



# UD-12: Generación y validación de formularios

Desarrollo Web en Entorno Servidor  
Curso 2020/2021



# Introducción

En esta unidad vamos a ver:

- Por un lado, cómo acceder desde el controlador a los datos introducidos en un formulario definido en la parte de la vista. O sea, las instrucciones equivalentes a `$_POST['...'], $_GET['...'],...`
- Por otro lado, veremos cómo definir formularios (directamente en el controlador) asociados a una determinada entidad, para que lo que se envíe en el formulario se asocie a un objeto de dicha entidad. Esto será útil para editar objetos, ya que podremos precargar el formulario con los datos de un objeto ya existente.



## Acceso a datos de un formulario en la vista

- Para acceder a los datos introducidos en un formulario definido en la vista, en el controlador correspondiente se incluye el parámetro (Request \$request).
- Este objeto nos permitirá acceder a las variables globales POST, GET y SERVER:
  - ✓ ->request para acceder a \$\_POST
  - ✓ ->query para acceder a \$\_GET
  - ✓ ->server para acceder a \$\_SERVER
- Ahora, dentro de cualquiera de ellos ->get para acceder a la información. Esto es así simplemente por ser una propiedad de un objeto.

## Acceso a datos de un formulario en la vista

- Por ejemplo. Si en la vista, tenemos este input en un formulario:

```
<input type="text" name="nombre"></input>
```

...podremos acceder desde el controlador a su contenido:

```
public function buscar(Request $request)
{
    ...
    $nombreIntroducido = $request->request->get('nombre');
    ...
}
```

accede a \$\_POST

- La línea en la función equivale a `$nombreIntroducido = $_POST['nombre']`



## Acceso a datos de un formulario en la vista

- El ejemplo de antes, un poco mejorado:

```
public function buscar(Request $request)
{
    ...
    if ($request->server->get('REQUEST_METHOD') == 'POST')
    {
        ...
        $nombreIntroducido = $request->request->get('nombre');
        ...
    }
    ...
}
```

- La primera línea de la función equivale a `if ($_SERVER['REQUEST_METHOD']=='POST')` y la segunda línea equivale a `$nombreIntroducido = $_POST['nombre']`.
- Con esto, podemos hacer todo lo que hicimos en la primera evaluación con formularios (comprobar si se ha enviado, guardar lo que se ha introducido en variables, etc.), pero con un cambio de sintaxis.



# Creación del formulario en el controlador

- Los formularios también pueden crearse en un controlador. Creamos u obtenemos el objeto asociado al formulario y cargamos un formulario con él.
- Veamos un ejemplo en nuestra aplicación de contactos. Crearemos un nuevo controlador que responda a la URI /contacto/nuevo y que muestre el formulario:

```
namespace App\Controller;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class ContactoController extends AbstractController
{
    /**
     * @Route("/contacto/nuevo", name="nuevo_contacto")
     */
    public function nuevo()
    {
        $contacto = new Contacto();
        $formulario = $this->createFormBuilder($contacto)
            ->add('nombre', TextType::class)->add('telefono', TextType::class)
            ->add('email', TextType::class)
            ->add('save', SubmitType::class, array('label' => 'Enviar'))
            ->getForm();
        return $this->render('nuevo.html.twig',
            array('formulario' => $formulario->createView()));
    }
}
```

A través del form builder de Symfony se crea el formulario. Después, se añaden tantos campos como atributos tenga la entidad (normalmente), asociando cada atributo con su campo por el nombre.

En cada campo especificamos también de qué tipo es. En nuestro caso, hemos definido tres cuadros de texto (TextType) para el nombre, teléfono y email, y un botón de submit (SubmitType) para poder enviar el formulario.



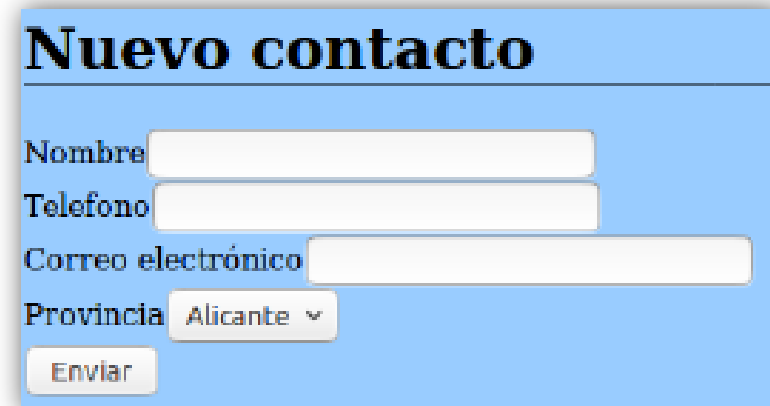
## Creación del formulario en el controlador

- Algunos tipos de campos que pueden resultarnos interesantes son:
  - **TextType** (cuadros de texto de una sola línea, como el ejemplo anterior)
  - **TextareaType** (cuadros de texto de varias líneas)
  - **EmailType** (cuadros de texto de tipo email)
  - **IntegerType** (cuadros de texto para números enteros)
  - **NumberType** (cuadros de texto para números en general)
  - **PasswordType** (cuadros enmascarados para passwords)
  - **DateType** (para fechas)
  - **CheckboxType** (para checkboxes)
  - **RadioType** (para botones de radio o radio buttons)
  - **HiddenType** (para controles ocultos)
  - **EntityType** (desplegables para elegir valores vinculados a otra entidad)
- Puedes consultar [aquí](#) un listado más detallado de los tipos de campos disponibles.

## Etiquetas personalizadas

- Podemos especificar un tercer parámetro en el método **add**: un array de propiedades del control en cuestión. Una de ellas es la propiedad **label**, que nos permite especificar el texto que tendrá asociado el control. Si no se especifica, se coge el nombre del atributo correspondiente en la entidad.
- Para el email, por ejemplo, podemos cambiarlo por un texto personalizado así:  

```
->add('email', EmailType::class, array('label' => 'Correo electrónico'))
```
- Con esto estamos diciendo “*aunque el input es para el campo llamado email, quiero que delante ponga Correo electrónico*”.



**Nuevo contacto**

Nombre

Telefono

Correo electrónico

Provincia





# Mejorando nuestro formulario

- Aprovechando la variedad de tipos de campos, mejoremos nuestro formulario:
  - ✓ Por un lado, el campo email podemos indicar que sea de tipo EmailType.
  - ✓ Por otro lado, para la provincia que añadiremos en la sesión de Doctrine, podemos usar un EntityType que tome sus datos de la entidad Provincia.

```
...
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
...
public function nuevo()
{
    $contacto = new Contacto();
    $formulario = $this->createFormBuilder($contacto)
        ->add('nombre', TextType::class)
        ->add('telefono', TextType::class)
        ->add('email', EmailType::class)
        ->add('provincia', EntityType::class,
            array('class' => Provincia::class, 'choice_label' => 'nombre'))
        ->add('save', SubmitType::class, array('label' => 'Enviar'))
        ->getForm();

    return $this->render('nuevo.html.twig',
        array('formulario' => $formulario->createView()));
}
```

Por defecto, el índice del array de cada elemento en el desplegable se utiliza como el texto que es mostrado al usuario.




## Renderizando el formulario

- El código del controlador anterior terminará renderizando una vista llamada `nuevo.html.twig` que muestre el formulario. Podría quedar así:

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

{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Nuevo contacto</h1>
    {{ form_start(formulario) }}
    {{ form_widget(formulario) }}
    {{ form_end(formulario) }}
{% endblock %}
```

- Las tres líneas bajo el encabezado `h1` son las responsables de la renderización del formulario, a partir del parámetro `formulario` que pasaremos desde el controlador.
- Ahora en `/contacto/nuevo` veríamos →



The screenshot shows a web form titled "Nuevo contacto" in a light blue box. It contains four input fields: "Nombre", "Telefono", "Email", and "Provincia". The "Provincia" field is a dropdown menu currently showing "Alicante". Below the fields is a button labeled "Enviar".

## Rellenando los campos del formulario

- Volvamos a nuestro controlador **nuevo**. Si construimos un objeto relleno con algunos datos, veremos cada dato en su campo asociado:

```
public function nuevo()  
{  
    $contacto = new Contacto();  
    $contacto->setNombre("Nacho");  
    $contacto->setTelefono("112233");  
    $contacto->setEmail("nacho@email.com");  
  
    $formulario = $this->createFormBuilder($contacto)  
        ...  
        ...  
}
```

- Esto es muy útil para editar objetos.



The screenshot shows a web form titled "Nuevo contacto" with a light blue background. It contains four input fields: "Nombre" with the value "Nacho", "Telefono" with the value "112233", "Email" with the value "nacho@email.com", and "Provincia" with a dropdown menu showing "Alicante". Below these fields is a button labeled "Enviar".

## Envío de datos

- Por defecto, el formulario se envía por POST a la misma URI que lo genera (en nuestro caso, a /contacto/nuevo).
- Para gestionar el envío de estos datos, hay que hacer algunos cambios en nuestro controlador ya que, ahora mismo, se volverá a cargar la vista del formulario, pero no habremos insertado nada.
  - En primer lugar, el controlador recibirá un objeto **Request**, que contendrá los datos del formulario enviado (en el caso de que se haya enviado).
  - En segundo lugar, dentro del código del controlador, debemos procesar esos datos (si los hay), validarlos (esto lo veremos a continuación) y si son válidos, realizar la correspondiente inserción o modificación.
  - Finalmente, podemos redirigir a otra ruta en caso de éxito, o volver a renderizar el formulario si hubo algún error o no se ha enviado.

# Envío de datos



Arriba habría que incluir la línea `use Symfony\Component\HttpFoundation\Request;`

```
/**
 * @Route("/contacto/nuevo", name="nuevo_contacto")
 */
public function nuevo(Request $request)
{
    $contacto = new Contacto();
    $formulario = $this->createFormBuilder($contacto)
        ->add('nombre', TextType::class)->add('telefono', TextType::class)
        ->add('email', EmailType::class)->add('provincia', EntityType::class,
            array('class' => Provincia::class, 'choice_label' => 'nombre'))
        ->add('save', SubmitType::class, array('label' => 'Enviar'))
        ->getForm();

    $formulario->handleRequest($request);

    if ($formulario->isSubmitted() && $formulario->isValid())
    {
        $contacto = $formulario->getData();

        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($contacto);
        $entityManager->flush();

        return $this->redirectToRoute('inicio');
    }

    return $this->render('nuevo.html.twig', array('formulario' => $formulario->createView()));
}
```

getData()  
devuelve los  
datos del  
formulario en  
un array,

Toma los datos POST, los procesa y ejecuta las validaciones

Si probamos ahora a cargar el formulario y realizar una inserción, nos enviará a la página de inicio, y podremos ver el nuevo contacto presente en la tabla correspondiente de la base de datos.



## Modificación de datos

- Ahora, haremos una modificación de un contacto existente. El funcionamiento es muy similar, pero con un pequeño cambio: la ruta del controlador recibirá como parámetro el código del contacto a modificar. Además, buscaremos el contacto y lo cargaremos en el formulario, incluyendo su id en un campo oculto.

```
...
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
...
/**
 * @Route("/contacto/editar/{codigo}", name="editar_contacto", requirements={"codigo"="\d+"})
 */
public function editar(Request $request, $codigo)
{
    $repositorio = $this->getDoctrine()->getRepository(Contacto::class);
    $contacto = $repositorio->find($codigo);
    $formulario = $this->createFormBuilder($contacto)
        ->add('id', HiddenType::class)->add('nombre', TextType::class)
        ->add('telefono', TextType::class)->add('email', EmailType::class)
        ->add('provincia', EntityType::class, array('class' => Provincia::class,
                                                    'choice_label' => 'nombre',))
        ->add('save', SubmitType::class, array('label' => 'Enviar'))
        ->getForm();

    $formulario->handleRequest($request);
    ...
}
```

## Modificación de datos

- Para que el código anterior funcione, debemos añadir a nuestra entidad **Contacto** un setter para poder asignarle un id. El motivo es que el método *getData()* trata de asignar mediante los correspondientes *setters* cada campo del formulario al atributo correspondiente de la entidad:

```
class Contacto
{
    ...
    public function getId()
    {
        return $this->id;
    }

    public function setId(int $id): self
    {
        $this->id = $id;

        return $this; //devuelve el objeto modificado
    }
    ...
}
```

- Ahora, en [symfony.contactos/contacto/editar/1](#) se carga el formulario con los datos del contacto cuya id es 1. Al enviarlo, se modifican los campos que hayamos cambiado y se carga la página de inicio.



## Validación de formularios

- Ahora veremos un último paso: la validación de datos de dichos formularios previa a su envío.
- En el caso de Symfony, la validación no se aplica al formulario, sino a la entidad subyacente (es decir, a la clase **Contacto** o **Libro**, por ejemplo).
- Por tanto, la validación la obtendremos añadiendo una serie de restricciones o comprobaciones a estas clases. Por ejemplo, para indicar que el nombre, teléfono y email del contacto no pueden estar vacíos, añadimos estas anotaciones en los atributos de la clase.
- A continuación modificaremos el archivo `src/Entity/Contacto.php` insertando unas aserciones que van a repercutir directamente sobre el código HTML del formulario, donde añadiremos el atributo **required** para que se validen los datos en el cliente.





# Validación de formularios

```
...
use Symfony\Component\Validator\Constraints as Assert;
...

/**
 * @ORM\Entity(repositoryClass="App\Repository\ContactoRepository")
 */
class Contacto{
    ...
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     */
    private $nombre;
    /**
     * @ORM\Column(type="string", length=15)
     * @Assert\NotBlank()
     */
    private $telefono;
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     */
    private $email;
    ...
}
```



# Validación de formularios

- En el caso del email, además, podemos especificar que queremos que sea un email válido con:

```
/**  
 * @ORM\Column(type="string", length=255)  
 * @Assert\NotBlank()  
 * @Assert\Email(message="El email {{ value }} no es válido")  
 */  
private $email;
```

- Estas funciones de validación admiten una serie de parámetros útiles. Uno de los más útiles es **message**, que se usa para determinar el mensaje de error para el usuario cuando el dato no sea válido. Sin embargo, como hemos definido el campo email de tipo **EmailType**, realmente no llegaremos a ver este mensaje de error, porque se activará antes la validación HTML5 en el cliente. Para poderlo comprobar, podemos dejarlo definido como **TextType**.

The screenshot shows a web form titled "Nuevo contacto" with the following fields and values:

- Nombre: aaaa
- Telefono: 12121212
- Email: cccc
- Provincia: Alicante (dropdown menu)
- Enviar: button

Below the Email field, there is a red error message: "El email "cccc" no es válido".



## Creación de clases para formularios

- Hemos visto cómo definir formularios directamente en los controladores afectados. En la aplicación de contactos, hemos definido un formulario para la ruta `/contacto/nuevo` y otro para la ruta `/contacto/editar`.
- Lo recomendado, según la documentación de Symfony, es no ubicar los formularios directamente en los controladores, sino crearlos en una clase aparte. De esta forma, podríamos reutilizar los formularios en varios controladores, y no repetir código innecesariamente.
- Veamos cómo quedaría esta clase para el formulario de inserción y edición de contactos. Podríamos crear el formulario donde queramos, pero por unificar criterios, y siguiendo los ejemplos de la documentación de Symfony, crearemos una carpeta **Form** en nuestra carpeta **src**, y pondremos dentro los formularios (dentro de la carpeta **Form** creamos el archivo **ContactoType.php**). Por tanto, crearemos una clase llamada **ContactoType**, que heredará de una clase base genérica de Symfony llamada **AbstractType**. Dentro, definiremos el método **buildForm** que se encargará de crear el formulario, como hacíamos antes en el método **nuevo** o en **editar**:



# Creación de clases para formularios

<?php

```
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use App\Entity\Provincia;

class ContactoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('id', HiddenType::class)->add('nombre', TextType::class)
            ->add('telefono', TextType::class)->add('email', EmailType::class)
            ->add('provincia', EntityType::class, array('class' => Provincia::class,
                                                    'choice_label' => 'nombre',))
            ->add('save', SubmitType::class, array('label' => 'Enviar'));
    }
}
```

?>



## Creación de clases para formularios

- Ahora, sólo nos queda reemplazar el código de crear el formulario en nuevo y editar por el uso de esta nueva clase:

```
public function nuevo(Request $request)
{
    $contacto = new Contacto();
    $formulario = $this->createForm(ContactoType::class, $contacto);
    $formulario->handleRequest($request);

    if ($formulario->isSubmitted() && $formulario->isValid())
        ...
}

public function editar(Request $request, $codigo)
{
    $repositorio = $this->getDoctrine()->getRepository(Contacto::class);
    $contacto = $repositorio->find($codigo);

    $formulario = $this->createForm(ContactoType::class, $contacto);
    $formulario->handleRequest($request);

    if ($formulario->isSubmitted() && $formulario->isValid())
        ...
}
```

## Añadiendo estilo a los formularios

- Los formularios que hemos generado en esta unidad carecen de estilos CSS propios. Si quisiéramos añadir CSS a estos formularios, tenemos varias opciones.
- Una opción rudimentaria consiste en añadir clases html (atributo class) a los controles del formulario para dar estilos personalizados. Después, en nuestro CSS, bastaría con indicar el estilo para la clase en cuestión.
- Además, Symfony ofrece diversos temas (themes) que podemos aplicar a los formularios (y webs en general) para darles una apariencia general tomada de algún framework conocido, como *Bootstrap* o *Foundation*. Si queremos optar por la apariencia de *Bootstrap*, debemos hacer lo siguiente:
  1. Incluir la hoja de estilos CSS y el archivo Javascript de Bootstrap en nuestras plantillas. Una práctica habitual es hacerlo en la plantilla **base.html.twig** para que cualquiera que herede de ella adopte este estilo. Para ello, en la sección *stylesheets* debemos añadir el código HTML que hay en la documentación oficial de *Bootstrap* para incluir su hoja de estilo, y en la sección *javascripts*, los enlaces a las diferentes librerías que se indican en la documentación de *Bootstrap* también.



# Añadiendo estilo a los formularios

```
<!DOCTYPE html>
<html>
...
{% block stylesheets %}
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn..."...>
    <link href="{ asset('css/estilos.css') }}" rel="stylesheet" />
{% endblock %}
...
{% block javascripts %}
    <script src="..."></script>
    <script src="..."></script>
    <script src="..."></script>
{% endblock %}
</body>
</html>
```

Puedes encontrar los citados enlaces en:  
<https://getbootstrap.com/docs/4.0/getting-started/introduction/>



## Añadiendo estilo a los formularios

2. Editar el archivo de configuración `config/packages/twig.yaml` e indicar que los formularios usarán el tema de *Bootstrap* (en este caso, Bootstrap 4):

```
twig:
  ...
  form_themes: ['bootstrap_4_layout.html.twig']
```

Existen otras alternativas, como por ejemplo no indicar esta configuración general en `config/packages/twig.yaml` e indicar formulario por formulario el tema que va a tener:

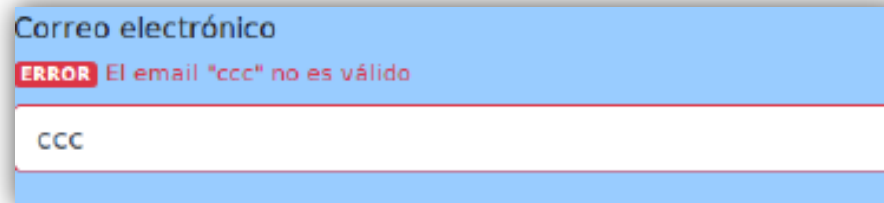
```
{% form_theme formulario 'bootstrap_4_layout.html.twig' %}
{{ form_start(formulario) }}
```

Hay también otros temas disponibles. Podemos consultar más información [aquí](#).



## Añadir estilo para las validaciones

- En el caso de las validaciones de datos del formulario, también podemos definir estilos para que los mensajes de error que se muestran (parámetro message o similares en las anotaciones de la entidad) tengan un estilo determinado.
- Esto se consigue fácilmente eligiendo alguno de los temas predefinidos de Symfony. Por ejemplo, eligiendo **Bootstrap**, la apariencia de los errores de validación queda así automáticamente:



The screenshot shows a form field with the label "Correo electrónico". Below the field, there is a red error message: "ERROR El email 'ccc' no es válido". The field itself contains the text "ccc". The entire form is styled with a light blue background and a red border around the input field.

- Estamos hablando de las validaciones en el servidor, ya que las que se efectúan desde el cliente por parte del propio HTML5 no tienen un estilo controlable desde Symfony. Podríamos desactivar esta validación para que todo corra a cargo del servidor, si fuera el caso. Para ello, basta con añadir lo siguiente:

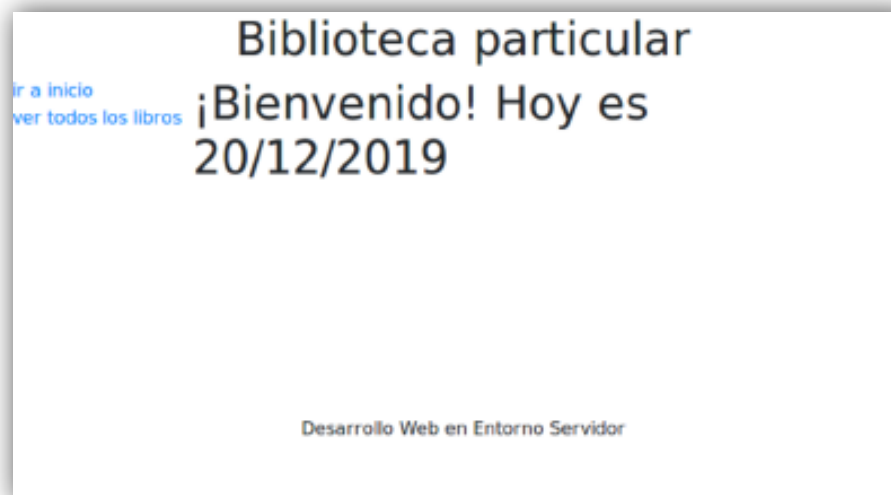
```
{{ form_start(formulario, {'attr': {'novalidate': 'novalidate'}}) }}
```

...

## EJERCICIO 1



- Antes de nada, incluye en **base.html.twig** las líneas de *Bootstrap* como se ha indicado anteriormente (alguna opción de las diapositivas 23-24).



- Para esta práctica, seguramente tengas que incluir estos **use** en **LibroController**:

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\EntityType;
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
use Symfony\Component\HttpFoundation\Request;
```



- Crea un apartado para insertar libros, cuyo controlador se llame **nuevo\_libro** y que se vincule a la ruta **/nuevo** y muestre la vista **nuevo\_libro.html.twig**.

En este caso, se creará el formulario directamente desde el controlador, en base a un libro vacío, con estos campos:

- ✓ **isbn**, con el ISBN del libro (de tipo `TextType`)
  - ✓ **titulo**, con el título del libro (de tipo `TextType`)
  - ✓ **autor**, con el autor del libro (de tipo `TextType`)
  - ✓ **paginas**, con el número de páginas del libro (de tipo `IntegerType`)
  - ✓ **editorial**, con las editoriales actuales de la base de datos (de tipo `EntityType`)
- Tras comprobar si se ha enviado el formulario, recogerá los datos del envío, los asignará a un objeto libro y hará la pertinente inserción, redirigiendo a la página donde se ven todos los libros tras ello. Si no se ha enviado el formulario, o no es válido, se seguirá en la vista **nuevo\_libro.html.twig**

## Biblioteca particular

### Nuevo libro

[Ir a Inicio](#)  
[Ver todos los libros](#)  
[Buscar libros](#)  
[Insertar libro](#)

Isbn

Título

Autor

Páginas

Editorial

Desarrollo Web en Entorno Servidor



- Si todo está relleno, se envía, se inserta y se redirige a la lista de todos los libros.



- Crea un apartado para buscar libros, cuyo controlador se llame **buscar** y que se vincule a la ruta **/buscar**. Dentro habrá un formulario creado en el archivo de la vista **buscar\_libros.html.twig**.
- Si se envía, el controlador recibirá lo que se ha introducido (con el objeto **\$request**) y buscará los libros cuyo título contenga dicha cadena, mostrándolos debajo:

Biblioteca particular

## Buscar Libros

[ir a inicio](#)  
[ver todos los libros](#)  
[buscar libros](#)  
[insertar libro](#)

Introduce una cadena:

Desarrollo Web en Entorno Servidor

Biblioteca particular

## Buscar Libros

[ir a inicio](#)  
[ver todos los libros](#)  
[buscar libros](#)  
[insertar libro](#)

Introduce una cadena:

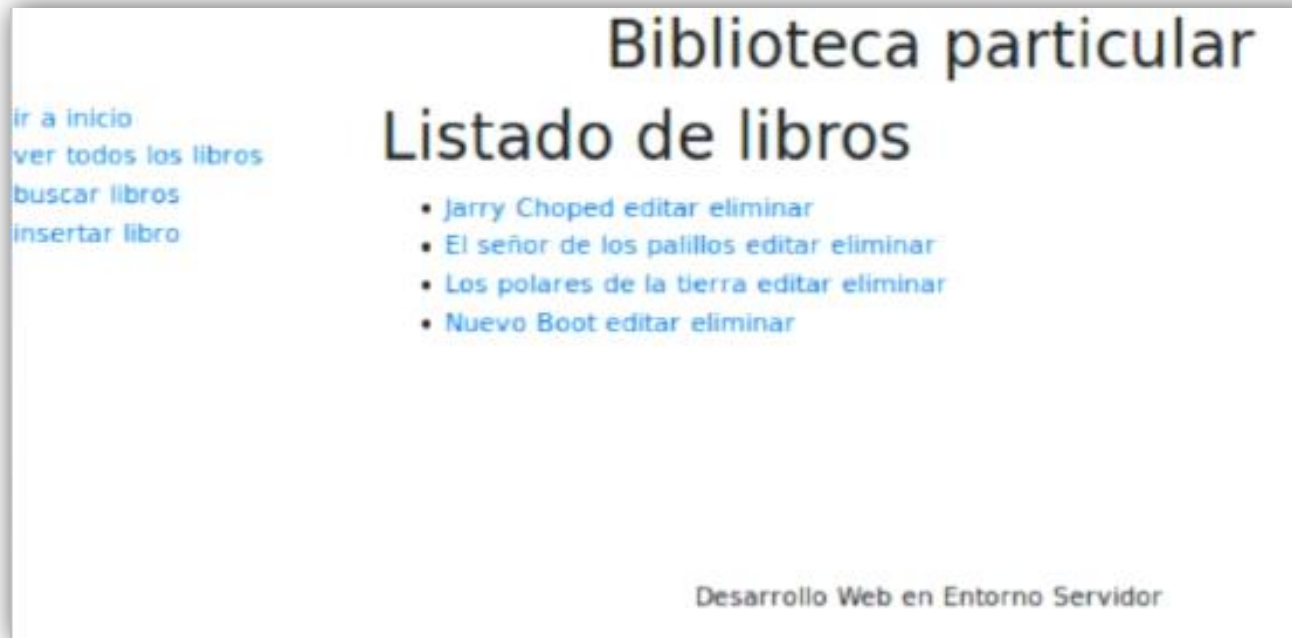
- El señor de los anillos [editar](#)  
[eliminar](#)
- Los polares de la tierra [editar](#)  
[eliminar](#)
- Harry Potter [editar](#)  
[eliminar](#)

Desarrollo Web en Entorno Servidor

- Tendrás que ampliar el repositorio con un método llamado **buscarLibros(\$cadena)**. Consejo: No pongas los % en la sentencia DQL, sino en el `setParameter` posterior, como en la diapositiva 23 de la unidad anterior.



- Edita la plantilla **lista\_libros.html.twig** para que también ponga [\[editar\]](#) en cada libro.





- Crea un controlador **editar\_libro** vinculado a **/libro/editar/{isbn}** y a la vista **editar\_libro.html.twig**, que puede heredar de **nuevo\_libro.html.twig**.

Este controlador será como el de insertar pero, en este caso, no creará el formulario en base a un libro vacío, sino en base al libro cuyo **isbn** es el recibido en la ruta.

Biblioteca particular

### Nuevo libro

Ir a inicio  
Ver todos los libros  
Buscar libros  
Insertar libro

isbn: A001

Título: Jarry Choped

Autor: JK Bowling

Páginas: 100

Editorial: Editorial1

Enviar

Desarrollo Web en Entorno Servidor

Biblioteca particular

### Nuevo libro

Ir a inicio  
Ver todos los libros  
Buscar libros  
Insertar libro

isbn: A010

Título: Harry Potter

Autor: JK Rowling

Páginas: 345

Editorial: Editorial1

Enviar

Desarrollo Web en Entorno Servidor

Biblioteca particular

### Listado de libros

- El señor de los anillos editar eliminar
- Las polares de la tierra editar eliminar
- Nunya Boot editar eliminar
- Harry Potter editar eliminar

Desarrollo Web en Entorno Servidor

- Investiga en el siguiente enlace <https://diego.com.es/constraints-en-symfony> cómo añadir un criterio de validación para el número de páginas, que obligue a que sea al menos 100 e introduce su respectivo mensaje de error.



**Biblioteca particular**

### Nuevo libro

[Ir a inicio](#)  
[ver todos los libros](#)  
[buscar libros](#)  
[insertar libro](#)

Isbn  
A1111

Título  
Nuevo 1111

Autor  
Marina Fornés

Páginas  
**ERROR** El número mínimo de páginas es 100  
96

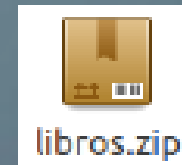
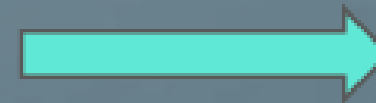
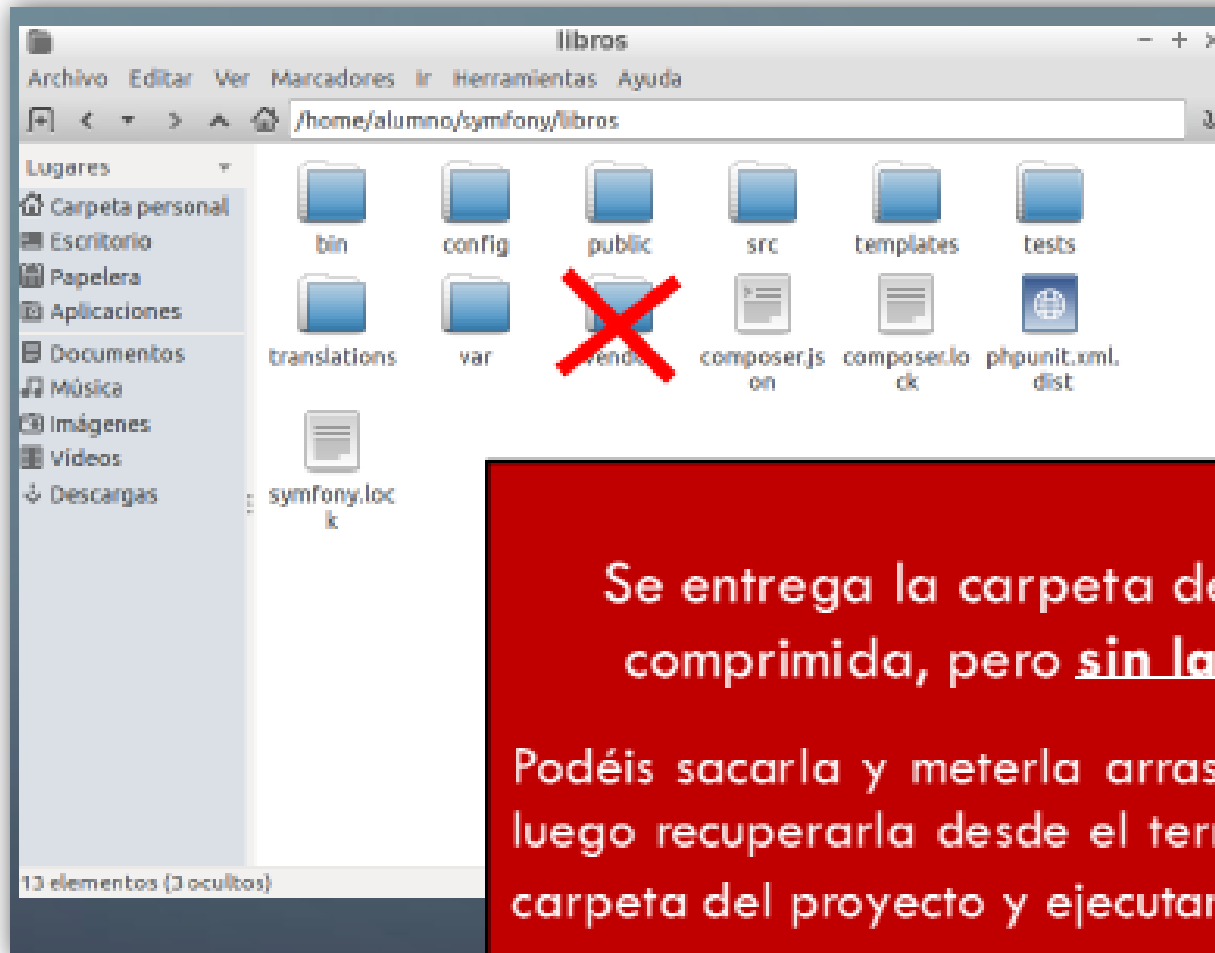
Editorial  
Editorial1

**Enviar**

Desarrollo Web en Entorno Servidor

- Si en el formulario de insertar libros pones un isbn existente debe dar error y mostrar su respectivo mensaje de error también.





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`