()  
 {  
 parent::*\_\_construct*('test', true);  
 }  
  
 public function registerBundles()  
 {  
 return [  
 new FrameworkBundle(),//just we will need it here  
 new PacoNumberGeneratorBundle(),  
 ];  
 }  
  
 protected function configureRoutes(RouteCollectionBuilder $routes)  
 {  
 //$routes->import(\_\_DIR\_\_.'/../../src/Resources/config/routes.yml')->prefix('/api');  
 //$routes->import(\_\_DIR\_\_.'/../../src/Resources/config/routes.xml')->prefix('/api');  
 //$routes->import(\_\_DIR\_\_.'/../../src/Resources/config/routes.yml', '/api');  
 $routes->import(*\_\_DIR\_\_*.'/../../src/Resources/config/routes.xml', '/api');  
 }  
  
  
 protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)  
 {  
 //configuration needed by default by FrameworkExtension  
 $c->loadFromExtension('framework', [  
 'secret' => 'F00',  
 'router' => ['utf8' => true],  
 ]);  
 }  
  
 public function getCacheDir()  
 {  
 return *\_\_DIR\_\_*.'/../cache/'.spl\_object\_hash($this);  
 }  
  
 public function registerContainerConfiguration(LoaderInterface $loader)  
 {  
 // *TODO: Implement registerContainerConfiguration() method.* }  
}

Nada me quedo clavado me da este error:

Uncaught PHP Exception Symfony\Component\HttpKernel\Exception\NotFoundHttpException: "Unable to find the cont

roller for path "/api"

**15.- Dispatching custom events**

Vale, realmente no existe un mecanismo en symfony para override Controllers pero si si nuestro controller esta definido como un servicio, no obstante, para poner las cosas fáciles al usuario final de nuestro bundle, le daremos la oportunidad de cambiar los datos que devuelve nuestro controlador mediante Eventos.

1. Creamos un nuevo Directorio Event y dentro una clase FilterApiResponseEvent

class FilterApiResponseEvent extends EventDispatcher  
{  
 private $data;  
 public function \_\_construct(array $data)  
 {  
 $this->data = $data;  
 }  
 public function getData(): array  
 {  
 return $this->data;  
 }  
 public function setData(array $data)  
 {  
 $this->data = $data;  
 }  
}

1. En nuestro caso el listener acedera a los datos que vamos a devolver en JSON y tb los podrá setear de ahí que hayamos añadido los setters y getters.
2. Ahora vamos al controller y despachamos un evento antes del response y usamos su Data para la respuesta:

use Paco\CustomPacoBundle\Event\FilterApiResponseEvent;  
use Paco\CustomPacoBundle\PacoNumberGenerator;  
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;  
use Symfony\Component\EventDispatcher\EventDispatcherInterface;  
  
class NumberGeneratorApiController extends AbstractController  
{  
 private $pacoNumberGenerator;  
  
 //Event  
 private $eventDispatcher;  
  
 public function \_\_construct(PacoNumberGenerator $pacoNumberGenerator, EventDispatcherInterface $eventDispatcher = null)  
 {  
 $this->pacoNumberGenerator = $pacoNumberGenerator;  
 $this->eventDispatcher = $eventDispatcher;  
 }  
  
 public function index()  
 {  
  
 $data =[  
 'number' => $this->pacoNumberGenerator->getNumber(),  
 ];  
 $event = new FilterApiResponseEvent($data);  
 //because maybe there aren't anyone listen to it.  
 if($this->eventDispatcher){  
 $this->eventDispatcher->dispatch('paco\_number\_generator.filter\_api', $event);  
 }  
  
 return $this->json($event->getData());  
  
 /\*return $this->json(  
 [  
 'number' => $this->pacoNumberGenerator->getNumber(),  
 ]  
 );\*/  
 }  
}

Tb tenemos que modificar el services.yml del bundle ya que ahora el controlador recibe otro parámetro

custom\_paco\_bundle.number\_generator\_api\_controller:  
 class: Paco\CustomPacoBundle\Controller\NumberGeneratorApiController  
 arguments: ['@custom\_paco\_bundle.paco\_number\_generator','@event\_dispatcher']  
 public: true  
 calls:  
 - [ setContainer,["@service\_container"] ]  
 tags:  
 - name: '

se supone que en Xml se puede setear que ese segundo parámetro puede ser nulo pero en yml no se como hacerlo por loque en el test tb cargo el FrameworkBundle

**16.- Event Constants and Event Docs**

En esta sección crearemos una clase final donde declarar el nombre del evento a dispatchear, realmente deberíamos crear una constante en esta clase por cada tipo de evento que lanzaramos, luego añadiremos un trozo de documentación a esta constante que será de ayuda en el autocompletado cuando alguien quiera hacer uso del mismo, asi, creamos la clase:

<?php  
  
  
namespace Paco\CustomPacoBundle\Event;  
  
  
final class PacoNumberGeneratorEvents  
{  
 */\*\*  
 \* Called directly before the Lorem Ipsum API data is returned.  
 \*  
 \* Listeners have the opportunity to change that data.  
 \*  
 \** ***@Event****("Paco\CustomBundle\Event\FilterApiResponseEvent")  
 \*/* const *FILTER\_API* = 'paco\_number\_generator.filter\_api';  
}

Y desde el controlador referenciamos como:

if($this->eventDispatcher){  
 //$this->eventDispatcher->dispatch('paco\_number\_generator.filter\_api', $event);  
 $this->eventDispatcher->dispatch(PacoNumberGeneratorEvents::*FILTER\_API*, $event);  
}

Ahora vamos a hacer una prueba en el sistema principal añadiendo un listener y modificando la salida del controlador de nuestro bundle haciendo uso del sistema de eventos, creamos carpeta EventSubscriber y dentro un subscriber tal que :

<?php  
  
  
namespace App\EventSubscriber;  
  
  
use Paco\CustomPacoBundle\Event\PacoNumberGeneratorEvents;  
use Paco\CustomPacoBundle\Event\FilterApiResponseEvent;  
use Symfony\Component\EventDispatcher\EventSubscriberInterface;  
  
class AddMessageToControllerResponseSubscriber implements EventSubscriberInterface  
{  
 public static function getSubscribedEvents()  
 {  
 return [  
 PacoNumberGeneratorEvents::*FILTER\_API* => 'onFilterApi',  
 ];  
 }  
 public function onFilterApi(FilterApiResponseEvent $event)  
 {  
 $data = $event->getData();  
 $data['message'] = 'Message Added on subscriber!';  
 $event->setData($data);  
 }  
}

Y a correr, cuando invocamos <http://127.0.0.1/api/numbergenerator/> vemos que en la salida ya se añade lo que hemos interceptado desde nuestra app:

|  |  |
| --- | --- |
| number | "Hi outside 202" |
| message | "Message Added on subscriber!" |