Symfony

Desarrollo de aplicaciones MVC en el servidor con un framework PHP

3. El contenedor de servicios y la inyección de dependencias

Ignacio Iborra Baeza
Julio Martínez Lucas

Índice de contenidos

1. Introducción

La **inyección de dependencias** es un concepto muy habitualmente ligado a frameworks de desarrollo web, y hace referencia a un patrón de diseño orientado a relacionar adecuadamente los objetos que componen la aplicación, de forma que se comparten ciertos recursos globales, o se suministran a cada clase que los necesite, en lugar de ser la propia clase quien los cree. Esto favorece el desacoplamiento de nuestra aplicación, al permitir que los elementos que la componen sean más independientes entre sí. Por ejemplo, imaginemos que debemos acceder desde varias clases o ficheros fuente a una base de datos MySQL. Sin la inyección de dependencias, deberíamos crear la conexión a la base de datos en cada una de esas clases o ficheros fuente. Sin embargo, contando con esta característica, algún elemento de la aplicación se encargará de crear la conexión, y facilitarla a los otros elementos que la necesiten.

Para gestionar esta inyección de dependencias, Symfony utiliza un objeto llamado **contenedor de servicios**, que es quien se va a encargar de crear las instancias de todos esos elementos que van a ser compartidos o accedidos desde diferentes lugares del código, y que llamaremos **servicios**. En esta sesión veremos cómo funciona dicho contenedor y cómo podemos añadirle servicios y acceder a ellos desde fuera.

Symfony ya cuenta con una serie de servicios predefinidos, y cada módulo de terceros (*bundle*) que añadamos a la aplicación, incorpora los suyos propios.

1.1. Configuración general de los servicios

De forma general, los servicios se configuran en el archivo **services.yaml** dentro de la carpeta *config* de nuestra aplicación. Si echamos un vistazo a la configuración de inicio, podemos distinguir cuatro apartados:

1. Configuración de parámetros globales para todos los servicios. Aquí definiremos las propiedades que podrán ser accedidas por todos los servicios que utilicemos o creemos. Por ejemplo, se tiene un parámetro *locale* para indicar que la localización de la aplicación emplea el idioma inglés por defecto

```
parameters:
    locale: 'en'
```

2. Después, se tiene la sección de *services*. El primer apartado dentro de esta sección indica la configuración por defecto que tendrán los servicios:

```
services:
    _defaults:
        autowire: true
        autoconfigure: true
        public: false
```

   - *autowire* hace referencia a que los servicios se auto-inyectan, es decir, cuando se pasa como parámetro un servicio a un método indicando el nombre de la clase, automáticamente Symfony crea el objeto correspondiente y lo pasa como parámetro. Por ejemplo, si hacemos algo así:

```
use Psr\Log\LoggerInterface;

class ...
```

```
{
    public function metodo(LoggerInterface $logger)
```

Symfony detectará la clase *LoggerInterface* como un servicio existente, creará una instancia del mismo y la pasará al método como parámetro

- *autoconfigure* indica que los servicios que se crean se registran automáticamente atendiendo a su tipo. Por ejemplo, si creamos una clase que hereda de *Command*, se registrará automáticamente como un comando.
- *public* indica el nivel de visibilidad de los servicios, que por defecto no es público (opción recomendada).

3. A continuación, hay una sección que permite que cualquier cosa que definamos en la carpeta *src* se pueda utilizar como servicio, e inyectarse en otros elementos, a excepción de los elementos indicados en la propiedad *exclude*

```
App\:
    resource: '../src/*'
    exclude: '../src/{DependencyInjection,Entity,Migrations,Tests,Kernel.php}'
```

4. Finalmente, hay un último apartado dedicado a los controladores, para permitir que los servicios se les puedan inyectar como argumentos, aunque no heredemos de ninguna clase base de controlador.

```
App\Controller\:
    resource: '../src/Controller'
    tags: ['controller.service_arguments']
```

1.2. Mostrar servicios actuales

En sesiones anteriores, al hablar del comando *bin/console* que tenemos disponible en cualquier proyecto Symfony, pusimos como ejemplo un comando que muestra todos los servicios que se tienen actualmente disponibles en nuestra aplicación:

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

2. Utilizar servicios

En Symfony existen multitud de servicios ya predefinidos y listos para utilizarse, como por ejemplo un *mailer* para enviar correos electrónicos, o un *logger* para generar mensajes de log de diferente índole (errores, *warnings*, etc). Para utilizarlos, basta con pasar como parámetro al controlador que lo requiera un objeto del tipo correspondiente. Por ejemplo, si queremos utilizar un *logger*, Symfony pone a nuestra disposición el *bundle* Monolog, a través de la clase *LoggerInterface.* Basta con que pasemos un parámetro de este tipo a nuestro controlador para utilizarlo.

Vayamos a nuestro proyecto de contactos, en concreto a la clase *src/Controller/InicioController*, y utilicemos este *logger* para sacar un mensaje con la fecha y hora del acceso a la página de inicio. La clase quedará así:

```
<?php
namespace App\Controller;
...
use Psr\Log\LoggerInterface;

class InicioController extends AbstractController
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    /**
    * @Route("/", name="inicio")
    */
    public function inicio()
    {
        $fecha_hora = new \DateTime();
        $this->logger->info("Acceso el " .
            $fecha_hora->format("d/m/Y H:i:s"));
        return $this->render('inicio.html.twig');
    }
}
?>
```

> NOTA: observa cómo hemos empleado la clase *DateTime* de PHP, anteponiéndole una barra invertida delante para que la reconozca como propia de PHP.

Podríamos haber pasado directamente el objeto *LoggerInterface* al método *inicio*, pero es más habitual pasar los servicios a un constructor de la clase y guardarlos en atributos de la misma, para poder ser utilizados por más de un método.

Si accedemos a la raíz de la aplicación (*http://symfony.contactos*), se generará el correspondiente mensaje de log. Estos mensajes se guardan por defecto en la subcarpeta

var/log, en concreto en el archivo *dev.log* si estamos en modo de desarrollo, o en *prod.log* si estamos en modo producción. Se pueden configurar estos archivos y otras opciones, pero el uso de esta librería no forma parte de los contenidos de este curso, la emplearemos sólo como ejemplo de uso de servicios, y para alguna depuración puntual de algún controlador.

Además del método *info* visto en el ejemplo, existen otros métodos para generar mensajes de mayor o menor prioridad, como por ejemplo *warning, error, critical…* Se puede obtener un listado echando un vistazo al código de LoggerInterface.

3. Crear servicios

Veamos ahora cómo crear nuestros propios servicios. Siguiendo con nuestra aplicación de ejemplo (contactos), vamos a extraer la "base de datos" de contactos a un servicio. Si recordamos, para evitar de momento utilizar una base de datos real, habíamos creado a mano un array de contactos en nuestra clase *src/Controller/ContactoController*. Lo que haremos ahora será definir ese array dentro de un servicio, para poder acceder a él desde cualquier elemento de la aplicación.

Los servicios se pueden crear en cualquier carpeta de *src*, ya que, como hemos visto, cualquier elemento de esta carpeta (salvo unos pocos preconfigurados) automáticamente se define como servicio. Para agruparlos todos, podemos crearlos, por ejemplo, en la subcarpeta *src/Service*. En nuestro caso, vamos a llamar a la clase del servicio *BDPrueba*. Definimos dentro el array, y un método que lo devuelva:

```
<?php

namespace App\Service;

class BDPrueba
{
    private $contactos = array(
        array("codigo" => 1, "nombre" => "Juan Pérez",
            "telefono" => "966112233", "email" => "juanp@gmail.com"),
        array("codigo" => 2, "nombre" => "Ana López",
            "telefono" => "965667788", "email" => "anita@hotmail.com"),
        array("codigo" => 3, "nombre" => "Mario Montero",
            "telefono" => "965929190", "email" => "mario.mont@gmail.com"),
        array("codigo" => 4, "nombre" => "Laura Martínez",
            "telefono" => "611223344", "email" => "lm2000@gmail.com"),
        array("codigo" => 5, "nombre" => "Nora Jover",
            "telefono" => "638765432", "email" => "norajover@hotmail.com"),
    );

    public function get()
    {
        return $this->contactos;
    }
}

?>
```

Nuestra clase *src/Controller/ContactoController* quedará de este modo:

```
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```php
use App\Service\BDPrueba;

class ContactoController extends AbstractController
{
    private $contactos;

    public function __construct(BDPrueba $datos)
    {
        $this->contactos = $datos->get();
    }

    // El resto queda igual
```

Como vemos, lo que se ha hecho es utilizar un constructor para pasarle como parámetro el servicio (objeto de tipo *BDPrueba*), y guardar los datos en un atributo interno de la clase, para poderlo usar tanto en el método *ficha* como en el método *buscar*. Recuerda añadir también la correspondiente línea *use* para utilizar la clase *BDPrueba*.

En este punto, puedes realizar el Ejercicio 1 de los propuestos al final de la sesión.

4. Otras opciones

Ahora que ya sabemos cómo crear servicios propios y utilizarlos, o bien utilizar servicios de Symfony o de *bundles* de terceros, vamos a ver algunas opciones algo más avanzadas que afectan al contenedor de servicios y a los servicios que utilicemos y desarrollemos.

4.1. Combinar servicios

¿Qué ocurriría si en una clase o método determinado necesitamos emplear más de un servicio? Tenemos dos alternativas:

1. Pasar tantos parámetros como servicios se requieran, normalmente al constructor de la clase. Por ejemplo, si necesitamos un objeto de tipo *BDPrueba* y otro de tipo *LoggerInterface* en una misma clase, podemos hacer esto:

```
class MiClase
{
    private $contactos;
    private $logger;

    public function __construct(BDPrueba $datos, LoggerInterface $logger,)
    {
        $this->contactos = $datos->get();
        $this->logger = $logger;
    }
```

2. Como segunda alternativa, también se puede crear una clase que encapsule los objetos necesarios (para no pasarlos por separado como parámetro), y luego utilizar un objeto de esa clase en el constructor. Así, para el mismo ejemplo anterior primero crearíamos una clase que encapsulara un objeto *BDPrueba* y otro *LoggerInterface*…

```
class ServicioCombinado
{
    private $contactos;
    private $logger;

    public function __construct(BDPrueba $datos, LoggerInterface $logger)
    {
        $this->contactos = $datos->get();
        $this->logger = $logger;
    }
    // Definir getters o métodos para acceder a lo que nos interese
```

… y luego lo utilizaríamos en la clase en cuestión:

```
class MiClase
{
    private $servicio;

    public function __construct(ServicioCombinado $servicio)
```

```
    {
        $this->servicio = $servicio;
    }
```

4.2. Argumentos sin “autowiring”

Hemos visto que la opción de configuración *autowiring* existente en el archivo *config/services.yaml* hace referencia al hecho de que, cuando pasamos un objeto de un servicio determinado a un método (indicando el tipo de objeto), Symfony automáticamente crea el objeto por nosotros y se lo pasa al método.

Sin embargo, existen algunos argumentos para los que Symfony no puede aplicar este mecanismo. Por ejemplo, supongamos que en nuestra clase *InicioController* queremos que el formato de fecha para el mensaje de log sea personalizable, y por tanto, se pueda pasar como argumento. La clase quedaría así:

```
...
class InicioController extends AbstractController
{
    private $logger;
    private $formatoFecha;

    public function __construct(LoggerInterface $logger, $formatoFecha)
    {
        $this->logger = $logger;
        $this->formatoFecha = $formatoFecha;
    }

    /**
    * @Route("/", name="inicio")
    */
    public function inicio()
    {
        $fecha_hora = new \DateTime();
        $this->logger->info("Acceso el " .
            $fecha_hora->format($this->formatoFecha));
        return $this->render('inicio.html.twig');
    }
}
```

Pero si intentamos utilizarlo (accediendo a *symfony.contactos*), obtendremos el siguiente mensaje de error:

```
Cannot autowire service "App\Controller\InicioController": argument
"$formatoFecha" of method "__construct()" has no type-hint, you should
configure its value explicitly.
```

En realidad, lo que ha ocurrido es bastante simple: para el primer argumento (*LoggerInterface*), Symfony sabe dónde obtenerlo y cómo construirlo, pero para el segundo, al no estar tipado ni tener un valor preasignado, Symfony no sabe qué hacer con él. Para solucionar el problema, podemos definir argumentos propios de un servicio, es decir, argumentos que va a utilizar un servicio determinado y que no son auto-inyectables. Para ello, editamos el archivo *config/services.yaml* indicando el nombre de la

clase afectada, y los parámetros no auto-inyectables que puede recibir el constructor. En nuestro caso, añadimos estas líneas al final del archivo:

```
App\Controller\InicioController:
    arguments:
        $formatoFecha: 'd/m/Y H:i:s'
```

Recuerda no usar el tabulador para indentar las propiedades (sino cuatro espacios). Lo que hemos hecho ha sido indicar que la clase *InicioController* tendrá un argumento llamado *formatoFecha* en su constructor, cuyo valor por defecto es el indicado.

4.3. Parámetros globales

Es posible también definir parámetros de configuración globales a todos los servicios, en la sección *parameters* del archivo *config/services.yaml*. De hecho, ya tenemos un parámetro global definido que indica la localización o idioma general de la página:

```
parameters:
    locale: 'en'
```

Volviendo al ejemplo anterior, podríamos añadir un nuevo parámetro que indique que el formato de fecha por defecto para cualquier servicio que lo requiera será el visto antes:

```
parameters:
    locale: 'en'
    formato_fecha_defecto: 'd/m/Y H:i:s'
```

Y podríamos utilizar este parámetro en *cualquier* archivo de configuración YAML. Para empezar, lo podemos utilizar más abajo, cuando especificamos el argumento *formatoFecha* para la clase *InicioController*. Ahora haremos que tome su valor del parámetro global, en lugar de ponérselo a mano:

```
App\Controller\InicioController:
    arguments:
        $formatoFecha: '%formato_fecha_defecto%'
```

Podemos acceder a los parámetros globales desde cualquier controlador, siempre que herede de la clase *Controller* (y no *AbstractController*) accediendo a su propiedad *container*:

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class MiController extends Controller
{
    ...
    public function miFuncion()
    {
        $formato = $this->container->getParameter('formato_fecha_defecto');
```

4.4. Asociar argumentos por nombre o por tipo

Para finalizar con este apartado, veremos que se pueden asociar o establecer los argumentos de los servicios tanto por el tipo de argumento (en el caso de que se especifique tipo) como por el nombre del mismo. Para ello, dentro del archivo *config/services.yaml*, y en concreto dentro del apartado *services* > *_defaults*, añadimos una propiedad *bind*, y en ella indicamos tantas asociaciones como queramos.

Veamos un ejemplo con el objeto *logger* que hemos usado para mostrar mensajes de log. Este objeto es de tipo *LoggerInterface*, que en realidad es una interfaz, y cualquier librería que la implemente puede servir como fuente para generar esos archivos de log. Una de ellas es Monolog, y es la que se utiliza por defecto, pero podrían ser otras.

4.4.1. Asociación por nombre

Por ejemplo, podemos hacer que, siempre que un servicio tenga un argumento llamado *logger*, se utilice la librería *Monolog*:

```
services:
    _defaults:
        ...
        bind:
            $logger: '@monolog.logger.request'
```

Si definimos un servicio con un constructor así:

```
private $logger;

public function __construct($logger)
{
    $this->logger = $logger;
}
```

automáticamente al argumento *logger* se le asignará un objeto del tipo *LoggerInterface* de *Monolog*, aunque no especifiquemos el tipo.

4.4.2. Asociación por tipo

Alternativamente, en lugar de usar el nombre del argumento, podemos hacer que, siempre que un servicio intente utilizar la interfaz *LoggerInterface* (la que hemos utilizado en nuestos ejemplos de esta sesión), se emplee por defecto la implementación que hace de ella la librería *Monolog*:

```
services:
    _defaults:
        ...
        bind:
            Psr\Log\LoggerInterface: '@monolog.logger.request'
```

De este modo, siempre que utilicemos un objeto de tipo *LoggerInterface*, se le asociará una instancia del logger de *Monolog* también.

En este punto, puedes realizar el Ejercicio 2 de los propuestos al final de esta sesión.

5. Ejercicios

5.1. Ejercicio 1

Vamos a nuestra aplicación de libros. Crea un servicio llamado *BDPruebaLibros* en la carpeta *src/Service* (que deberás crear). Traslada allí el array de libros que habíamos creado en *src/Controller/LibroController*, y define un método para devolver el array de libros, igual que hemos hecho en el ejemplo de contactos.

Una vez tengas este servicio hecho, utilízalo en dos lugares:

- En la propia clase *src/Controller/LibroController*, para reemplazar lo que había por el uso de este servicio (en lugar de tener el array local, ahora utilizaremos el servicio para acceder al catálogo de libros)
- En la clase *src/Controller/InicioController*, haz que la página de inicio muestre un listado de todos los libros, mostrando para cada uno su título, que será un enlace a la ficha del libro. Deberá quedarte algo así:

  Para ello, deberás modificar la plantilla *inicio.html.twig* para que reciba como parámetro el listado de libros (desde *InicioController::inicio*) y los muestre con el formato indicado.

  Opcionalmente, también puedes añadir un enlace en la plantilla de la ficha del libro para volver a la página de inicio.

5.2. Ejercicio 2

En la aplicación de libros, edita el archivo *config/services.yaml* y añade un *bind* para que, siempre que un servicio reciba un argumento llamado *$bdPrueba*, se le asigne una instancia del servicio *BDPruebaLibros* creado en el ejercicio anterior. Comprueba su funcionamiento en la clase *InicioController*.