Symfony

Desarrollo de aplicaciones MVC en el servidor con un framework PHP

2. El patrón MVC en Symfony. Gestión de plantillas con Twig

Ignacio Iborra Baeza
Julio Martínez Lucas

Índice de contenidos

1. El patrón MVC

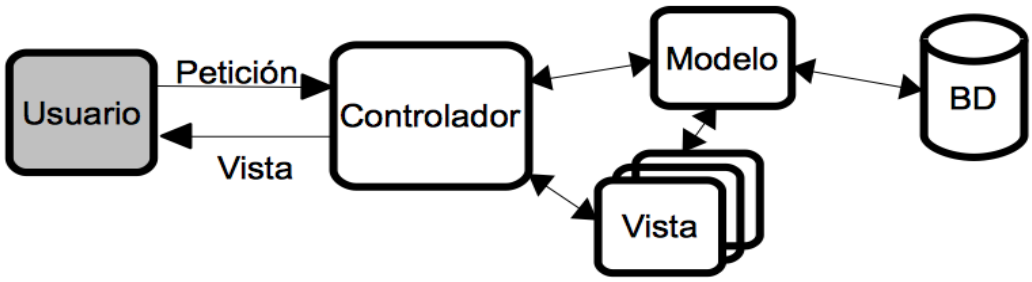
MVC son las siglas de *Modelo-Vista-Controlador* (o en inglés, *Model-View-Controller*), y es el patrón de arquitectura software por excelencia ahora mismo en el mundo de las aplicaciones web, e incluso muchas aplicaciones de escritorio.

Como su nombre indica, este patrón se basa en dividir el diseño o la estructura de una aplicación web en tres componentes fundamentales:

- El **modelo**, que podríamos resumir como el conjunto de todos los datos o información que maneja la aplicación. Típicamente serán variables u objetos extraídos de una base de datos o cualquier otro sistema de almacenamiento, por lo que el código del modelo normalmente estará formado por instrucciones para conectar con la base de datos, recuperar información de ella y almacenarla en algunas variables determinadas. Por tanto, no tendrá conocimiento del resto de componentes del sistema.
- La **vista**, que es el intermediario entre la aplicación y el usuario, es decir, lo que el usuario ve en pantalla de la aplicación. Por lo tanto, la vista la compondrán las diferentes páginas, formularios, etc, que la aplicación mostrará al usuario para interactuar con él.
- El **controlador** (o controladores), que son los fragmentos de código encargados de coordinar el funcionamiento general de la aplicación. Ante peticiones de los usuarios, las recogen, las identifican, y acceden al modelo para actualizar o recuperar datos, y a su vez, deciden qué vista mostrarle al usuario a continuación de la acción que acaba de realizar.

Es un patrón de diseño muy conciso y bien estructurado, lo que le ha valido la fama que tiene hoy en día. Entre sus muchas ventajas, permite aislar el código de los tres elementos involucrados (vista, modelo y controlador), de forma que el trabajo es mucho más modular y divisible, pudiendo encargarse de las vistas, por ejemplo, un diseñador web que no tenga mucha idea de programación en el servidor, y del controlador un programador PHP que no tenga muchas nociones de HTML.

En forma de esquema, podríamos representarlo así:



Las peticiones del usuario llegan al controlador, que las identifica, y se comunica con el modelo para obtener los datos necesarios, y con las vistas para decidir qué vista mostrar a continuación y llenarla con los datos del modelo, para después servírsela al usuario como respuesta. En esta sesión veremos cómo definir controladores en Symfony, y asociarlos a rutas, de forma que muestren algún contenido o vista como respuesta.

2. Controladores y rutas en Symfony

Para crear páginas en una aplicación Symfony se necesitan dos elementos: una ruta (es decir, una URI que indique a qué contenido acceder de la web) y un controlador asociado a ella, que será el encargado de mostrar el resultado (la página) para esa petición de ruta.

2.1. El concepto de ruta en Symfony

Las rutas en Symfony pueden ser tradicionales o amigables. Una ruta tradicional es aquella cuya parte dinámica se especifica en la *query string*. Por ejemplo, si queremos saber la ficha de un contacto a través de su código, tendríamos una URL como esta:

`http://symfony.contactos/contacto.php?codigo=3`

Las rutas amigables son aquellas que separan sus elementos únicamente por barras /, de forma que la parte dinámica de la ruta se intercala entre esas barras. La URL anterior, en forma amigable, quedaría así:

`http://symfony.contactos/contacto/3`

De paso, con las rutas amigables se suele enmascarar el tipo de archivo que se está solicitando, con lo que se omite la información de si es una página PHP, o HTML, o de cualquier otro tipo. Nos centraremos en estas otras rutas en este curso.

2.2. Nuestro primer controlador

Un controlador en Symfony es básicamente una función o método PHP cuyo único propósito es obtener la petición del usuario para una ruta concreta, procesarla y enviarle una respuesta. Normalmente son métodos que se definen dentro de clases, y dichas clases suelen tener el sufijo *Controller* para identificarlas como controladores (aunque esto no tiene por qué ser así).

Los controladores se ubican dentro de la carpeta *src* de nuestro proyecto Symfony, concretamente en el espacio de nombres *Controller* que ya está creado. Por ejemplo, vayamos a nuestra aplicación de contactos (*/home/alumno/symfony/contactos*), e imaginemos que queremos crear un controlador para gestionar el acceso a la raíz de la aplicación. Podemos crear una clase llamada *InicioController* en dicho espacio de nombres, con un método que será el encargado de obtener la petición del usuario y emitir una respuesta. En este caso, lo que haremos será simplemente mostrar un mensaje de bienvenida:

```
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class InicioController
{
```

```
    /**
     * @Route("/", name="inicio")
     */
    public function inicio()
    {
        return new Response("Bienvenido a la web de contactos");
    }
}
?>
```

Observa el código de la clase, y el del método *inicio* en concreto. Simplemente muestra un mensaje de bienvenida a través de un objeto *Response*, que se emplea para definir la respuesta a enviar. Si guardamos los cambios y accedemos a *http://symfony.contactos*, veremos este mensaje de bienvenida.

2.2.1. Definir los espacios de nombres (namespaces)

Si echas un vistazo al código del controlador anterior, verás que comienza con la línea:

```
namespace App\Controller;
```

Lo que estamos haciendo es ubicar la clase (*InicioController*, en este caso) dentro de un espacio de nombres. Cada subcarpeta que hay dentro de la carpeta *src* constituye un espacio de nombres, de forma que cuando ubicamos un archivo fuente dentro de una de esas subcarpetas, debemos indicar que pertenece a dicho espacio de nombres. Todos estos espacios de nombres cuelgan de una raíz *App*, por lo que el espacio de nombres para nuestro controlador es *App\Controller* (los subespacios se separan con barras invertidas).

Los espacios de nombres son útiles en aplicaciones con muchos archivos fuente, como suelen ser las aplicaciones web más o menos importantes, ya que se corre el riesgo de llamar exactamente igual a dos clases que estén en carpetas distintas. Si las agrupamos por espacios de nombres, no habrá problema en llamar igual a las clases y a los archivos. El concepto es similar al de paquete (*package*) en lenguajes como Java. De hecho, en Java tenemos muchos ejemplos de clases que se llaman igual y pertenecen a paquetes diferentes, con lo que son fácilmente diferenciables.

2.2.2. Incluir otros espacios de nombres en el fichero actual

Volvamos de nuevo al código del controlador. Tras definir a qué espacio de nombres pertenecerá dicho controlador (*App\Controller*), podemos necesitar utilizar objetos de otros espacios de nombres. En este caso, por ejemplo, utilizamos un objeto *Response* que pertenece al espacio *Symfony\Component\HttpFoundation*. Para poder utilizar este objeto de forma cómoda y no tener que colocar todo este prefijo cada vez que queramos hacer referencia al tipo *Response*, debemos añadir una instrucción *use* indicando la ruta completa hasta la clase que vamos a utilizar:

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

Después de esto, ya podremos referenciar a la clase *Response* directamente por su nombre en cualquier lugar de este fichero fuente. Del mismo modo, podemos añadir todas

las líneas *use* que consideremos, para hacer referencia a todos los elementos externos que vayamos a necesitar.

También podemos definir un *alias* para nombrar a la clase incorporada dentro de nuestro archivo fuente. Si, en lugar de utilizar *Response*, queremos referenciar a este tipo con una forma más abreviada (*Res*, por ejemplo), haríamos esto:

```
use Symfony\Component\HttpFoundation\Response as Res;
…
return new Res(…);
```

2.2.3. Crear controladores por línea de comandos

Mediante el comando *bin/console* podemos crear un controlador, con la instrucción:

```
php bin/console make:controller NombreControlador
```

Se creará automáticamente un archivo *NombreControlador.php* en la carpeta *src/Controller*, y una plantilla asociada al mismo, en la carpeta *templates* o alguna subcarpeta. Como ventaja destacable de esta forma de crear controladores, nos crea automáticamente el *namespace* y añade los recursos externos (*use*) que necesitemos. Pero, como desventaja, nos crea una plantilla y unas conexiones con ella que normalmente no necesitaremos, y tendremos que retocar. En lo sucesivo, crearemos los controladores de forma manual en este curso, incluyendo nosotros las referencias a otras clases que necesitemos.

2.3. Definiendo rutas

Echemos un vistazo más al código del controlador que hemos hecho. Antes del método *inicio* hay un comentario, en el que se ve una anotación llamada *Route* que está mapeando ese método con una ruta.

```
/**
 * @Route("/", name="inicio")
 */
public function inicio()
...
```

Lo que viene a decir esa anotación es que, cuando accedamos a la raíz de la aplicación, se activará este método y se enviará la respuesta correspondiente (el mensaje de bienvenida). Además, asocia la ruta con un nombre (*name*) llamado “inicio”, lo que nos servirá para hacer que el controlador sea independiente de la ruta, como veremos después.

2.3.1. Otra forma de definir rutas: el archivo “config/routes.yaml”

Existe una forma alternativa de definir rutas sin utilizar anotaciones, que consiste en editar el archivo *config/routes.yaml* y añadir la nueva ruta con el controlador y nombre asociados. Por ejemplo, para el caso anterior, si queremos que al acceder a la raíz de la aplicación se active el método *inicio* del controlador *InicioController*, asignándole a la ruta

el nombre *inicio* (tal y como hemos hecho en el ejemplo anterior), añadiríamos estas líneas al fichero:

```
inicio:
    path: /
    controller: App\Controller\InicioController::inicio
```

Sin embargo, si atendemos a la documentación oficial de Symfony, se recomienda definir las rutas mediante anotaciones, por lo que de ahora en adelante utilizaremos este mecanismo en los apuntes.

2.3.2. Comprobar las rutas de nuestra aplicación

Utilizando la consola de Symfony (archivo *bin/console* de nuestro proyecto) podemos comprobar qué rutas hay actualmente definidas en nuestra aplicación, mediante este comando:

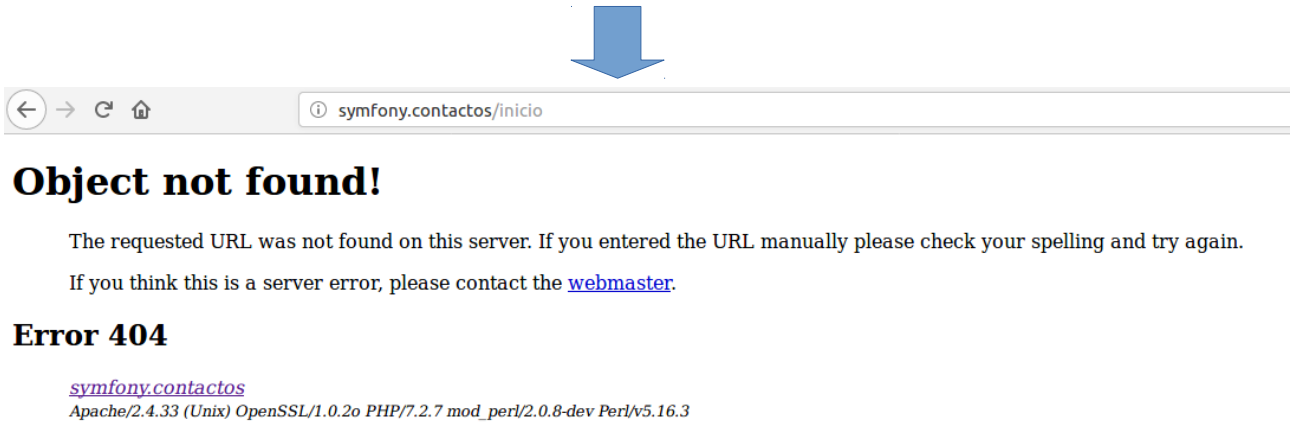
```
php bin/console debug:router
```

Mostrará el listado de rutas, indicando su nombre, y la ruta asociada. Además de nuestra ruta raíz, aparecerán otras rutas creadas por defecto para opciones de depuración y testeo, como por ejemplo las rutas *profiler* para rastrear y obtener detalles de las peticiones realizadas. No entraremos en esos detalles en este curso.

2.3.3. Configurar la reescritura de rutas

Antes de continuar, hay algo que debemos tener en cuenta: el controlador que hemos hecho de prueba (*InicioController::inicio*) funciona porque hace referencia a la raíz de la aplicación. Si cambiamos la ruta por cualquier otra, como por ejemplo */inicio*, no funcionará:

```
/**
* @Route("/inicio", name="inicio")
*/
public function inicio() ...
```



El motivo es que aún no tenemos configurado nuestro proyecto para que reescriba las rutas de forma amigable. Para ello, deberíamos tener un archivo *.htaccess* en la carpeta *public* con los parámetros de configuración de Apache para esa reescritura. Como esa tarea es un tanto manual, la herramienta *composer* pone a nuestra disposición un par de

comandos para hacerlo por nosotros. Basta con colocarnos en la raíz de nuestro proyecto (*/home/alumno/symfony/contactos*, en este caso) y escribirlos:

```
composer config extra.symfony.allow-contrib true
composer req symfony/apache-pack
```

A partir de este punto, ya podremos crear las rutas amigables que queramos en nuestro proyecto. **Recuerda repetir estos comandos en todos los proyectos Symfony que utilicen rutas amigables**.

2.3.4. Rutas con partes variables

Existen algunas rutas que tienen partes variables. Por ejemplo, si quisiéramos mostrar la ficha de un contacto (su nombre, teléfono y e-mail, por ejemplo), a partir de su código, podríamos plantear una ruta como *http://symfony.contactos/contacto/3*, y que el controlador nos mostrara la información del contacto número 3. Pero, si llamamos a esa ruta con otro código, deberá mostrar la información del contacto con ese otro código.

Vamos a crear entonces un nuevo controlador en nuestro espacio de nombres *Controller* dentro de la carpeta *src*. En este caso, llamaremos a la clase *ContactoController*. Definiremos un método llamado *ficha*, que mostrará la ficha del contacto cuyo código le llegue en la ruta, aunque de momento sólo va a mostrar un mensaje con el código del contacto indicado en la ruta:

```php
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ContactoController
{
    /**
     * @Route("/contacto/{codigo}", name="ficha_contacto")
     */
    public function ficha($codigo)
    {
        return new Response("Datos de contacto con código $codigo");
    }
}

?>
```

Observa cómo en la anotación añadimos el código del contacto como un elemento variable, gracias a las llaves (a esta notación se le llama *wildcard*). El método asociado a la ruta debe tener un parámetro con ese mismo dato (el código), de forma que podamos utilizarlo dentro del método. Si ahora accedemos a la URI *symfony.contactos/contacto/3* nos mostrará el mensaje “Datos de contacto con código 3”.

Vamos a mejorar de forma “local” (y un poco de andar por casa) este controlador, para simular que accede a una fuente de datos para obtener la información. Hasta ver en

sesiones posteriores cómo conectar con MySQL, vamos a crearnos nuestro propio array de contactos en la clase *ContactoController*, y haremos que el método *ficha* muestre los datos del contacto cuyo código se haya indicado. El controlador quedará así:

```
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ContactoController
{
    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"),
    );

    /**
     * @Route("/contacto/{codigo}", name="ficha_contacto")
     */
    public function ficha($codigo)
    {
        $resultado = array_filter($this->contactos,
        function($contacto) use ($codigo)
        {
            return $contacto["codigo"] == $codigo;
        });

        if (count($resultado) > 0)
        {
            $respuesta = "";
            $resultado = array_shift($resultado);
            $respuesta .= "<ul><li>" . $resultado["nombre"] . "</li>" .
            "<li>" . $resultado["telefono"] . "</li>" .
            "<li>" . $resultado["email"] . "</li></ul>";
            return new Response("<html><body>$respuesta</body></html>");
        }
        else
            return new Response("Contacto no encontrado");
    }
```

```
}

?>
```

Como vemos, hemos definido un array de contactos, y dentro del método filtramos con *array_filter* aquel cuyo código coincida con el indicado, mostrando sus datos en la respuesta. Observa también cómo podemos incluir código HTML en la propia respuesta, aunque en seguida veremos que hay formas más cómodas de hacer esto.

2.3.5. Añadir requisitos a las *wildcards*

Imaginemos que queremos definir un buscador de contactos, de forma que le pasamos como parte variable de la ruta una parte del nombre, y nos devuelve todos los contactos que coincidan. Así, para la "base de datos" anterior, si accediéramos a la ruta */contacto/Ma* nos mostraría los contactos "Mario Montero" y "Laura Martínez". Definimos un segundo método en nuestra clase *ContactoController* para procesar esta ruta. Quedaría así:

```
/**
* @Route("/contacto/{texto}", name="buscar_contacto")
*/
public function buscar($texto)
{
    $resultado = array_filter($this->contactos,
    function($contacto) use ($texto)
    {
        return strpos($contacto["nombre"], $texto) !== FALSE;
    });

    $respuesta = "";
    if (count($resultado) > 0)
    {
        foreach ($resultado as $contacto)
            $respuesta .= "<ul><li>" . $contacto["nombre"] . "</li>" .
            "<li>" . $contacto["telefono"] . "</li>" .
            "<li>" . $contacto["email"] . "</li></ul>";
        return new Response("<html><body>" . $respuesta . "</body></html>");
    }
    else
        return new Response("No se han encontrado contactos");
}
```

Si intentamos lanzar la URL anterior (*http://symfony.contactos/contacto/Ma*), obtendremos como resultado "Contacto no encontrado", que es el mensaje correspondiente al controlador anterior (la ficha de contacto) cuando no se encontraba el contacto con el código indicado. Es decir, se ha lanzado el controlador equivocado, y el motivo es simple: hemos definido dos rutas a priori diferentes:

- */contacto/{codigo}* para la ficha del contacto
- */contacto/{texto}* para buscar contactos por nombre

Sin embargo, a efectos prácticos, ambas rutas son lo mismo: el prefijo */contacto* seguido de lo que sea. En esta situación, Symfony lanza el primero de los controladores cuya ruta coincida con la indicada (el de la ficha de contacto, en este caso).

Para diferenciar ambas rutas, necesitamos un criterio, y en este caso, el criterio será que la ficha del contacto necesita que el parámetro *codigo* sea numérico. Esto se especifica mediante la propiedad *requirements* y una expresión regular, al definir la ruta de la ficha:

```
/**
* @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
*/
public function ficha($codigo)
...
```

Desde Symfony 4.1, también se puede emplear esta otra notación más abreviada para incluir el requerimiento en el *wildcard*:

```
/**
* @Route("/contacto/{codigo<\d+>}", name="ficha_contacto")
*/
public function ficha($codigo)
```

2.3.6. Añadir valores por defecto a las *wildcards*

En algunas ocasiones, también nos puede interesar dar un valor por defecto a una *wildcard* para que, si en la ruta no se especifica nada, tenga dicho valor por defecto. Esto se consigue asignando un valor por defecto al parámetro asociado en el controlador. En el caso de la ficha del contacto anterior, si quisiéramos que cuando se introduzca la ruta */contacto* (sin código), se mostrara por defecto el contacto con código 1, haríamos esto:

```
/**
* @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
*/
public function ficha($codigo = 1)
```

aunque desde Symfony 4.1 también se puede especificar en la propia anotación, de esta otra forma:

```
/**
* @Route("/contacto/{codigo<\d+>?1}", name="ficha_contacto")
*/
```

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

3. Definiendo plantillas de vistas con Twig

Los ejemplos de controladores vistos hasta ahora quedan algo limitados, porque el diseño brilla por su ausencia. Nos hemos limitado a mostrar un texto plano con los datos para comprobar que el controlador funciona, o en todo caso, a generar un HTML rudimentario en el objeto *Response* para mostrar una lista. Pero si queremos generar una vista más complicada, no es buena idea hacerlo añadiendo los elementos en la cadena de texto para la respuesta. Ahora veremos cómo podemos generar vistas con algo de diseño, gracias al motor de plantillas Twig.

La filosofía de utilizar motores de plantillas como Twig es separar todo lo posible el código PHP de la estructura HTML de la página, de forma que toda la lógica de negocio queda fuera de la vista (en el controlador, normalmente), y en ésta dejamos lo necesario para mostrar el contenido al cliente.

Para ello, lo que suele hacerse es, desde el controlador, acceder al modelo para obtener o modificar los datos necesarios, almacenarlos en variables y pasarle esas variables a las vistas o plantillas, de forma que éstas sólo tengan que encargarse de mostrar esa información con la estructura y diseño adecuados. Aislamos, por tanto, el trabajo del programador por un lado (controlador y modelo), y el del diseñador por otro (vistas)

3.1. Nuestra primera plantilla

Vamos a ver cómo renderizar una plantilla con Twig. En primer lugar, debemos hacer algunos pequeños cambios en el controlador que vaya a usar Twig: haremos que la clase del controlador herede de *AbstractController* (incorporando esta clase de su correspondiente espacio de nombres):

```
namespace App\Controller;

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

class NombreController extends AbstractController
{
```

Esto lo haremos para poder utilizar algunas de las facilidades que nos brinda *AbstractController*, como por ejemplo el método *render* para renderizar vistas. Como hemos comentado en sesiones anteriores, las vistas se almacenan en la carpeta *templates* de nuestro proyecto Symfony, y si empleamos Twig como motor de plantillas, tienen la extensión *.html.twig*. Por ejemplo, vamos a crear una plantilla llamada *inicio.html.twig* en nuestra aplicación de contactos, con este código:

```
<html>
    <body>
        <h1>Contactos</h1>
        <h2>Bienvenido a la web de contactos.</h2>
        <p>Página de inicio</p>
    </body>
```

```
</html>
```

Ahora vamos a modificar la clase *src/Controller/InicioController* para que herede de *AbstractController…*

```
namespace App\Controller;

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

class InicioController extends AbstractController
{
```

… y modificamos también el método *inicio* para que, en lugar de mostrar una respuesta de texto plano, renderice la vista *inicio.html.twig* que acabamos de hacer. Para ello, el código será el siguiente:

```
/**
* @Route("/", name="inicio")
*/
public function inicio()
{
    return $this->render('inicio.html.twig');
}
```

Observa cómo empleamos el objeto $*this* (recuerda, ahora nuestra clase es un subtipo de *AbstractController*) para acceder al método *render* y renderizar la vista que le indiquemos, que automáticamente se buscará desde la carpeta *templates*.

3.2. Plantillas con partes variables

La plantilla anterior no es algo demasiado habitual, ya que únicamente contiene texto estático. Lo normal es que haya alguna parte que varíe, y que le sea proporcionada desde el controlador. Vamos a ver otro ejemplo con la ficha del contacto: vamos a nuestra clase *src/Controller/ContactoController* y hacemos que también herede de *AbstractController*:

```
namespace App\Controller;

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

class ContactoController extends AbstractController
{
```

Vamos a generar una plantilla llamada *ficha_contacto.html.twig* en nuestra carpeta *templates*. Recibirá como parámetro del controlador el contacto con el código indicado (luego veremos cómo), y mostrará en la plantilla sus datos. El código de la plantilla puede quedar así:

```
<html>
    <body>
```

```
        <h1>Ficha de contacto</h1>
        <ul>
            <li><strong>{{ contacto.nombre }}</strong></li>
            <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
            <li><strong>E-mail</strong>: {{ contacto.email }}</li>
        </ul>
    </body>
</html>
```

Empleamos la notación de la doble llave {{ … }} para ubicar variables, que normalmente son datos que esperamos recibir de fuera (del controlador, en este caso). Nos faltaría, en el método *ficha* de *ContactoController*, obtener el contacto deseado (eso ya lo tenemos hecho) y pasárselo a la vista, de este modo:

```
/**
* @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
*/
public function ficha($codigo)
{
    $resultado = array_filter($this->contactos,
    function($contacto) use ($codigo)
    {
        return $contacto["codigo"] == $codigo;
    });

    if (count($resultado) > 0)
        return $this->render('ficha_contacto.html.twig', array(
          'contacto' => array_shift($resultado)
        ));
    else
        return new Response("Contacto no encontrado");
}
```

Como ves, basta con utilizar un array de parámetros en el método *render* para pasarle a la vista todo lo que necesite, identificando cada cosa con el nombre que queramos, y que coincidirá con el que se utilizará en el código de la vista (el parámetro *contacto*, en este caso).

3.3. Estructuras de control en plantillas

La plantilla anterior es un ejemplo para añadir partes dinámicas en el contenido de la misma, pero está algo “coja”: ¿qué pasa si no encontramos el contacto en la lista? En este caso, el controlador se limita a devolver una respuesta de texto plano que dice “Contacto no encontrado”, pero podríamos emplear la misma vista (u otra) para mostrar esta información más “decorada”. Así, el controlador renderizará la misma vista, pasándole un contacto válido o nulo, según el caso:

```
/**
* @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
*/
public function ficha($codigo)
```

```
{
    $resultado = array_filter($this->contactos,
    function($contacto) use ($codigo)
    {
        return $contacto["codigo"] == $codigo;
    });

    if (count($resultado) > 0)
        return $this->render('ficha_contacto.html.twig', array(
            'contacto' => array_shift($resultado)
        ));
    else
        return $this->render('ficha_contacto.html.twig', array(
            'contacto' => NULL
        ));
}
```

y la vista distinguirá si hay o no contacto, para mostrar una u otra información:

```
<html>
    <body>
        <h1>Ficha de contacto</h1>
        {% if contacto %}
            <ul>
                <li><strong>{{ contacto.nombre }}</strong></li>
                <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
                <li><strong>E-mail</strong>: {{ contacto.email }}</li>
            </ul>
        {% else %}
            <p>Contacto no encontrado</p>
        {% endif %}
    </body>
</html>
```

Observa cómo hemos incluido un bloque {% … %}, que son bloques de acción, empleados para definir ciertas sentencias de control (condiciones, bucles) e incluir dentro el código asociado a dicha sentencia.

Del mismo modo, para el controlador de búsqueda de contactos por nombre, podemos crear una nueva vista (por ejemplo, “lista_contactos.html.twig”), que muestre el listado de contactos que reciba ya filtrado del controlador:

```
<html>
    <body>
        <h1>Listado de contacto</h1>
        {% if contactos %}
            {% for contacto in contactos %}
            <ul>
                <li><strong>{{ contacto.nombre }}</strong></li>
                <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
                <li><strong>E-mail</strong>: {{ contacto.email }}</li>
            </ul>
```

```
            {% endfor %}
        {% else %}
            <p>No se han encontrado contactos</p>
        {% endif %}
    </body>
</html>
```

Así, el código del controlador se limitará a filtrar los contactos y pasárselos a la vista:

```
/**
* @Route("/contacto/{texto}", name="buscar_contacto")
*/
public function buscar($texto)
{
    $resultado = array_filter($this->contactos,
    function($contacto) use ($texto)
    {
        return strpos($contacto["nombre"], $texto) !== FALSE;
    });

    return $this->render('lista_contactos.html.twig', array(
        'contactos' => $resultado
    ));
}
```

Observa cómo podemos emplear bucles en las plantillas para recorrer colecciones de datos pasadas desde el controlador.

3.4. Herencia de plantillas

La herencia de plantillas nos permite reaprovechar el código de unas en otras. En realidad, esto es algo muy habitual en el diseño web: que todas las páginas (o varias) de una web compartan la misma cabecera y pie, por ejemplo. Así, podemos definir una estructura o *layout* base en una plantilla, y hacer que otra(s) hereden de ella para rellenar ciertos huecos. Veamos un ejemplo con nuestra web de contactos.

En primer lugar, definiremos la plantilla base. Tenéis un ejemplo en que basaros ya hecho, en el archivo *templates/base.html.twig*, que proporciona un esqueleto que podríamos aprovechar para muchas aplicaciones:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
```

```
</html>
```

Como podemos observar, la parte “rellenable” de la plantilla se define mediante bloques (*blocks*), de forma que en las diferentes subplantillas podemos indicar qué bloques de la plantilla padre queremos rellenar. Por ejemplo, vamos a definir una subplantilla para la página de inicio. Retocamos nuestra plantilla *inicio.html.twig* y la dejamos así:

```
{% extends 'base.html.twig' %}

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Contactos</h1>
    <h2>Bienvenido a la web de contactos.</h2>
    <p>Página de inicio</p>
{% endblock %}
```

Es importante que, si una plantilla hereda de otra, el primer código que haya en esa plantilla (sin contar comentarios previos) sea una instrucción {% extends … %} para indicar que es una herencia. Después, basta con rellenar los bloques cuyo contenido queramos modificar o establecer: en este ejemplo, los bloques *title* y *body*, definidos en la plantilla base, aunque el bloque *title* se podría dejar predefinido como “Contactos” en la plantilla base también.

Del mismo modo, definiríamos las plantillas *ficha_contacto…*

```
{% extends 'base.html.twig' %}

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Ficha de contacto</h1>
    {% if contacto %}
        <ul>
            <li><strong>{{ contacto.nombre }}</strong></li>
            <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
            <li><strong>E-mail</strong>: {{ contacto.email }}</li>
        </ul>
    {% else %}
        <p>Contacto no encontrado</p>
    {% endif %}
{% endblock %}
```

… y *lista_contactos*:

```
{% extends 'base.html.twig' %}

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Listado de contactos</h1>
    {% if contactos %}
        {% for contacto in contactos %}
        <ul>
            <li><strong>{{ contacto.nombre }}</strong></li>
            <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
```

```
            <li><strong>E-mail</strong>: {{ contacto.email }}</li>
        </ul>
        {% endfor %}
    {% else %}
        <p>No se han encontrado contactos</p>
    {% endif %}
{% endblock %}
```

3.4.1. Incluir plantillas dentro de otras

Otra opción interesante, aparte de la herencia, es la de poder incluir una plantilla como parte del contenido de otra. Basta con utilizar la instrucción *include*, seguida del nombre de la plantilla y, si los necesita, sus parámetros asociados. Por ejemplo, podríamos sacar la lista de datos de un contacto a una plantilla llamada *datos_contacto.html.twig*:

```
<ul>
    <li><strong>{{ contacto.nombre }}</strong></li>
    <li><strong>Teléfono</strong>: {{ contacto.telefono }}</li>
    <li><strong>E-mail</strong>: {{ contacto.email }}</li>
</ul>
```

E incluirla tanto en *ficha_contacto…*

```
{% extends 'base.html.twig' %}

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Ficha de contacto</h1>
    {% if contacto %}
        {{ include ('datos_contacto.html.twig', { 'contacto': contacto }) }}
    {% else %}
        <p>Contacto no encontrado</p>
    {% endif %}
{% endblock %}
```

… como en *lista_contactos*:

```
{% extends 'base.html.twig' %}

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Listado de contactos</h1>
    {% if contactos %}
      {% for contacto in contactos %}
        {{ include ('datos_contacto.html.twig', { 'contacto': contacto }) }}
      {% endfor %}
    {% else %}
      <p>No se han encontrado contactos</p>
    {% endif %}
{% endblock %}
```

3.5. Enlaces a rutas y a elementos estáticos

Para finalizar este apartado de edición de plantillas, nos quedan dos aspectos importantes a tratar:

- Cómo incluir contenido estático (hojas de estilo, imágenes… y todo lo que, en general, esté dentro de la carpeta “public” del proyecto)
- Cómo añadir enlaces a otras rutas

3.5.1. Añadir contenido estático en plantillas

Para ilustrar cómo añadir contenido estático en plantillas, vamos a definir en nuestra carpeta *public* de la web de contactos una subcarpeta *css*, y dentro un archivo *estilos.css* (que quedará, por tanto, en *public/css/estilos.css*. Definimos dentro un estilo básico para probar. Por ejemplo:

```
body
{
    background-color: #99ccff;
}
h1
{
    border-bottom: 1px solid black;
}
```

Ahora, vamos a añadir este estilo a nuestra web. Como tenemos un bloque *stylesheets* en nuestra plantilla *base.html.twig*, podemos aprovecharlo e incluir el CSS dentro de dicho bloque, para que lo utilicen todas las subplantillas:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}
            <link href="{{ asset('css/estilos.css') }}" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Del mismo modo, si tuviéramos una imagen en, por ejemplo, *public/imgs/imagen.png*, podríamos añadirla en la plantilla que hiciera falta con algo así:

```
<img src="{{ asset('imgs/imagen.png') }}" />
```

Podemos también emplear rutas absolutas, empleando la instrucción *absolute_url*:

```
<img src="{{ absolute_url(asset('imgs/imagen.png')) }}" />
```

En el caso de archivos Javascript (librerías para la parte cliente, normalmente), se añadirían en el bloque *javascripts* de la plantilla base (o de alguna subplantilla, si se quiere). Por ejemplo, suponiendo que tenemos un archivo *libreria.js* colgando de la subcarpeta *public/js*, haríamos algo así:

```
{% block javascripts %}
    <script src="{{ asset('js/libreria.js') }}"></script>
{% endblock %}
```

3.5.2. Enlazar a otras rutas de la aplicación

Si lo que queremos es definir un enlace a otra ruta o página de nuestra aplicación, en ese caso utilizamos la función *path* para indicar el nombre (*name*) que hayamos asignado a la ruta a la que queremos ir. Por ejemplo, si queremos ir a la ficha de un contacto cuyo código está almacenado en la variable *codigo*, haríamos algo así:

```
<a href="{{ path('ficha_contacto', {'codigo': codigo}) }}">...</a>
```

3.6. Otras características interesantes de Twig

Además de todo lo expuesto durante este apartado, existen otras características interesantes de Twig. Veamos algunas de ellas rápidamente en esta subsección.

3.6.1. Uso de filtros

Cuando mostramos información en una plantilla con la sintaxis de la doble llave {{…}}, podemos emplear **filtros** para procesar la información a mostrar y darle cierto formato. Los filtros en Twig se activan mediante la barra vertical, seguida del filtro a aplicar. Por ejemplo, si queremos mostrar el nombre del contacto en mayúsculas, haríamos algo así:

```
{{ contacto.nombre | upper }}
```

Existen otros filtros útiles, como **lower** (para mostrar la información en minúsculas), o **date**, para formatear fechas con el formato que se quiera:

```
{{ dato_de_tipo_fecha | date("d/m/Y")}}
```

3.6.2. Comentarios

Es posible también añadir líneas de comentarios en las plantillas Twig, mediante la sintaxis {# … #}:

```
{# Esto es un comentario #}
```

3.6.3. Reutilizar contenido de plantillas padre

Hemos visto que podemos sobreescribir el contenido de un bloque (*block*) de una plantilla padre, simplemente definiendo el mismo bloque en la subplantilla hija. Pero también es posible reutilizar el contenido del padre y añadir el propio de la hija, llamando al padre con *parent*. Por ejemplo, imaginemos que, además de los estilos CSS que tengamos definidos en *base.html.twig*, queremos añadir otros particulares para una plantilla, sin perder los del padre. Lo haríamos así:

```
{% block stylesheets %}
    {{ parent() }}
```

```
    <link href="{{ asset('css/otros_estilos.css') }}" rel="stylesheet" />
{% endblock %}
```

3.6.4. Ciclos

La opción *cycle* es muy útil cuando queremos alternar cíclicamente ciertos valores en un bucle. Por ejemplo, para mostrar un listado con 10 filas y estilos de fila alternos, podemos hacer algo así:

```
{% for i in 1..10 %}
    <div class="{{ cycle(['par', 'impar'], i) }}">
        ...
    </div>
{% endfor %}
```

3.6.5. Otras opciones

Existen otras opciones que nos dejamos en el tintero, y podéis consultar en la web oficial de Twig.

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

4. Ejercicios

4.1. Ejercicio 1

Vamos a ir a nuestra aplicación de libros que creamos en la primera sesión (la puedes descargar de las soluciones de la sesión si no la hiciste), y vamos a añadir estos controladores en la carpeta *src/Controller*:

- Una clase llamada *InicioController*, con un controlador llamado *inicio* que estará unido a la ruta raíz ("/"), y con el nombre "inicio" y mostrará como respuesta el mensaje "Biblioteca particular".
- Una clase llamada *LibroController* con un controlador llamado *ficha*, que estará unido a la ruta */libro/{isbn}*, y con el nombre "ficha_libro", donde recibirá como parámetro en la ruta el ISBN del libro (un código alfanumérico). Crea a mano una lista de al menos 3 o 4 libros en esta misma clase, que tengan como campos el ISBN, título, autor y número de páginas. Aquí tienes un ejemplo de libro:

  ```
  array("isbn" => "A111B3", "titulo" => "El juego de Ender", "autor" =>
  "Orson Scott Card", "paginas" => 350)
  ```

  Utiliza esta lista en este controlador para buscar el libro con el ISBN indicado. Si se encuentra, mostrará sus datos en el formato que prefieras, y si no se encuentra, mostrará "Libro no encontrado".

4.2. Ejercicio 2

Vamos a definir las plantillas para los controladores hechos en el ejercicio anterior, y alguna que otra adicional:

- Haz que la plantilla *base.html.twig* del proyecto de libros cargue un archivo CSS ubicado en *public/css/estilos.css*, con estos estilos:

  ```
  body
  {
      margin: 5% 10%;
      background-color: #B28E6D;
  }

  h1, h2, h3
  {
      color: #ffff66,
  }
  ```

- Define una plantilla llamada *inicio.html.twig* que herede de la anterior y muestre este contenido (un H1 y un párrafo):

Haz que esta plantilla se renderice desde el controlador *InicioController::inicio* definido en el ejercicio 1 (en lugar de mostrar el texto plano que se muestra)

- Define otra plantilla llamada *ficha_libro.html.twig* que herede de *base.html.twig* y muestre los datos del libro, con este formato (un H1 y una lista con el título en mayúsculas, autor en cursiva y número de páginas):

Haz que esta plantilla se renderice desde el controlador *LibroController::ficha*, definido en el ejercicio 1 (en lugar de mostrar los datos del libro de forma rudimentaria, como habrás hecho en el ejercicio en cuestión). Recuerda pasar a la plantilla el libro encontrado, o NULL si no hay libro, y mostrar una u otra información en consecuencia.