Capítulo 9
Formularios

1. Un componente MVC

Los formularios son elementos esenciales de los sitios web: constituyen la manera principal que utilizan los usuarios para interactuar con la aplicación.

Se muestran en las páginas (capa Vista) y, una vez enviados, normalmente se utilizan para modificar datos (capa Modelo), todo ello orquestado por el controlador.

Encontramos, pues, a los protagonistas de nuestro famoso patrón de diseño MVC (consulte Arquitectura del framework - El modelo de diseño MVC).

Esta particularidad hace que el componente «Form» de Symfony no sea el más difícil de aprender, pero sí uno de los más completos. Además, para entender mejor este capítulo, es necesario dominar el Contralor y Twig.

1.1 El modelo

En su uso más común, los formularios permiten interactuar con la capa Modelo (aunque el componente puede trabajar con tablas). Por lo tanto, los «objetivos» de los formularios son objetos.

Imagine una clase que representa a un cliente:

```
<?php
namespace App\Model;

class Cliente
{
   private $nombre;
   private $fechaDeNacimiento;

   public function setNombre($nombre)
   {
       $this->nombre = $nombre;

       return $this;
   }

   public function getNombre()
   {
       return $this->nombre;
   }

   public function setFechaDeNacimiento($fechaDeNacimiento)
   {
       $this->fechaDeNacimiento = $fechaDeNacimiento;

       return $this;
   }

   public function getFechaDeNacimiento()
   {
       return $this->fechaDeNacimiento;
   }
}
```

Es una clase clásica de PHP, pero podría muy bien corresponder a una entidad Doctrine si alguna vez se definiera su mapeo.

Uno de los objetivos del componente «Form» será crear o modificar instancias de esta clase, basadas en datos transmitidos por el usuario, a través de un formulario.

1.2 El controlador

El controlador, como punto central, es responsable de:

– crear o recuperar el objeto de la capa Modelo,
– crear el formulario y hacerle consciente de este objeto de la capa Modelo, que permitirá la modificación,
– pasar el formulario a la vista para que se encargue de la visualización de los diferentes campos,
– una vez que formulario ha sido enviado, validar los datos de modo que se puedan mostrar mensajes de error si es necesario (por ejemplo, «Este campo es obligatorio», «Esta no es una dirección de correo electrónico válida», etc.).

A continuación, se muestra el código dentro de una acción (no es necesario que lo entienda en detalle por el momento; por supuesto, volveremos sobre este tema más adelante en este capítulo):

```
...
/**
 * @Route("/")
 */
public function index(Request $request)
{
   $cliente = new Cliente();

   $form = $this->createFormBuilder($cliente)
       ->add('nombre', TextType::class)
       ->add('fecha_de_nacimiento', BirthdayType::class)
       ->add('validar', SubmitType::class)
       ->getForm()
   ;

   $form->handleRequest($request);

   if ($form->isSubmitted() && $form->isValid()) {
       return new Response('El formulario es válido.');
   }

   return $this->render('default/index.html.twig',
       [ 'form' => $form->createView() ]
   );
}
```

1.3 La vista

Al final de nuestra acción, enviamos nuestro formulario a la vista dentro de una variable **form**. El propósito de la template es dar formato a este formulario, mostrando sus diferentes campos.

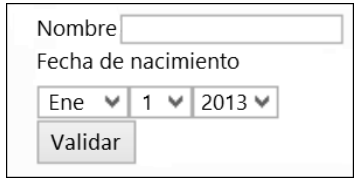
Observación

*Tenga en cuenta que el objeto formulario no se envía tal cual, sino que se debe invocar el método **$form->createView()**, que transforma el formulario en un objeto especialmente adaptado para las templates.*

A nivel de plantilla, la visualización del formulario es desconcertantemente simple:

```
<html>
<body>
    {{ form(form) }}
</body>
</html>
```

Es suficiente con invocar la función Twig **form()**, pasando como argumento el objeto que representa al formulario en las vistas.



De este modo, se genera el formulario completo. Obviamente, la representación será personalizable; nos detendremos en ella más adelante en este capítulo.

2. Funcionamiento del componente

2.1 El objeto «Form»

El objeto **Form** es el elemento principal utilizado a nivel de controlador. En el ejemplo anterior, está contenido en la variable **$form**. Representa el conjunto de campos del formulario en forma jerárquica.

Este es el punto central alrededor del que gira todo. El objeto que representa la consulta HTTP o el objeto de la capa Modelo, se «inyecta» en él. Las tareas de procesamiento típicas (como administrar el envío, validar o crear la vista de formulario) también se realizan a través de este objeto **Form**.

2.1.1 Envío (submit)

El objeto **Form** tiene un método llamado **handleRequest()**, que recibe como parámetro el objeto **Request** (la consulta HTTP actual). Este método es capaz de realizar introspección.

De esta introspección surgirá una conclusión, a saber, una respuesta a la siguiente pregunta: durante la consulta HTTP actual, ¿el usuario envía el formulario o solo lo muestra?

Si se está enviando (normalmente esta acción se reconoce porque el método de consulta HTTP actual es de tipo POST), los datos publicados se adjuntan al formulario y el objeto de la capa Modelo también se actualiza (siempre que los datos introducidos por el usuario sean válidos; veremos esto con más detalle más adelante).

De forma predeterminada, invocar el método **handleRequest()** equivaldría a:

```
if ($request->isMethod('POST')) {
    $form->submit($request);
}
```

Observación

*El método **submit()** permite adjuntar manualmente datos a un formulario.*

2.1.2 Validación

El método **isValid()** permite saber si los valores enviados por el usuario son correctos, basándose en un conjunto de reglas (denominadas «restricciones») definidas por el desarrollador.

Estas reglas se pueden definir sobre la marcha, es decir, durante la creación del formulario o directamente en el objeto de la capa Modelo.

Observación

La validación se aborda en detalle más adelante en este capítulo.

2.1.3 Vista

Para terminar, un objeto especial, una «vista» del formulario, se puede recuperar del objeto **Form** utilizando su método **createView()**. Aquí no sería interesante entrar en detalles; solo es necesario saber que las templates necesitan este objeto especial, y no el objeto principal **Form**, para funcionar.

2.2 Tipos

Las clases como **TextType** o **SubmitType** que usamos anteriormente se corresponden con los **tipos** de campo de nuestro formulario. Los referenciamos con su FQCN usando la palabra clave **::class** (disponible desde PHP 5.5).

2.3 Opciones

Al configurar los diferentes campos del formulario, se puede usar un tercer argumento del método **add()** para pasar opciones, a través de una tabla.

Estas opciones se utilizan para personalizar el campo: cambiar su etiqueta, su valor predeterminado o renderizado son algunas de las posibilidades que ofrecen las opciones.

Vamos a ilustrar esto con un ejemplo concreto. Imaginemos que el campo **fecha_de_nacimiento** está restringido a jóvenes de 7 a 77 años:

```
$form = $this->createFormBuilder($cliente)
    ->add('nombre', TextType::class)
    ->add('fecha_de_nacimiento', BirthdayType::class, array(
        'years' => range(date('Y') - 77, date('Y') – 7)
    ))
    ->add('validar', SubmitType::class)
    ->getForm()
;
```

Para esto utilizamos la opción **years**, que acepta una tabla de valores correspondientes a los años disponibles para el campo.

Observación

*Hay opciones comunes a todos los tipos, mientras que otras son específicas para un tipo de campo determinado. Por ejemplo, aquí la opción **years** es específica de los campos relacionados con la gestión de las fechas (no tendría sentido para un campo de tipo **textarea**). La lista completa de las opciones disponibles para cada tipo se detalla a continuación.*

2.4 Los objetos «Form» y «FormBuilder»

2.4.1 El FormBuilder

El «FormBuilder» es a «Form» lo que, en Doctrine, el «QueryBuilder» es a «Query». Su función es configurar y crear objetos **Form**.

Anteriormente, lo hemos utilizado de la siguiente manera:

```
$form = $this->createFormBuilder($cliente)
    ->add('nombre', TextType::class)
    ->add('fecha_de_nacimiento', BirthdayType::class)
    ->add('validar', SubmitType::class)
    ->getForm()
;
```

Aquí, usamos el método heredado **createFormBuilder()**, lo que permite, como su nombre indica, crear una instancia de **FormBuilder**.