Capítulo 4
261

1. Funciones

1.1 Introducción

Al igual que en otros lenguajes de programación, PHP ofrece la posibilidad de definir sus propias funciones (llamadas funciones del "usuario") con todas las ventajas asociadas (modularidad, uso de mayúsculas...). Una función es un conjunto de instrucciones identificadas por un nombre, cuya ejecución devuelve un valor y cuya llamada se puede utilizar como operando en una expresión. Un procedimiento es un conjunto de instrucciones identificadas por un nombre que puede ser llamado como una instrucción.

1.2 Declaración y llamada

La palabra clave `función` permite introducir la definición de una función.

Sintaxis

```
función nombre_función([parámetro]) [: tipo]{
  instrucciones;
}
```

`nombre_función`	Nombre de la función (debe respetar las reglas de denominación presentes en el capítulo Introducción a PHP - Estructura básica de una página PHP). En este nombre no se diferencian mayúsculas y minúsculas (para PHP, las funciones `unafunción` y `UnaFunción` son las mismas).
`parámetro`	Parámetros posibles de la función expresados como una lista de variables (véase la sección Parámetros): `$parámetro1, $parámetro2, ...`
`tipo`	Declaración del tipo de datos devuelto por la función. Valores posibles: `int`, `float`, `string`, `bool`, `array`, `callable`, `iterable`, `object`, `mixed`, `void`, un nombre de clase o de interfaz (véase en este capítulo la sección Clases) o una unión de tipos. El nombre del tipo se puede preceder por un punto de interrogación (`?`) que indica que la función puede devolver un valor `NULL`. En el capítulo Introducción a PHP puede encontrar la definición de los tipos de datos (apartado Las bases del lenguaje PHP - Tipos de datos).
`instrucciones`	Conjunto de instrucciones que componen la función.

El nombre de la función no debe ser una palabra reservada de PHP (nombre de función nativa, de instrucción) ni ser igual al nombre de otra función definida de antemano.

Una función de usuario se puede llamar como una función nativa de PHP: en una asignación, en una comparación, etc.

Si la función devuelve un valor, es posible utilizar la instrucción `return` para definir el valor de retorno de la función.

Sintaxis

```
return expresión;
```

`expresión`	Expresión cuyo resultado constituye el valor de retorno de la función (`NULL` por defecto).

El resultado de una función puede ser de cualquier tipo (cadena, número, matriz, etc.).

La instrucción `return` detiene la ejecución de la función y devuelve el resultado de `expresión` a quien realiza la llamada. Si hay varias instrucciones `return` en la función, la primera que se encuentre en el desarrollo de las instrucciones es la que define el valor de retorno y provoca la interrupción de la función. Si la función no incluye ninguna instrucción `return` (o si no se ha ejecutado ninguna instrucción `return`), el valor de retorno de la función es `NULL`.

© Editions ENI - All rights reserved

Ejemplo

```
<?php
// Función sin parámetro que muestra "¡Hola!"
// Sin valor de retorno.
function mostrar_hola() {
  echo '¡Hola!<br />';
}
// Función con dos parámetros que devuelve el producto
// de los dos parámetros.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
// Llamada de la función mostrar_hola.
mostrar_hola();
// Usos de la función producto:
// - en una asignación
$resultado = producto(2,4);
echo "2 x 4 = $resultado<br />";
// - en una comparación
if (producto(10,12) > 100) {
  echo '10 x 12 es superior a 100.<br />';
}
?>
```

Resultado

```
¡Hola!
2 x 4 = 8
10 x 12 es superior a 100.
```

Observación

En el lenguaje PHP, no existe un procedimiento real. Para definir algo equivalente a un procedimiento, basta con definir una función que no devuelva ningún valor y llamar a la función como si se tratara de una instrucción (como la función `mostrar_hola`*, por ejemplo). Una función que no devuelve nada, se puede declarar de manera explícita con el tipo de retorno* `void`*.*

Como ya mencionamos anteriormente, el contenido de una matriz se puede transformar en una lista de parámetros dentro de una llamada de función gracias al operador `...` (tres puntos suspensivos).

Ejemplo

```
<?php
// Función con tres parámetros que devuelve la suma
// de los tres parámetros.
function suma($valor1,$valor2,$valor3) {
  return $valor1 + $valor2 + $valor3;
}
// Transformación del contenido de una matriz en
// lista de parámetros.
$valores = [1,2,3];
echo '1 + 2 + 3 = ',suma(...$valores),'<br />';
// Lo mismo para una parte solamente de los parámetros
// con una matriz definida directamente en la llamada.
echo '1 + 2 + 4 = ',suma(1,...[2,4]),'<br />';
?>
```

Resultado

```
1 + 2 + 3 = 6
1 + 2 + 4 = 7
```

Cuando una función devuelve una matriz, es posible acceder directamente a un elemento de la matriz llamando a la función con una sintaxis de tipo `función(...)[clave]`.

Ejemplo

```
<?php
// Definición de una función que devuelve una matriz.
function quien() {
  return ['Olivier','Heurtel'];
}
// Llamada a la función y recuperación directa del nombre almacenado
// en el índice 0 de la matriz devuelta.
$nombre = quien()[0];
echo "quien()[0] = $nombre<br />";
?>
```

Resultado

```
quien()[0] = Olivier
```

Esta técnica funciona también cuando la función devuelve una matriz multidimensional con una sintaxis de tipo `función(...)[clave1][clave2]`.

Es posible utilizar una función antes de definirla.

© Editions ENI - All rights reserved

Ejemplo

```
<?php
// Utilización de la función producto.
echo producto(5,5);
// Definición de la función producto.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
?>
```

Resultado

```
25
```

Por lo tanto, no hay ningún problema para definir funciones que se llamen entre ellas.

Observación

Una función se puede utilizar solo en el script donde se define. Para utilizarla en varios scripts, es necesario o bien copiar su definición en los diferentes scripts (se pierde el interés por definir una función) o bien definirla en un archivo incluido donde la función sea necesaria.

Ejemplo

– Archivo `funciones.inc` que contiene la definición de funciones:

```
<?php
// Definición de la función producto.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
?>
```

– Script que utiliza las funciones definidas en `funciones.inc`:

```
<?php
// Inclusión del archivo contenedor de la definición de las funciones.
include('funciones.inc');
// Utilización de la función producto.
echo producto(5,5);
?>
```

Declaración del tipo de retorno

Es posible definir el tipo de datos devuelto por una función.

Cuando es el caso, en el modo de funcionamiento por defecto (a diferencia de modo estricto que se presenta a continuación), PHP realiza si es necesario una conversión automática del valor devuelto al tipo de datos declarado.

Ejemplo

```
<?php
// Declaración de dos funciones que devuelven el producto
// de dos parámetros, el segundo especifica un tipo
// de datos "entero" para el valor de retorno.
function producto1($valor1,$valor2) {
  return $valor1 * $valor2;
}
function producto2($valor1,$valor2) : int {
  return $valor1 * $valor2;
}
// Llamada de dos funciones con los mismos parámetros
echo 'producto1(20,1/7) => ',var_dump(producto1(20,1/7)),'<br />';
echo 'producto2(20,1/7) => <b>',var_dump(producto2(20,1/7)),'</b><br />';
?>
```

Resultado

```
producto1(20,1/7) => float(2.8571428571428568)
producto2(20,1/7) => int(2)
```

En este ejemplo, podemos ver claramente que PHP ha convertido el valor devuelto por la segunda función a un valor entero (con las reglas de conversión mencionadas en el capítulo Introducción a PHP - Las bases del lenguaje PHP - Tipos de datos).

Si PHP no es capaz de realizar la conversión (tipos de datos no convertibles entre ellos), se produce una excepción `TypeError`. Esta excepción detiene el script si no se maneja (véase en este capítulo la sección Clases - Excepciones).

Ejemplo

```
<?php
// Declaración y llamada de una función que debe devolver una
// matriz pero que devuelve una cadena de caracteres.
function quien() : array {
  return 'Olivier Heurtel';
}
echo 'quien()[0] = ',quien()[0];
?>
```

Resultado

```
quien()[0] =
Fatal error: Uncaught TypeError: Return value of quien() must be of the
type array, string returned in /app/scripts/index.php:5 Stack trace: #0
/ app/scripts/index.php(7): quien() #1 {main} thrown in/app/scripts/
index.php on
line 5
```

© Editions ENI - All rights reserved

Una función declarada con un tipo de retorno diferente de `void`, debe devolver un valor no `NULL`. Si este no es el caso, se devuelve un error, diferente según el caso:

Ausencia de instrucción `return`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
the type int, none returned in ...
```

Instrucción `return vacía`

```
Fatal error: A function with return type must return a value in ...
```

Instrucción `return NULL`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
type int, null returned in ...
```

Para autorizar una función para que devuelva un valor `NULL`, hay que utilizar un punto de interrogación (`?`), delante del nombre del tipo (diferente de `void`).

```
<?php
// Declaración y llamada de una función que especifica un
// tipo de datos de retorno que puede ser NULL
function cubo($valor) : ?int {
  if (is_null($valor)) {
    return NULL;
  } else {
    return $valor ** 3 ;
  }
}
echo 'cubo(2) => <b>',var_dump(cubo(2)),'</b><br />';
echo 'cubo(NULL) => <b>',var_dump(cubo(NULL)),'</b><br />';
?>
```

Resultado

```
cubo(2) => int(8)
cubo(NULL) => NULL
```

Incluso con esta opción, la función debe tener una instrucción `return` no vacía. Si no es el caso, se devuelve un error, diferente según el caso:

Ausencia de instrucción `return`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
type int, none returned in ...
```

Instrucción `return` vacía

```
Fatal error: A function with return type must return a value (did you
mean "return null;" instead of "return;"?) in ...
```

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.

© Editions ENI - All rights reserved

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.

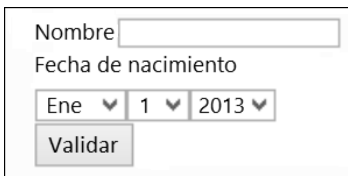
Observación

*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.

© Editions ENI - All rights reserved

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.

© Editions ENI - All rights reserved

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**.