UD-10: Controladores, rutas y plantillas TWIG

Desarrollo Web en Entorno Servidor

Curso 2020/2021

Repaso: el patrón MVC

- MVC son las siglas de Model-View-Controller y es el patrón de arquitectura software por excelencia en el mundo de las aplicaciones web. Este patrón se basa en dividir la estructura de una aplicación web en tres componentes:
  - **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 BD u otro sistema de almacenamiento.
  - **Vista**, que es el intermediario entre la aplicación y el usuario, es decir, lo que el usuario ve en pantalla de la aplicación. Por tanto, la vista la compondrán las diferentes páginas, formularios, etc.
  - **Controladores**, que son fragmentos de código encargados de coordinar el funcionamiento general de la aplicación. Recogen las peticiones de los usuarios, las identifican y acceden al modelo para actualizar o recuperar datos, y a su vez, deciden qué vista mostrar al usuario.
- Permite programar de forma mucho más modular, pudiendo encargarse de las vistas, un diseñador web sin mucha idea de programación de servidor, y de los controladores, un programador PHP sin muchas nociones de HTML.

Rutas

- Para mostrar páginas en una aplicación Symfony se necesita una **ruta**, que indique a qué contenido acceder, y un **controlador** asociado a ella, que será el encargado de mostrar la página para esa petición de ruta.
- Llamaremos ***ruta tradicional*** a aquella cuya *parte dinámica* se especifica en la *query string*. Por ejemplo, obtener la ficha de un contacto por su código:

  http://localhost/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://localhost/contacto/3

- En este segundo trimestre utilizaremos rutas amigables, con las que, además, se enmascara el tipo de archivo de la página (PHP, HTML, etc.).

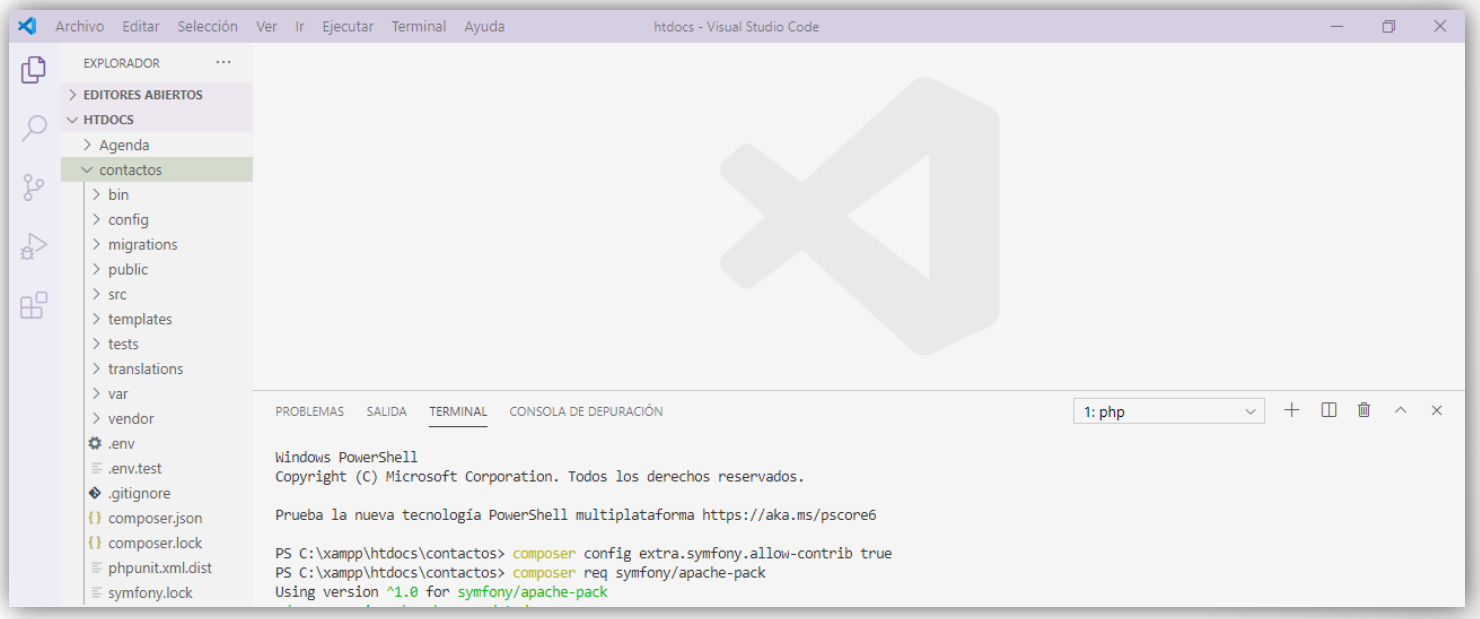
Rutas amigables

- Por defecto nuestro proyecto no está configurado para reescribir las rutas de forma amigable. Por eso, para que funcionen las friendly URL, nos colocamos con el terminal en la raíz de nuestro proyecto (/…/*contactos*) y escribimos:

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

- Resulta muy cómodo hacerlo directamente en el terminal del Visual Studio Code

- Recuerda repetir estos comandos en todos los proyectos Symfony que crees.



Rutas amigables

Hay dos formas de definir rutas en Symfony:

1. Usar anotaciones encima de cada función del controlador
   - Utilizaremos esta forma, ya que es menos engorrosa y más visual.
   - Próximamente, veremos cómo se hace.
2. Editar el archivo `config/routes.yaml`

- Por ejemplo, para que al acceder a la raíz de la aplicación se active el método `inicio` del controlador `InicioController`, añadiremos estas líneas al fichero:

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

Esta opción puede ser engorrosa, pues todos los archivos de configuración .yaml lo son. Tienen una sintaxis rigurosa, por tanto sólo tocaremos archivos .yaml cuando no haya alternativas. De hecho, la propia documentación de Symfony recomienda la otra forma.

Controladores

- Un controlador en Symfony, básicamente, es una función PHP que recibe la petición del usuario, la procesa y envía una respuesta.
- La clase a la que pertenece dicha función, normalmente, tendrá el sufijo *Controller* para identificarla como controlador. Puede haber varios archivos para la parte de Controlador (que contendrán dichas funciones).
- En unidades anteriores, trabajamos solo con un archivo Controller.php. Y, dentro de él, había una función para cada apartado de la web. Ahora, cada apartado seguirá teniendo una función asociada, pero esa función estará en un archivo independiente y que terminará por Controller. Se pueden hacer varios, simplemente agrupándolos por temática. Por ejemplo, *InicioController.php, LibroController.php, LoginController.php*, etc.
- Los controladores están dentro de la carpeta **src** de nuestro proyecto Symfony, concretamente en el espacio de nombres *Controller* que ya está creado. Más adelante, veremos qué es un espacio de nombres.

Nuestro primer controlador

- En nuestra primera aplicación llamada *contactos*, crearemos un controlador para el acceso a la raíz de la aplicación.
- Podemos crear una clase llamada *InicioController.php* en el espacio de nombres ***Controller*** (carpeta **Controller**, en **src**), con un método que será el encargado de obtener la petición del usuario y emitir una respuesta (en este caso, simplemente mostrar un mensaje de bienvenida).

```php
<?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");
        }
    }
?>
```

Esta no es la versión final del controlador *inicio*, ya que no se está llamando a ninguna vista.

Sólo se utiliza Response, que sirve para enviar una respuesta (en este caso, una frase).

Más adelante, mejoraremos este controlador…

Espacio de nombres (namespace)

- Cada subcarpeta dentro de **src**, se dice que constituye un espacio de nombres.
- El ejemplo de controlador anterior, comienza con: `namespace App\Controller;` Aquí se está ubicando la clase *InicioController* dentro del espacio de nombres, lo cual significa que *"esta clase pertenece a la subcarpeta Controller de src"*.
- Los espacios de nombres son útiles en aplicaciones con muchos archivos (aplicaciones web más o menos importantes). Se hace para reducir el riesgo de llamar exactamente igual a dos clases que estén en carpetas distintas. Si las agrupamos por espacios de nombres, no habrá problema.
- El concepto es similar al de paquete (*package*) en lenguajes como Java.

Espacio de nombres (use)

- En el ejemplo anterior de controlador, también aparecía esta línea:

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

- Normalmente, necesitaremos objetos de otros espacios de nombres. Para poder usarlos de forma cómoda, los incluimos con **use.**

  De este modo, podemos hacer luego: `return new Response(…);`

  …en vez de: `return new Symfony\Component\HttpFoundation\Response(…);`

- Podemos usar varias líneas use para incluir todo lo que necesitemos. Podemos también definir un alias para lo que incluimos.

  Por ejemplo, si ponemos: `use Symfony\Component\HttpFoundation\Response as Res;`

  …luego podríamos hacer `return new Res(...);`

Definir rutas

- Continuando con el ejemplo anterior, antes del método **`inicio`** hay un comentario, que contiene una anotación **`Route`** que está mapeando ese método con una ruta:

```
/**
* @Route("/", name="inicio")
*/
public function inicio()
{ … }
```

- Significa que cuando accedamos a la raíz de la aplicación (ruta "/") se ejecutará este método del controlador.
- Además, asocia la ruta con el nombre (`name`) `"inicio"`. Aunque se llaman igual, podrían llamarse de forma diferente.

Definir rutas

- Si tuviéramos, por ejemplo, un apartado de sugerencias en la web, la definición podría ser:

```
/**
* @Route("/sugerencias", name="sugerencias")
*/
public function sugerencias()
{ … }
```

- Entonces, se accedería a dicho apartado en la dirección:

  http://localhost/sugerencias

- Sin embargo, hay rutas que tienen partes variables. Por ejemplo, para mostrar los datos de un contacto a partir de su código, podríamos plantear una ruta como:

  http://localhost/contacto/3 (donde el número final sea variable)

Definir rutas con parte variable

- Vamos a crear un nuevo archivo controlador llamado `ContactoController`.
- En él definiremos un método llamado **`ficha`**, que mostrará la ficha completa del contacto cuyo `código` le llegue en la ruta *(aunque de momento sólo mostraremos 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("Aquí los datos del contacto de código $codigo");
        }
    }
?>
```

Definir rutas con parte variable

- Observa cómo en la anotación hemos presentado el elemento variable de la ruta entre llaves *(a esta notación se le llama **wildcard**)*. La función asociada a la ruta debe recibir un argumento con ese mismo dato *(el código)*, para poder usarlo dentro del método.

- Si ahora accedemos a la URI http://localhost/contacto/3, nos mostrará:

  `"Aquí los datos del contacto con código 3".`

Definir rutas con parte variable con requisito

- Si quisiéramos definir un buscador de contactos que al acceder a `/contacto/Ma` nos muestre los contactos “María Moreno” y “Eva Martín”, haremos lo siguiente:
- Podríamos pensar en escribirlo a continuación:

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

/**
* @Route("/contacto/{texto}", name="buscar_contacto")
*/
public function buscar($texto)
{...}
```

Sin embargo, aquí surge un problema. Si lanzamos `http://localhost/contacto/Ma`, No se ejecutaría **buscar** porque como ambas rutas son ‘iguales’, elige la primera. Este conflicto lo podemos solucionar añadiendo un requisito.

Definir rutas con parte variable con requisito

- Para diferenciar ambas rutas necesitamos un criterio y, en este caso, el criterio será 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) {...}

/**
* @Route("/contacto/{texto}", name="buscar_contacto")
*/
public function buscar($texto) {...}
```

- Ahora, si el parámetro es un número, se ejecutará el controlador **`ficha`**, y si es texto, se ejecutará **`buscar`**.
- Una alternativa más corta es `@Route("/contacto/{codigo<\d+>}", name="ficha_contacto")`

Definir rutas con parte variable con requisito

- También nos puede interesar dar un valor por defecto a una ***wildcard***, de modo que, si no se especifica en la ruta, tenga un valor por defecto.
- 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)
{ … }
```

- Usando forma abreviada, también se puede especificar en la propia anotación:

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

Plantillas de vistas usando TWIG

- En los ejemplos vistos hasta ahora sólo hemos mostrado un texto plano con `Response`. Ahora haremos que el controlador conecte con *la vista*. Para ello, usaremos el motor de plantillas **Twig.**
- **Twig** está basado principalmente en dos conceptos: *bloques* y *herencia*.
- El código de una plantilla Twig se divide en **bloques**:
  - Las plantillas pueden heredar unas de otras.
  - Al heredar de otra plantilla, puedo especificar sólo algunos de sus bloques (los que quiero cambiar)
  - Incluso se pueden definir bloques vacíos, para que las plantillas que hereden los redefinan.
  - También pueden meterse una plantilla dentro de otra.

Plantilla BASE.HTML.TWIG

- Como hemos dicho, las plantillas Twig usan herencia para reaprovechar el código de unas en otras.
- Esto es algo muy habitual en el diseño web, es decir, que varias páginas de una web compartan la misma cabecera y pie, por ejemplo.
- Así, podemos definir una estructura en una plantilla y hacer que otras hereden de ella para rellenar ciertos huecos.
- Normalmente, en un proyecto Symfony todas las plantillas parten de una plantilla llamada **base.html.twig** que ya está en la carpeta **templates**.
- Esta plantilla simplemente proporciona un esqueleto básico que podemos aprovechar en cualquier aplicación.

Plantilla BASE.HTML.TWIG

- Así es el archivo **base.html.twig** (incluido en **templates**):

```
<!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>
```

Plantilla BASE.HTML.TWIG

- Como ves, define sólo el esqueleto básico de etiquetas (<html>, <head>, <body>…).
- Aparecen también unos bloques (*partes rellenables*) que podrán redefinir las plantillas que hereden de esta plantilla **base.html.twig**:
  - En el bloque **title** *(entre `<title>` y `</title>`)* está la palabra `Welcome!` Esto significa que si una plantilla hereda de **base.html.twig** y no redefine dicho bloque title, su título del navegador será `Welcome!`.
  - El bloque **body** está vacío. Quiere decir "Ahí habrá algo, pero ya lo especificarán las plantillas que hereden de mí".
  - Sucede lo mismo con los bloques **stylesheets** (para CSS) y **javascripts** (que no usaremos). Están vacíos, pero pueden llenarlos las plantillas que hereden de **base.html.twig**

Plantilla BASE.HTML.TWIG

- Vamos a hacerle una pequeña modificación al archivo **base.html.twig**, añadiendo un CSS (para no tener que incluirlo en 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>
```

¿Dónde estará el archivo CSS?
En la carpeta **public** creamos una carpeta **css**, y se coloca dentro de ella (public/css/estilos.css)

La función **asset** tiene como objetivo hacer la aplicación más portable (que encuentre el archivo aunque movamos de carpeta el proyecto). Podría ponerse la ruta sin **asset**, pero se recomienda su uso por dicho motivo.

La doble llave {{…}} se utiliza para el contenido dinámico, el cual, vendrá desde el controlador.

¡¡¡Trabajaremos todo esto más adelante!!!

Plantilla INICIO.HTML.TWIG

- En nuestro ejemplo de contactos, definimos una plantilla para la página de inicio. Creamos en la carpeta **templates** un archivo **inicio.html.twig** como este:

```
{% 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 %}
```

En primer lugar, heredamos de la plantilla base.

A continuación, redefinimos dos bloques: *title* y *body*.

Los bloques que no se redefinen (en este caso, *stylesheets* y *javascripts*) se quedan tal cual están en la plantilla de la que heredamos.

Heredar de AbstractController

- Para poder vincular cada archivo de controlador con su vista, usaremos el método `render`. Este método pertenece a la clase **AbstractController**. Por ello, añadiremos **`extends AbstractController`** (y el correspondiente **`use`**):

```php
<?php
    namespace App\Controller;

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

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

Plantillas con partes variables

- En Twig, se utiliza la doble llave {{ … }} para el contenido dinámico, el cual vendrá, desde el controlador, añadido como argumento en la función `render` que conecta el controlador con la vista.
- Así pues, suponiendo que nos llega desde el controlador un objeto ***contacto***, una plantilla **ficha_contacto.html.twig** podría ser así:

```
{% 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 %}
```

El contenido dinámico, que viene desde el controlador, va entre {{ y }}. En este caso, como queremos mostrar campos del objeto que nos ha llegado, ponemos {{contacto.nombre}}, {{contacto.telefono}}…

Pero, si nos hubiera llegado un string llamado fecha, simplemente pondríamos {{fecha}}

Incluir una plantilla dentro de otra

- Otra opción interesante, 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.

```
{% 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 %}
```

- Y la plantilla **datos_contacto.html.twig** quedaría así…

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

Simulando una Base de Datos

- Como no trabajaremos con BD hasta más adelante, en el archivo de controlador **ContactoController.php** declararemos un array que simulará una BD de contactos:

```
<?php
    namespace App\Controller;
    use Symfony\Component\Routing\Annotation\Route;
    use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

    class ContactoController extends AbstractController
    {
        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" => "María Moreno", "telefono" =>
        "965929190", "email" => "maria.mor@gmail.com"),
        array("codigo" => 4, "nombre" => "Eva Martín", "telefono" => "611223344",
        "email" => "em2000@gmail.com"),
        array("codigo" => 5, "nombre" => "Nora Fuentes", "telefono" =>
        "638765432", "email" => "norafuentes@hotmail.com")
        );
```

```php
    /**
    * @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
    */
    public function ficha($codigo)
    {
        $contacto = NULL;
        foreach ($this->contactos as $con){
            if ($con["codigo"] == $codigo)
                $contacto = $con;
        }
        return $this->render('ficha_contacto.html.twig', array('contacto' => $contacto));
    }

    /**
    * @Route("/contacto/{texto}", name="buscar_contacto")
    */
    public function buscar($texto)
    {
        $resultado = array_filter($this->contactos,
                        function($con) use ($texto){
                            return strpos($con["nombre"], $texto) !== false;
                        });
        return $this->render('lista_contactos.html.twig', array('contactos' => $resultado));
    }
}
```

Recuerda que este controlador se ejecutaría al entrar, por ejemplo, en: ***http://localhost/contacto/3***

Simplemente buscamos si hay en el array de contactos uno cuyo código sea 3, y si es así se guarda en la variable $contacto.

https://www.php.net/manual/es/function.array-filter.php

Plantilla LISTA_CONTACTOS.HTML.TWIG

- Veamos ahora la última plantilla de este ejemplo. Resulta interesante conocer cómo se recorre, en Twig, un array que nos llega desde el controlador:

```
{% 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 %}
```

Con {% for elemento in array %} {% endfor %}, recorremos el array que nos haya venido desde el controlador y dentro mostramos cada elemento como nos interese.

Enlaces con PATH

- Es muy probable que en una plantilla pongamos un enlace a otra página de nuestra aplicación. Para ello, se recomienda utilizar la función **path.**
- Imagina que estás en `localhost/contacto/3` y quieres poner un enlace a la página **sugerencias**. En vez de poner `<a href="../../sugerencias">Pulsa aquí</a>`, basta con poner `<a href="{{ path('sugerencias') }}">Pulsa aquí</a>`
- ¡Ojo! Siendo `'sugerencias'` el nombre de la ruta.

```
/**
* @Route("/sugerencias", name="sugerencias")
*/
public function sugerencias()
{
```

- Si estamos en `localhost/sugerencias` y queremos enlazar a `localhost/contacto/3`. En vez de poner `<a href="../contacto/3">Pulsa aquí</a>`, basta con poner `<a href="{{path('ficha_contacto',{'codigo':3})}}">Pulsa aquí</a>`.
- ¡Ojo! Siendo `'ficha_contacto'` el nombre de la ruta.

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

Filtros y comentarios

- 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, mostrar el nombre del contacto en mayúsculas:

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

- También existen otros como **lower**. Podemos encontrar más en internet o en la documentación de TWIG.

- En Twig, pueden incluirse comentarios así:

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

- Para aprender más opciones, puedes consultar en https://twig.symfony.com/

EJERCICIO 1

- Crea un proyecto llamado **libros**.
- Añade los siguientes archivos de controlador en la carpeta **src/Controller**:
  - Una clase llamada **`InicioController`**, con un controlador llamado **`inicio`** que estará asociado a la ruta raíz (“/”).
  - Una clase **`LibroController`**, con un controlador **`ficha`** asociado a la ruta **/libro/{isbn}**, que reciba como parámetro en la ruta el ISBN del libro.
  - Un controlador **`libros`** asociado a la ruta **/libros**. Incluye en la clase este array privado:

```
private $libros = array(
    array("isbn" => "A001", "titulo" => "Jarry Choped", "autor" => "JK
    Bowling", "paginas" => 100),
    array("isbn" => "A002", "titulo" => "El señor de los palillos", "autor"
    => "JRR TolQuien", "paginas" => 200),
    array("isbn" => "A003", "titulo" => "Los polares de la tierra", "autor"
    => "Ken Follonett", "paginas" => 300),
    array("isbn" => "A004", "titulo" => "Los juegos de enjambre", "autor"
    => "Suzanne Collonins", "paginas" => 400)
);
```

- En cuanto a las vistas, habrá una parte que no cambiará (<header></header>, <nav></nav> y <footer></footer>) y otra que sí lo hará, según el apartado en el que estemos (<main></main>).
- Los archivos que deberás crear para las vistas son:

layout.html.twig — heredará de base.html.twig, y definirá esas partes fijas de nuestra web

inicio.html.twig

ficha_libro.html.twig

lista_libros.html.twig

} heredarán de layout.html.twig, y "rellenarán" lo que layout "dejó vacío"

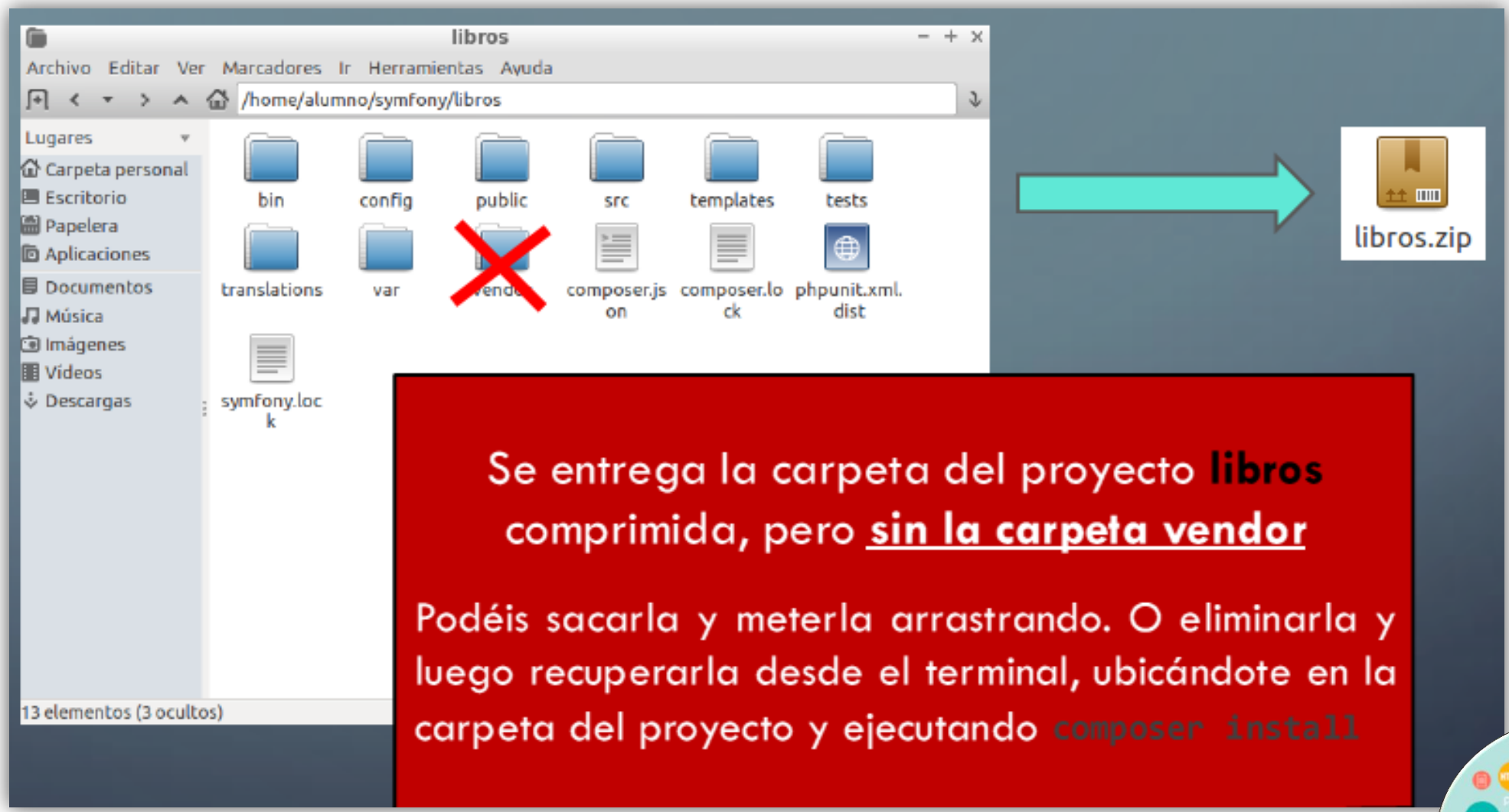
- En cuanto a **base.html.twig** (recuerda que ya viene hecha), sólo hay que tocarla para añadirle el CSS, como hicimos en el ejemplo.

- Esto es lo que aparece en la ruta raíz (/). Se está ejecutando el controlador **`InicioController::inicio`**, y se está mostrando la plantilla **inicio.html.twig**
- El menú de la izquierda (<nav>, está definido en layout.html.twig) son enlaces a **/** (raíz) y **/libros**. Recuerda usar path en los enlaces.

- Esto es lo que aparece en la ruta **/libros**. Se está ejecutando el controlador **`LibroController::libros`**, y se está mostrando la plantilla **lista_libros.html.twig**
- De cada libro sólo se muestra su título, que es un enlace a su ficha (con path):

```
{{ path('ficha_libro',{'isbn':libro.isbn}) }}
```

- Esto es lo que aparece en la ruta **/libro/A001**. Se está ejecutando el controlador `LibroController::ficha,` y se está mostrando la plantilla **ficha_libro.html.twig**
- Se muestra “Libro no encontrado” cuando no existe un libro con ese isbn, como en el ejemplo de contactos, gracias al **if** de Twig que comprueba si el libro es NULL.


libros
Archivo Editar Ver Marcadores Ir Herramientas Ayuda
/home/alumno/symfony/libros
Lugares
Carpeta personal
Escritorio
Papelera
Aplicaciones
Documentos
Música
Imágenes
Vídeos
Descargas
bin
config
public
src
templates
tests
translations
var
composer.json
composer.lock
phpunit.xml.dist
symfony.lock
13 elementos (3 ocultos)
libros.zip
Se entrega la carpeta del proyecto libros comprimida, pero sin la carpeta vendor
Podéis sacarla y meterla arrastrando. O eliminarla y luego recuperarla desde el terminal, ubicándote en la carpeta del proyecto y ejecutando composer install


HTML5
PHP
CSS3
JAVA

ARCHIVOS IMPLICADOS

- libros
  - public
    - css
      - estilos.css
  - src
    - Controller
      - InicioController.php [inicio]
      - LibroController.php [ficha, libros]
  - templates
    - base.html.twig
    - layout.html.twig
    - inicio.html.twig
    - ficha_libro.html.twig
    - lista_libros.html.twig