

Symfony

*Desarrollo de aplicaciones MVC en el servidor con un
framework PHP*

7. Los bundles de Symfony

Ignacio Iborra Baeza
Julio Martínez Lucas

Índice de contenidos

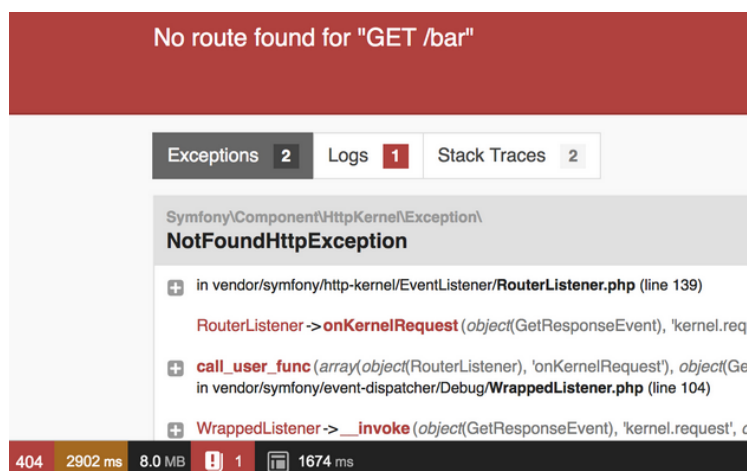
Symfony.....	1
1.Introducción.....	3
1.1.Bundles preinstalados.....	3
2.Instalación de bundles.....	5
2.1.Ejemplo: Doctrine.....	6
2.2.Instalación de otros bundles conocidos.....	8
2.3.Más bundles interesantes.....	9
2.3.1.Ejemplo: EasyAdmin.....	9
2.3.2.Ejemplo: Jens Segers Date.....	10
2.4.Symfony Flex.....	11
2.5.Creación de bundles propios.....	11
3.Ejercicios.....	12
3.1.Ejercicio 1.....	12
3.2.Ejercicio 2.....	12
3.3.Ejercicio 3 (opcional).....	12

1. Introducción

Los bundles en Symfony son elementos conocidos como *módulos* o *plugins* en otros frameworks. Se definen en el archivo `config/bundles.php` para cada entorno (desarrollo, producción, etc), de modo que pueden activarse sólo para ciertos entornos.

```
1 // config/bundles.php
2 return [
3     // 'all' means that the bundle is enabled for any Symfony environment
4     Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
5     Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],
6     Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
7     Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
8     Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle::class => ['all' => true],
9     Doctrine\Bundle\DoctrineBundle\DoctrineBundle::class => ['all' => true],
10    Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
11    // this bundle is enabled only in 'dev' and 'test', so you can't use it in 'prod'
12    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
13 ];
```

El ejemplo más habitual de bundle específico para un entorno es el bundle *profiler*, que se encarga de mostrar la ventana de error y depuración cada vez que se produce un error en la aplicación, mostrando el mensaje y traza del error, usuario autenticado y otros datos que no son recomendables en un entorno de producción. Este bundle sólo está habilitado para el entorno de desarrollo y pruebas, pero no para producción.



1.1. Bundles preinstalados

Si echamos la vista atrás, a la primera sesión del curso, vimos que existían dos formas de crear un proyecto Symfony:

```
composer create-project symfony/website-skeleton nombre-proyecto
```

```
composer create-project symfony/skeleton nombre-proyecto
```

La primera de ellas es la que hemos usado hasta ahora, y deja preinstalado un conjunto de bundles útiles para desarrollar una aplicación web con Symfony. Entre ellos está el motor de plantillas Twig, el ORM Doctrine y otros.

Si optamos por crear un proyecto mediante la segunda opción, obtendremos un proyecto sin (casi) ninguna funcionalidad añadida. Esta forma de definir proyectos se ha habilitado desde Symfony 4 para dar cabida a proyectos más ligeros, donde sólo se instale lo necesario. Sin embargo, como ya comentamos antes, se ha dejado disponible la otra alternativa para desarrollar proyectos de forma cómoda, con muchas funcionalidades preinstaladas, aunque realmente algunas no las vayamos a utilizar.

En cualquier caso, conviene tener presente que todas esas funcionalidades preinstaladas en nuestro proyecto, y otras muchas que podemos requerir, son bundles o *plugins* externos al núcleo de Symfony.

En esta sesión veremos cómo añadir estos bundles a los proyectos, y hablaremos de algunos bundles interesantes además de los que ya hemos visto.

2. Instalación de bundles

Vamos a probar a crear un proyecto básico sin esqueleto web. Por ejemplo, ve a tu carpeta de trabajo (*/home/alumno/symfony*) y crea este proyecto:

```
composer create-project symfony/skeleton prueba_bundles
```

Si echas un vistazo a la estructura del proyecto, verás que en la carpeta *vendor* apenas tiene un par de elementos, mientras que esa misma carpeta en la aplicación de contactos o libros que hemos venido haciendo tiene muchos más.

Vamos a hacer ahora lo mismo que hicimos cuando creamos los proyectos de contactos y libros: configurar Apache para poder acceder al proyecto mediante un *virtual host*. Recordemos los pasos:

1. Edita el archivo */etc/hosts* (con permisos de root) para añadir un nuevo dominio local para nuestra aplicación. Llamaremos a este dominio *symfony.bundles*:

```
127.0.0.1 symfony.bundles
```

2. Ya tendremos un par de pasos hechos de proyectos previos: configurar la carpeta */home/alumno/symfony* con los permisos de acceso necesarios para los proyectos que tendremos dentro, y habilitar la creación de hosts virtuales en Apache. Así que estos pasos podemos saltarlos ahora

3. Edita el archivo */opt/lampp/etc/extra/httpd-vhosts.conf* y añade un nuevo host virtual para nuestra aplicación. Deberá quedarte así:

```
<VirtualHost *:80>
```

```
    DocumentRoot "/home/alumno/symfony/prueba_bundles/public"
```

```
    ServerName symfony.bundles
```

```
</VirtualHost>
```

4. Reinicia el servidor Apache, e intenta acceder a *http://symfony.bundles*, para ver la página de inicio.

Welcome to Symfony 4.1.6



Your application is now ready. You can start working on it at:
`/home/alumno/symfony/prueba_bundles/`

What's next?



Read the documentation to learn
[How to create your first page in Symfony](#)

2.1. Ejemplo: Doctrine

Vamos a hacer una prueba para trabajar con Doctrine en nuestro nuevo proyecto. Para empezar, vamos a crear una base de datos llamada *peliculas* para almacenar cierta información básica de películas. Para crear la base de datos, editamos el archivo *.env* del proyecto y definimos la URL de conexión a la base de datos, como hemos hecho para los proyectos anteriores:

```
DATABASE_URL=mysql://root@127.0.0.1:3306/peliculas
```

y después, creamos la base de datos con el comando:

```
php bin/console doctrine:database:create
```

Lo que obtendremos al ejecutar este comando es un mensaje de error en la consola que indica que no encuentra el comando. Necesitamos instalar el bundle *Doctrine* para solucionar el problema. En realidad, instalaremos un bundle llamado *orm-pack*, que contiene a Doctrine, junto con el bundle *maker* que es el que permite generar cierto código automáticamente, como por ejemplo las entidades. Para ello, escribimos estos comandos desde la carpeta principal del proyecto:

```
composer require symfony/orm-pack
```

```
composer require symfony/maker-bundle --dev
```

Tras estos pasos, editamos el archivo *.env* nuevamente, y definimos la URL de conexión a la base de datos (es posible que se haya duplicado con esta instalación). Después, volvemos a intentar crear la base de datos, y todo funcionará correctamente.

Intentemos ahora crear una entidad. En este caso, vamos a crear una entidad llamada *Pelicula*, con un id autogenerado, un título (string) y un año.

```
php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. OrangePizza):
```

```
> Pelicula
```

```
...
```

```
New property name (press <return> to stop adding fields):
```

```
> titulo
```

```
Field type (enter ? to see all types) [string]:
```

```
> string
```

```
Field length [255]:
```

```
> 255
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
> no
```

Add another property? Enter the property name (or press <return> to stop adding fields):

> anyo

Field type (enter ? to see all types) [string]:

> integer

Can this field be null in the database (nullable) (yes/no) [no]:

> no

Add another property? Enter the property name (or press <return> to stop adding fields):

>







Success!

A continuación, migramos los cambios a la base de datos:

```
php bin/console make:migration
```

```
php bin/console doctrine:migration:migrate
```

Ya tenemos la base de datos creada, con su tabla *pelicula*. Podemos insertar ahora un par de películas de prueba, a mano desde phpMyAdmin:

					id	titulo	anyo
<input type="checkbox"/>		Edit		Copy		Delete	1 Los Vengadores 2012
<input type="checkbox"/>		Edit		Copy		Delete	2 Origen 2010

Definamos ahora un controlador que busque todas las películas y las muestre en la página. Creamos una clase *PeliculaController* en la carpeta *src/Controller*, con un código como este:

```
<?php
```

```
namespace App\Controller;
```

```
use Symfony\Component\HttpFoundation\Response;
```

```
use Symfony\Component\Routing\Annotation\Route;
```

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

```
use App\Entity\Pelicula;
```

```
class PeliculaController extends AbstractController
```

```
{
```

```
    /**
```

```

* @Route("/peliculas", name="lista_peliculas")
*/
public function lista_peliculas()
{
    $repositorio = $this->getDoctrine()->getRepository(Pelicula::class);
    $peliculas = $repositorio->findAll();

    if (count($peliculas) > 0)
    {
        $resultado = "";
        foreach($peliculas as $pelicula)
            $resultado .= $pelicula->getTitulo() . " (" .
                $pelicula->getAnyo() . ")<br />";
        return new Response($resultado);
    } else {
        return new Response("No se han encontrado películas");
    }
}
}

?>

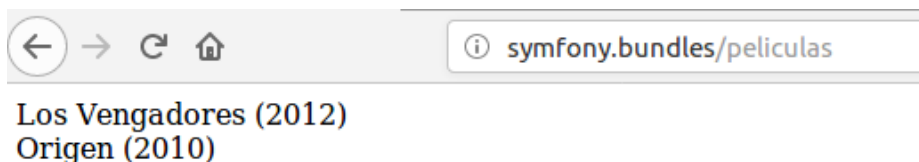
```

Antes de comprobar el funcionamiento, recuerda configurar la reescritura de rutas en la aplicación, escribiendo estos comandos desde la carpeta principal del proyecto:

```
composer config extra.symfony.allow-contrib true
```

```
composer req symfony/apache-pack
```

Después ya podremos acceder a *symfony.bundles/peliculas* y ver el listado en pantalla:



En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

2.2. Instalación de otros bundles conocidos

A lo largo de estas sesiones de curso hemos estado utilizando ciertos bundles de Symfony que también necesitan ser instalados (se instalaron automáticamente a través del *website-skeleton*, y por eso no nos hemos percatado). Hagamos un repaso:

- Para poder utilizar **anotaciones** (como la anotación *@Route* que empleamos para definir las rutas de la aplicación), necesitamos instalar el bundle *annotations*. Este bundle es de los pocos que se instalan en el *skeleton* básico de Symfony. Aún así, para instalarlo manualmente si fuera el caso, deberíamos ejecutar el comando:

```
composer require annotations
```


- Para poder emplear el gestor de **formularios** de Symfony, y crearlos y validarlos mediante código, debemos instalar los bundles *forms* y *validator*, respectivamente, de este modo:

```
composer require symfony/form
```

```
composer require symfony/validator
```

- Para aplicar la infraestructura de **seguridad** de Symfony en nuestra aplicación, y poder definir roles, rutas protegidas, etc, necesitaremos instalar el bundle *security*, así:

```
composer require symfony/security-bundle
```

- Para poder disponer del motor de plantillas **Twig**, necesitaremos instalarlo también:

```
composer require twig-bundle
```

Como decimos, todos estos bundles se preinstalan al crear el proyecto a partir del *website-skeleton*, pero no están disponibles (salvo las anotaciones), si lo creamos a partir del *skeleton* básico. En una sección posterior veremos otros bundles adicionales que puedan resultarnos útiles, además de estos que ya hemos venido usando.

2.3. Más bundles interesantes

Ahora que ya sabemos cómo crear un proyecto casi vacío en Symfony e instalar manualmente los bundles que necesitamos, veamos algunos bundles adicionales que pueden resultar útiles, además de los que hemos comentado ya, y venido usando en sesiones previas (Twig, Doctrine, seguridad...).

Existen distintas webs donde poder obtener información de estos bundles:

- packagist.org, un repositorio de paquetes PHP en general, entre los que podemos encontrar numerosos bundles de Symfony, muchos de ellos desarrollados por el propio equipo de Symfony, entre los que se incluyen los ya vistos (*orm-pack*, *validator*, *twig-bundle*, etc).
- knpbundles.com, una web mantenida por la compañía KNP, que es una de las que más soporte dan a Symfony en cuanto a bundles se refiere. En ella encontraremos tanto bundles desarrollados por dicha compañía, como por otras, como por ejemplo FOS (*FriendsOfSymfony*), un grupo de desarrollo que empezó formando parte del equipo de KNP, pero que luego decidió desarrollar bundles bajo un espacio de nombres propio.

2.3.1. Ejemplo: EasyAdmin

Veamos cómo instalar y trabajar con el bundle *EasyAdmin*. Este bundle permite definir de forma automática un administrador para gestionar las entidades de nuestra aplicación. Su instalación es muy sencilla, a través de este comando:

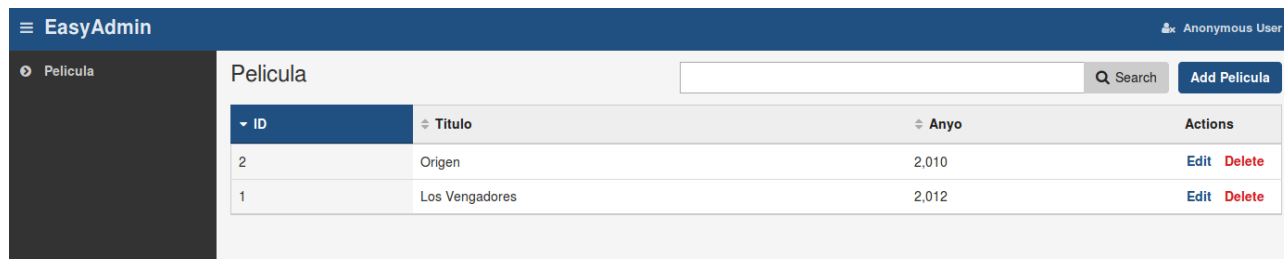
```
composer require admin
```

Después, debemos editar el archivo *config/packages/easy_admin.yaml* que se habrá creado, y añadir las entidades que queramos gestionar:

```
easy_admin:
```

```
entities:
    # List the entity class name you want to manage
    - App\Entity\Pelicula
```

Y eso es todo. Accediendo a la URI `/admin` en nuestra aplicación, podremos gestionar las entidades indicadas. Si seguimos estos pasos en nuestra aplicación *symfony.bundles*, y añadimos la entidad *Pelicula* como en el ejemplo anterior, podemos acceder a *symfony.bundles/admin* y gestionar dicha tabla y sus elementos:



ID	Título	Año	Actions
2	Origen	2,010	Edit Delete
1	Los Vengadores	2,012	Edit Delete

Existen otras opciones de configuración del *bundle*, como por ejemplo proteger el acceso con contraseña y otras opciones. Para más información, podéis consultar la [documentación oficial](#).

En este punto, puedes realizar el [Ejercicio 2](#) de los propuestos al final de la sesión.

2.3.2. Ejemplo: Jens Segers Date

Este otro bundle permite trabajar con fechas de forma sencilla, convirtiendo cadenas a fechas con un formato determinado, estableciendo *locales* para diferentes localizaciones, haciendo cálculos entre fechas, etc.

Se instala con el siguiente comando:

```
composer require jenssegers/date
```

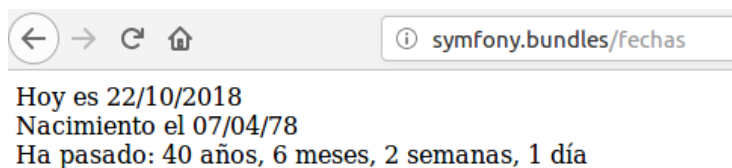
Después, basta con utilizar la clase *Date* para acceder a sus propiedades y métodos. Para probar su uso, crearemos un nuevo método en nuestra clase *PeliculaController*, asociado a la ruta `/fechas`. Dentro, crearemos la fecha actual, que mostraremos con un formato determinado por pantalla. También crearemos una fecha a partir de una cadena de texto, y calcularemos la diferencia entre esa fecha y hoy.

```
use Jenssegers\Date\Date;
...
/**
 * @Route("/fechas", name="fechas")
 */
public function fechas()
{
    Date::setLocale('es');
    $hoy = Date::now();
    $nacimiento = Date::createFromFormat('d/m/Y', '7/4/1978');
    $diferencia = $nacimiento->timespan();

    return new Response('Hoy es ' . $hoy->format('d/m/Y') . '<br />' .
        'Nacimiento el ' . $nacimiento->format('d/m/y') . '<br />' .
        'Han pasado: ' . $diferencia);
}
```

}

La salida será algo parecido a esto (variará dependiendo de la fecha actual):



Intenta realizar el [Ejercicio 3](#) propuesto al final de la sesión, de carácter opcional.

2.4. *Symfony Flex*

Symfony Flex es una nueva forma de instalar y gestionar componentes en proyectos Symfony, disponible desde su versión 3.3. Viene a reemplazar al anterior instalador de Symfony, y al *Symfony Standard Edition*, que, como comentamos en la primera sesión del curso, suponía una instalación mucho más extensa y monolítica del framework.

Con Symfony Flex se automatizan algunas tareas habituales, como instalar o desinstalar bundles. De hecho, desde Symfony 4 es el método que se emplea por defecto para estas instalaciones (aunque su uso sigue siendo opcional), y nos evita tener que editar a mano el archivo `config/bundles.php` para dar de alta los nuevos bundles instalados, o quitar los que ya no estemos empleando. Por este motivo hemos podido emplear Doctrine en un ejemplo anterior sin haber tenido que tocar la configuración del proyecto.

No entraremos en detalles sobre los archivos de configuración y los datos que emplea Flex para gestionar estas tareas, ya que no forma parte del núcleo del curso, pero sí consideramos necesario que se conozca su existencia para saber por qué se autoconfiguran ciertas cosas en los proyectos sin que tengamos que intervenir.

2.5. *Creación de bundles propios*

Además de instalar y utilizar bundles de terceros, también podemos elaborar los nuestros propios. De hecho, hasta la aparición de Symfony 4 se recomendaba que toda la aplicación estuviera estructurada en bundles (es decir, que los propios componentes que desarrolláramos para la aplicación también fueran bundles), pero a partir de esta versión 4 ya no se recomienda esta estructura, y se indica que los bundles se empleen para compartir código entre múltiples aplicaciones. En cualquier caso, para crear nuestro bundle debemos seguir una estructura recomendada desde la documentación oficial de Symfony.

La creación de bundles propios queda fuera de los objetivos principales del curso, por lo que no la veremos con detalle. Además, existen multitud de bundles ya predefinidos que permitirán cubrir la inmensa mayoría de nuestras necesidades. En cualquier caso, [aquí](#) tenéis un punto de partida que seguir para saber más al respecto.

3. Ejercicios

3.1. Ejercicio 1

Vamos a crear un proyecto similar al de pruebas que hemos hecho en el primer ejemplo de esta sesión. En este caso, vamos a crear un proyecto para gestionar tareas pendientes. El comando de creación del proyecto (desde nuestra carpeta `/home/alumno/symfony`) será éste:

```
composer create symfony/skeleton tareas
```

Después, sigue estos pasos:

1. Haz que este nuevo proyecto sea accesible desde el dominio `symfony.tareas`.
2. Instala Doctrine (bundles `orm-pack` y `maker`, como se ha hecho en el ejemplo anterior), y crea y conecta con una base de datos llamada `tareas`, editando el archivo `.env` para ello.
3. Crea una entidad llamada `Tarea` con estos campos:
 - Un `id` autogenerado
 - La *descripción* de la tarea (*string* de 255 caracteres de longitud, sin nulos)
 - La *fecha* de la tarea (tipo `date`, sin nulos)
 - La *prioridad* de la tarea (entero, sin nulos)

Actualiza la base de datos con esta entidad, y crea a mano un par de tareas en ella a través de phpMyAdmin. Procura que las prioridades tengan valores entre 1 (ALTA) y 3 (BAJA), ya que definiremos este rango en la próxima sesión.

4. Crea un controlador en `src/Controller` llamado `TareaController`. Define un método que, al cargarse la URI `/tareas` muestre un listado de las tareas en pantalla, sin un formato específico. Recuerda configurar la reescritura de rutas para que esta URI funcione.

NOTA: para mostrar la fecha como parte de una cadena, puedes utilizar el método `format` del propio objeto `DateTime`:

```
$tarea->getFecha()->format("d/m/Y").
```

3.2. Ejercicio 2

Instala el bundle `EasyAdmin`, configúralo para trabajar con la entidad `Tarea` y crea con él unas cuantas tareas más en la base de datos.

3.3. Ejercicio 3 (opcional)

Utiliza el bundle `jenssegers/date` en la aplicación de tareas, y define un nuevo método en `TareaController` con URI `/tiempos`, que indique para cada tarea de la base de datos cuánto tiempo falta para que finalice el plazo. Por ejemplo:



Acabar curso Symfony: 1 mes, 1 semana, 1 día
Pasar notas Diciembre: 1 mes, 3 semanas, 5 días
Hacer compras navideñas: 1 mes, 3 semanas, 3 días