

Symfony

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

5. Generación de formularios y validación de datos

Ignacio Iborra Baeza
Julio Martínez Lucas

Índice de contenidos

Symfony.....	1
1. Generación de formularios.....	3
1.1. <i>Creación del formulario en el controlador.....</i>	<i>3</i>
1.1.1. Etiquetas personalizadas.....	4
1.1.2. Mejorando nuestro formulario.....	5
1.2. <i>Renderizando el formulario.....</i>	<i>5</i>
1.3. <i>Rellenando los campos del formulario.....</i>	<i>6</i>
2. Envío de formularios.....	7
2.1. <i>Modificación de datos.....</i>	<i>8</i>
3. Validación de formularios.....	10
4. Otras consideraciones finales.....	12
4.1. <i>Creación de clases para formularios.....</i>	<i>12</i>
4.2. <i>Añadiendo estilo a los formularios.....</i>	<i>13</i>
4.2.1. Añadir estilos para las validaciones.....	15
5. Ejercicios.....	16
5.1. <i>Ejercicio 1.....</i>	<i>16</i>
5.2. <i>Ejercicio 2 (opcional).....</i>	<i>17</i>
5.3. <i>Ejercicio 3.....</i>	<i>17</i>
5.4. <i>Ejercicio 4.....</i>	<i>17</i>

1. Generación de formularios

Hasta esta sesión hemos aprendido algunos conceptos útiles de Symfony y algunos de sus *bundles* más destacados, como por ejemplo la generación de vistas con el motor de plantillas *Twig*, el desarrollo de servicios, o la comunicación con la base de datos a través del ORM Doctrine. Hemos hecho algunos controladores de ejemplo para buscar datos, o para insertar. Pero, en este último caso, al no disponer aún de un mecanismo para que se envíen datos de inserción desde el cliente, hemos optado por ahora por insertar unos datos prefijados o *dummy data*, es decir, un contacto (o un libro) con unos datos ya predefinidos en el código.

En esta sesión veremos de qué forma se pueden definir formularios en Symfony asociados a una determinada entidad, para que lo que se envíe en el formulario se asocie a un objeto de dicha entidad, y para que podamos pre-cargar el formulario con los datos de una entidad ya existente, con el fin de poderlos modificar.

1.1. Creación del formulario en el controlador

Los formularios pueden crearse fácilmente desde cualquier controlador. Basta con que creamos u obtengamos el objeto asociado al formulario (por ejemplo, un contacto), y carguemos un formulario con él. En nuestra aplicación de contactos, vamos a crear un nuevo controlador que responda a la URI `/contacto/nuevo`, y que cree un contacto vacío y 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()
```

```

        ));
    }
    ...

```

Como podemos ver, a través del *form builder* de Symfony se crea el formulario. Después, añadimos 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 e-mail, y un botón de *submit* (*SubmitType*) para poder enviar el formulario. Podéis consultar [aquí](#) un listado más detallado de los tipos de campos que tenemos disponibles. Algunos 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 e-mail)
- *IntegerType* (cuadros de texto para números enteros)
- *NumberType* (cuadros de texto para números en general)
- *PasswordType* (cuadros enmascarados para passwords)
- *EntityType* (desplegables para elegir valores vinculados a otra entidad)
- *DateType* (para fechas)
- *CheckboxType* (para *checkboxes*)
- *RadioType* (para botones de radio o *radio buttons*)
- *HiddenType* (para controles ocultos)
- ... etc.

1.1.1. Etiquetas personalizadas

Como podemos ver para el caso del botón de *submit*, podemos especificar un tercer parámetro en el método *add* que es un array de propiedades del control en cuestión. Una de ellas es la propiedad *label*, que nos permite especificar qué texto tendrá asociado el control. Por defecto, se asocia el nombre del atributo correspondiente en la entidad, pero podemos cambiarlo por un texto personalizado. Para el e-mail, por ejemplo, podríamos poner:

```
->add('email', EmailType::class, array('label' => 'Correo electrónico'))
```

1.1.2. Mejorando nuestro formulario

Aprovechando la variedad de tipos de campos que ofrece Symfony, vamos a mejorar un poco nuestro formulario:

- Por un lado, el campo *email* podemos indicar que sea de tipo *EmailType*
- Por otro lado, para la provincia que añadimos en la sesión de Doctrine, podemos emplear un *EntityType* que tome sus datos de la entidad *Provincia*.

Con esto, el formulario quedaría así:

```
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
...
public function nuevo()
{
    ...
    $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();
    ...
}
```

1.2. Renderizando el formulario

El código del controlador anterior termina renderizando una vista llamada *nuevo.html.twig* que, por ahora, no existe. En esta vista deberemos renderizar 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 en negrita bajo el encabezado *h1* son las responsables de la renderización del formulario propiamente dicha, a partir del parámetro *formulario* que le pasamos desde el controlador.

Si ahora accedemos a *symfony.contactos/contacto/nuevo* podremos ver el formulario:



Nuevo contacto

Nombre

Telefono

Email

Provincia

1.3. Rellenando los campos del formulario

Volvamos a nuestro controlador *nuevo*. Si construimos un objeto relleno con 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)  
    ...  
}
```



Nuevo contacto

Nombre

Telefono

Email

Provincia

2. Envío de formularios

Hablemos ahora sobre cómo enviar el formulario. Por defecto, si no se indica nada, el formulario se envía por POST a la misma URI que lo genera (en nuestro caso, a `/contacto/nuevo`). De hecho, si intentamos enviar el formulario en este momento, con los datos que sean, se volverá a cargar la vista del formulario, pero no habremos insertado nada.

Para gestionar el envío de estos datos, hay que hacer algunas modificaciones sobre nuestro controlador:

- 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 en caso de error, o en caso de que no se haya enviado (por ejemplo, cuando cargamos el formulario para rellenarlo).

Uniando estas premisas, nuestro controlador quedaría así:

```
/**
 * @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()
    ));
}

```

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.

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

2.1. Modificación de datos

Lo que hemos hecho en el ejemplo anterior es una inserción de un nuevo contacto, pero... ¿cómo sería hacer una modificación de contacto existente?. El funcionamiento sería muy similar, pero con un pequeño cambio: la ruta del controlador recibirá como parámetro el código del contacto a modificar, y a partir de ahí, buscaríamos el contacto y lo cargaríamos en el formulario, incluyendo su *id* en un campo oculto. De esta forma, al hacer *persist* se modificaría el contacto existente.

Podemos probarlo con este controlador:

```

/**
 * @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);
    ...
}

```

Vemos que el código es muy similar al de la inserción, salvo por el parámetro *codigo* que recibe el controlador con el código del contacto a editar, la búsqueda inicial de dicho

contacto en la base de datos, y el campo oculto *id* en el formulario. Recuerda añadir el correspondiente *use* al principio para que reconozca la clase *HiddenType*.

```
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
```

Para que el código anterior funcione, debemos añadir a nuestra entidad *Contacto* un *setter* para poder asignarle un *id*, ya 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;
    }
    ...
}
```

Ahora, si accedemos a *symfony.contactos/contacto/editar/1*, por ejemplo (suponiendo que tengamos un contacto con *id = 1* en la base de datos), se cargará el formulario con sus datos, y al enviarlo, se modificarán los campos que hayamos cambiado, y se cargará la página de inicio.

NOTA: procura que el contacto a editar tenga un *id* de provincia válido, ya que cuando añadimos el campo *provincia* indicamos que no podía tener nulos, y si había datos insertados con anterioridad, se les habrá asignado un *id* de provincia 0, que no es válido.

En este punto, puedes realizar el [Ejercicio 2](#) del final de la sesión, que es de carácter opcional.

3. Validación de formularios

Ahora que ya sabemos crear, enviar y gestionar formularios, veamos un último paso, que sería 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 lo tanto, la validación la obtendremos añadiendo una serie de restricciones o comprobaciones a estas clases. Por ejemplo, para indicar que el nombre, teléfono y e-mail del contacto no pueden estar vacíos, añadimos estas anotaciones en los atributos de la clase *Contacto*:

```
...
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;

    ...
}
```

Estas aserciones repercuten directamente sobre el código HTML del formulario, donde se añadirá el atributo *required* para que se validen los datos en el cliente.

En el caso del e-mail, además, podemos especificar que queremos que sea un e-mail válido, lo que se consigue con esta otra anotación:

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Email()
```

```

* @Assert\NotBlank()
* @Assert\Email()
*/
private $email;

```

Estas funciones de validación admiten una serie de parámetros útiles. Uno de los más útiles es *message*, que se emplea para determinar el mensaje de error que mostrar al usuario en caso de que el dato no sea válido. Por ejemplo, para el e-mail, podemos especificar este mensaje de error:

```

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

```

Y se disparará cuando no escribamos un e-mail válido e intentemos enviar el formulario:

NOTA: 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*.

Podéis encontrar más información sobre qué aserciones se pueden usar para validar datos en la [documentación oficial de Symfony](#).

En este punto, intenta realizar el [Ejercicio 3](#) de los propuestos al final de la sesión.

4. Otras consideraciones finales

Para finalizar esta sesión, veamos algunas cuestiones que hemos dejado en el tintero y no dejan de ser igualmente importantes.

4.1. Creación de clases para formularios

Hasta ahora, hemos definido los formularios directamente en los controladores afectados. Así, en el caso de la aplicación de contactos, hemos definido un formulario para la ruta */contacto/nuevo* y otro (el mismo, prácticamente) 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. Como siempre, podemos 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. En nuestro caso, crearemos una clase llamada *ContactoType*, que heredará de una clase base genérica de Symfony llamada *AbstractType*. Dentro, definimos el método *buildForm* que se encargará de crear el formulario, como hacíamos antes en el método *nuevo* o en *editar*:

```
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'));
    }
}

```

Ahora, sólo nos queda reemplazar el código de crear el formulario en *nuevo* o *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())
        ...
}

```

4.2. Añadiendo estilo a los formularios

Los formularios que hemos generado en esta sesión son muy funcionales, pero poco vistosos, ya que carecen de estilos CSS propios. Si quisiéramos añadir CSS a estos formularios, tenemos varias opciones.

Una opción rudimentaria consiste en añadir clases (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. Al final tendremos algo como esto (se omiten las URLs de CSS y Javascript porque son algo largas):

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

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']
```

Con estos dos cambios, la apariencia de nuestro formulario de contactos queda así:



The image shows a web form titled "Nuevo contacto" in a blue header. Below the title are four input fields: "Nombre", "Telefono", "Correo electrónico", and "Provincia". The "Provincia" field is a dropdown menu with "Alicante" selected. At the bottom of the form is a grey button labeled "Enviar".

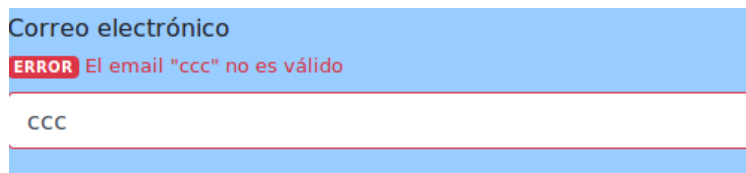
Se tienen otras alternativas, como por ejemplo no indicar esta configuración general en *config/packages/twig.yaml* e indicar formulario por formulario el tema que va a tener:

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

Existen también otros temas disponibles que utilizar. Podéis consultar más información [aquí](#).

4.2.1. Añadir estilos 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:

A screenshot of a web form field. The field is labeled "Correo electrónico" in a light blue header. Below the header, there is a red error message: "ERROR El email 'ccc' no es válido". The input field itself contains the text "ccc" and has a red border. The entire form is set against a light blue background.

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 un atributo *novalidate* en el formulario al renderizarlo:

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

En este punto, puedes realizar el [Ejercicio 4](#) de los propuestos al final de la sesión.

5. Ejercicios

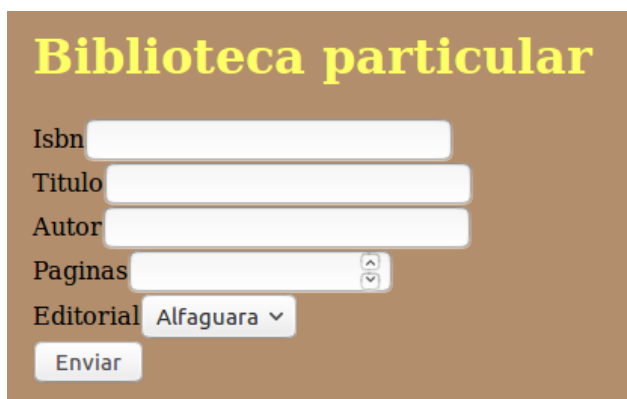
5.1. Ejercicio 1

Vamos a definir un formulario en nuestra aplicación de libros para insertar nuevos libros. Sigue estos pasos:

1. Crea un nuevo controlador en *LibroController* que se llame *nuevo* y responda a la ruta */libro/nuevo*. Deberá crear un libro vacío, y un formulario 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 la editorial del libro (de tipo *EntityType*, asociado a la entidad *Editorial* que creamos en la sesión anterior).

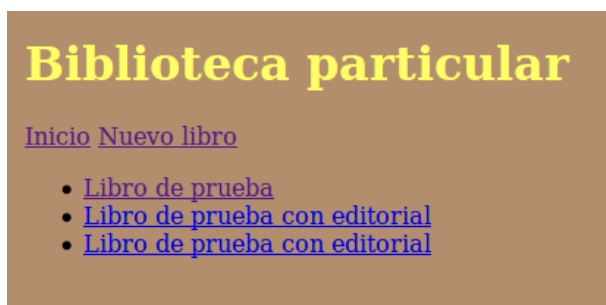
Tras comprobar si se ha enviado el formulario, recogerá los datos del envío, los asignará a un objeto *libro* y hará la pertinente inserción, redirigiendo a la página de inicio tras ello. Si no se ha enviado el formulario, o no es válido, se cargará la vista *nuevo.html.twig*

2. Crea la plantilla *nuevo.html.twig* de forma similar a la que has hecho para el ejemplo de contactos, mostrando el formulario en la zona apropiada:



The screenshot shows a web form titled "Biblioteca particular" in yellow text on a brown background. The form contains the following fields: "Isbn" (text input), "Titulo" (text input), "Autor" (text input), "Paginas" (number input with up/down arrows), and "Editorial" (dropdown menu with "Alfaguara" selected). At the bottom is an "Enviar" button.

3. Puedes aprovechar también para retocar la plantilla base (*base.html.twig*) y hacer un menú general para todas las páginas, con dos enlaces: uno para ir al inicio, y otro para insertar un nuevo libro:

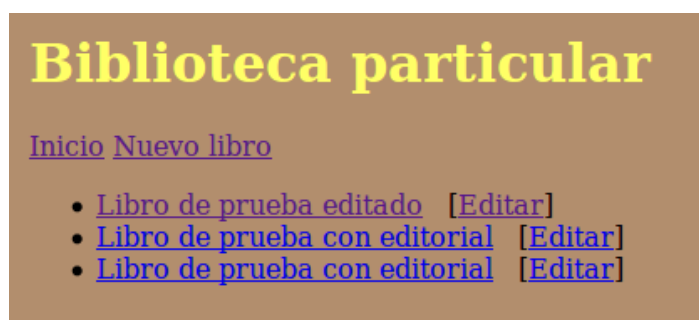


Prueba a realizar la inserción de algún libro para comprobar el funcionamiento del controlador.

5.2. Ejercicio 2 (opcional)

Partiendo del formulario anterior, vamos a realizar ahora la edición de libros existentes. Para ello, sigue estos pasos:

1. Crea un controlador que responda a la URI `/libro/editar/{isbn}`, que recibirá como parámetro el ISBN del libro a editar. Al igual que hemos hecho para el ejemplo de contactos, el controlador creará el formulario con los datos precargados del libro en cuestión, recogerá el envío del formulario, lo validará y realizará la modificación pertinente. Si todo es correcto, enviará a la página de inicio, y si no (o si no se ha enviado ningún formulario, renderizará el formulario `nuevo.html.twig` del ejercicio anterior.
2. Modifica la vista `inicio.html.twig` para que, junto a cada libro, se muestre un enlace para poderlo editar. Este enlace enviará al controlador anterior.



5.3. Ejercicio 3

Vamos a validar la clase *Libro* de nuestra aplicación de libros. Los criterios de validación serán los siguientes:

- El ISBN, título, autor y número de páginas no pueden estar vacíos
- El número de páginas, además, debe ser un dato numérico entero. Esto queda validado en el cliente, al haber utilizado un campo de tipo *IntegerType*, pero averigua cuál sería la anotación de validación correspondiente para el atributo *paginas* de la clase *Libro*.
 - Como parte opcional, averigua cómo hacer para asegurarnos de que sea mayor o igual que 100.

5.4. Ejercicio 4

Crea una clase `src/Form/LibroType` en el proyecto de libros, donde definir el formulario para inserción y edición de libros, de forma similar a como hemos hecho el de contactos. Sustituye las líneas de creación del formulario en los controladores *nuevo* y *editar* por el uso de esta nueva clase.

Opcionalmente, puedes también añadir el estilo de Bootstrap 4 al formulario de libros.