DESARROLLO WEB ÁGIL CON **SYMFONY2**

Javier Eguiluz

Esta página se ha dejado vacía a propósito

Desarrollo web ágil con Symfony2

Javier Eguiluz

Esta página se ha dejado vacía a propósito

Sobre esta edición

Desarrollo web ágil con Symfony2

Esta obra se publicó el 09-09-2016 haciendo uso del gestor de publicaciones easybook versión 4.9.0, una herramienta para publicar libros que ha sido desarrollada con varios componentes de Symfony (http://symfony.com/components) .

Symfony es una marca registrada por Fabien Potencier. Este libro hace uso de la marca gracias al consentimiento expreso otorgado por su autor y bajo las condiciones establecidas en http://symfony.com/trademark

Otras marcas comerciales: el resto de marcas, nombres, imágenes y logotipos citados o incluidos en esta obra son propiedad de sus respectivos dueños.

Límite de responsabilidad: el autor no ofrece garantías sobre la exactitud o integridad del contenido de esta obra, por lo que no se hace responsable de los daños y/o perjuicios que pudieran producirse por el uso y aplicación de los contenidos. Asimismo, tampoco se hace responsable de los cambios realizados por los sitios y aplicaciones web mencionadas desde la publicación de la obra.

Esta página se ha dejado vacía a propósito

Licencia

© Copyright 2016 Javier Eguiluz

Derechos de uso: todos los derechos reservados. No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito del titular del Copyright.

El autor prohíbe expresamente la publicación o compartición de esta obra en cualquier sitio web o aplicación informática que permita el libre acceso, lectura o descarga de la obra por parte de otras personas, robots o máquinas. Esta prohibición se extiende incluso a aquellos casos en los que no exista ánimo de lucro.

Si eres formador, puedes usar esta obra para impartir cursos, talleres, jornadas o cualquier otra actividad formativa relacionada directa o indirectamente con el objeto principal de la obra. Este permiso obliga al reconocimiento explícito de la autoría de la obra y no exime del cumplimiento de todas las condiciones anteriores, por lo que no puedes distribuir libremente copias de la obra entre tus alumnos.

Esta página se ha dejado vacía a propósito

Dedicado a toda la comunidad Symfony,
especialmente a su creador, Fabien Potencier,
cuyo trabajo me inspira cada día.

Esta página se ha dejado vacía a propósito

Índice de contenidos

Sección 1 **Introducción** 17

Capítulo 1 **Lo que debes saber antes de comenzar** 19

1.1 Cómo leer este libro 19
1.2 Introducción a Symfony 19
1.3 Preparación del entorno de trabajo 20
1.4 Introducción a las nuevas funcionalidades de PHP 21
1.5 Introducción a YAML 25

Capítulo 2 **El proyecto** 29

2.1 Funcionamiento detallado de la aplicación 29
2.2 Wireframes 31
2.3 La base de datos 34
2.4 Aplicando la filosofía de Symfony 36
2.5 Entidades 37
2.6 Bundles 37
2.7 Enrutamiento 38

Capítulo 3 **Instalando y configurando Symfony** 41

3.1 El instalador de Symfony 41
3.2 Creando la aplicación Symfony 41
3.3 Actualizando la aplicación Symfony 42
3.4 Instalado una aplicación Symfony existente 43
3.5 Comprobando la instalación de Symfony 44
3.6 Accediendo a la aplicación Symfony 44
3.7 Estructura de las aplicaciones Symfony 45

Sección 2 **Frontend** 47

Capítulo 4 **Creando las primeras páginas** 49

4.1 La filosofía de Symfony 49
4.2 La primera página 50
4.3 Configurando los permisos 60
4.4 Configurando *la barra del final* en las URL 62

Capítulo 5 **La base de datos 63**
5.1 Entidades 63
5.2 Creando y configurando la base de datos 81
5.3 El *Entity Manager* 83
5.4 Archivos de datos o *fixtures* 88
5.5 Alternativas para generar el modelo 97
Capítulo 6 **Creando la portada 101**
6.1 Arquitectura MVC 101
6.2 El enrutamiento 102
6.3 El controlador 103
6.4 La plantilla 105
6.5 Entornos de ejecución 111
6.6 Depurando errores 115
6.7 Refactorizando el Controlador 120
6.8 Refactorizando el Modelo 125
6.9 Refactorizando la Vista 131
6.10 Funcionamiento interno de Symfony 136
6.11 El objeto `Request` 137
6.12 El objeto `Response` 141
Capítulo 7 **Completando el frontend 145**
7.1 Herencia de plantillas a tres niveles 145
7.2 Assets web (hojas de estilo y archivos JavaScript) 148
7.3 Mejorando la gestión de las imágenes 151
7.4 Seleccionando la ciudad activa 151
7.5 Creando la página de detalle de una oferta 158
7.6 Completando las plantillas con extensiones de Twig 164
7.7 Creando la página de ofertas recientes de una ciudad 169
7.8 Creando la portada de cada tienda 173
7.9 Refactorización final 177
Capítulo 8 **Registrando usuarios 183**
8.1 Creando la página de compras recientes 183
8.2 Restringiendo el acceso 188

8.3 Creando proveedores de usuarios 192
8.4 Añadiendo el formulario de *login* 194
8.5 Modificando las plantillas 205
8.6 Creando los archivos de datos de usuarios 210
8.7 Formulario de registro 212
8.8 Visualizando el perfil del usuario 238
Capítulo 9 **RSS y los formatos alternativos 253**
9.1 Formatos alternativos 253
9.2 Generando el RSS de las ofertas recientes de una ciudad 255
9.3 Generando el RSS de las ofertas recientes de una tienda 260
Capítulo 10 **Internacionalizando el sitio web 265**
10.1 Configuración inicial 265
10.2 Rutas internacionalizadas 266
10.3 Traduciendo contenidos estáticos 270
10.4 Traduciendo contenidos dinámicos 277
10.5 Traduciendo páginas estáticas 278
10.6 Traduciendo fechas 279
Capítulo 11 **Tests unitarios y funcionales 283**
11.1 Primeros pasos 283
11.2 Tests unitarios 284
11.3 Test funcionales 295
11.4 Configurando PHPUnit en Symfony 312
Sección 3 **Extranet 317**
Capítulo 12 **Planificación 319**
12.1 Bundles 319
12.2 Enrutamiento 320
12.3 Layout 321
Capítulo 13 **Seguridad 325**
13.1 Definiendo la nueva configuración de seguridad 325
13.2 Preparando el proveedor de usuarios de las tiendas 328
13.3 Creando el formulario de login 330
13.4 Creando un *security voter* propio 334

Capítulo 14 **Creando la parte de administración 341**
14.1 Creando la portada de la *extranet* 341
14.2 Mostrando las ventas de una oferta 345
14.3 Mostrando el perfil de la tienda 347
Capítulo 15 **Administrando las ofertas. 353**
15.1 Creando ofertas 353
15.2 Modificando las ofertas 365
Sección 4 **Backend 373**
Capítulo 16 **Planificación 375**
16.1 Bundles 375
16.2 Seguridad 376
Capítulo 17 **Admin generator 381**
17.1 SensioGeneratorBundle. 381
17.2 EasyAdminBundle 385
Capítulo 18 **Comandos de consola. 387**
18.1 Creando comandos de consola 387
18.2 Generando la *newsletter* de cada usuario 398
18.3 Enviando la newsletter 402
Capítulo 19 **Mejorando el rendimiento 409**
19.1 Mejorando el rendimiento de la parte del cliente 409
19.2 Mejorando el entorno de ejecución 409
19.3 Desactivando las funcionalidades que no utilizas 409
19.4 Mejorando la carga de las clases 410
19.5 Mejorando el rendimiento de Doctrine 411
19.6 Mejorando el rendimiento de la aplicación con cachés 421
Capítulo 20 **Caché 423**
20.1 La caché del estándar HTTP 423
20.2 Estrategias de caché. 425
20.3 Cacheando con *reverse proxies*. 436
20.4 ESI 442
Sección 5 **Apéndices 449**

Apéndice A **El motor de plantillas Twig 451**
A.1 Sintaxis básica 451
A.2 Twig para diseñadores 452
A.3 Twig para programadores 457
A.4 Extensiones 486
A.5 Usando Twig en Symfony 497
Apéndice B **Inyección de dependencias 505**
B.1 Entendiendo la inyección de dependencias. 505
B.2 La inyección de dependencias en Symfony 513

Sección 1

Introducción

Esta página se ha dejado vacía a propósito

CAPÍTULO 1

Lo que debes saber antes de comenzar

El libro que estás leyendo explica paso a paso cómo desarrollar una aplicación web completa utilizando el *framework* Symfony. Los contenidos del libro empiezan desde cero y por tanto, no es necesario que tengas conocimientos previos sobre cómo programar con Symfony.

En este primer capítulo se hace una introducción muy breve sobre Symfony y se explican brevemente algunas tecnologías relacionadas que debes conocer para poder programar con Symfony, como por ejemplo YAML y los *namespaces* de PHP.

1.1 Cómo leer este libro

Si estás empezando con Symfony, te recomiendo que leas el libro secuencialmente, desde el primer hasta el último capítulo. La primera vez que lo leas, es muy recomendable que tengas instalada la aplicación de prueba *Cupon* ((https://github.com/javiereguiluz/Cupon)) , para echar un vistazo a su código terminado y para probar la aplicación a medida que se desarrolla.

Cuando releas el libro por segunda vez, ya podrás desarrollar la aplicación a medida que leas cada capítulo. Además, podrás probar tus propias modificaciones en la aplicación y serás capaz de solucionar rápidamente cualquier error que se produzca.

Si eres un programador experto en Symfony, puedes leer el libro en cualquier orden, empezando por ejemplo por los capítulos que más te interesen (Caché (página 423), internacionalización (página 265), mejorando el rendimiento (página 409), etc.)

1.2 Introducción a Symfony

Symfony es un proyecto de software libre que publica pequeñas librerías PHP independientes llamadas *"componentes"*. Combinando varios de esos componentes, también ha publicado un *framework* para desarrollar aplicaciones web.

Debido al uso de los componentes, la arquitectura interna del *framework* está completamente desacoplada, lo que permite reemplazar o eliminar fácilmente aquellas partes que no encajan en tu proyecto.

Symfony también es el framework que más ideas incorpora del resto de frameworks, incluso de aquellos que no están programados con PHP. Si has utilizado alguna vez Ruby On Rails, Django o Spring encontrarás muchas similitudes en algunos de los componentes de Symfony.

Symfony 2.8 se publicó en noviembre de 2015, es la última versión de la rama 2.x y tiene soporte hasta noviembre de 2019. La versión 3.0, publicada al mismo tiempo, es exactamente igual que la 2.8, pero elimina todo el código que se había declarado obsoleto. Así que si aprendes a programar con Symfony 2.8, también sabrás programar para Symfony 3.0.

El sitio web oficial del proyecto es symfony.com (http://symfony.com) y las referencias imprescindibles para cualquier programador son:

- El libro oficial (http://librosweb.es/symfony_2_4/)
- Las *recetas* o artículos breves (http://symfony.com/doc/2.4/cookbook)
- La documentación de su API (http://api.symfony.com/2.4/index.html)

Para estar al día de las novedades de Symfony, puedes consultar el blog oficial (http://symfony.com/blog) y el sitio symfony.es (http://symfony.es) , que publica regularmente artículos de interés para la comunidad hispana del framework.

1.3 Preparación del entorno de trabajo

1.3.1 PHP

Symfony 2.8 es compatible con PHP 5.3.9 o superior mientras que Symfony 3 exige PHP 5.5.9 o superior. Asegúrate de disponer de una versión actualizada de PHP correctamente instalada en tu ordenador de desarrollo.

1.3.2 Composer

Composer (https://getcomposer.org/) es el gestor de paquetes utilizado por todas las aplicaciones PHP modernas, incluyendo Symfony. Composer es imprescindible para cualquier programador PHP, ya que permite instalar, actualizar y desinstalar librerías, paquetes y plugins, normalmente llamadas **dependencias**.

Si no lo has hecho ya, instala Composer en tu ordenador de desarrollo tal y como se explica en https://getcomposer.org/download/.

1.3.3 Entorno de desarrollo

Antes de empezar a trabajar con Symfony es muy recomendable instalar un buen entorno de desarrollo. Mi recomendación personal es que utilices PHPStorm (https://www.jetbrains.com/phpstorm/) porque es el mejor entorno de desarrollo para PHP. Además PHPStorm cuenta con un espectacular plugin para Symfony (https://plugins.jetbrains.com/plugin/7219) que multiplica tu productividad.

Cuando ya tengas experiencia con Symfony, puedes utilizar cualquier editor o entorno de desarrollo que te guste. Pero al empezar con Symfony, utilizar PHPStorm hará que cometas muchos menos errores y la curva de aprendizaje sea más suave.

1.4 Introducción a las nuevas funcionalidades de PHP

Symfony 2.8 utiliza todas las funcionalidades avanzadas de PHP introducidas desde la versión 5.3. De todas ellas, las más relevantes para los programadores Symfony son las funciones anónimas y los *namespaces*, que se explican en las siguientes secciones.

1.4.1 Funciones anónimas

Las funciones anónimas, también conocidas como *closures*, son funciones sin nombre que permiten crear *callbacks* de manera sencilla. El código fuente de Symfony hace un uso extensivo de estas funciones, como por ejemplo puedes ver en la clase `Symfony/Component/Console/Application.php`:

```
public function renderException($e, $output)
{
    $strlen = function ($string) {
        if (!function_exists('mb_strlen')) {
            return strlen($string);
        }

        if (false === $encoding = mb_detect_encoding($string)) {
            return strlen($string);
        }

        return mb_strlen($string, $encoding);
    };

    // ...

    $len = $strlen($title);
}
```

La variable `$strlen` almacena una función anónima que calcula la longitud de una cadena de texto. Esta función se adapta a las características del sistema en el que se ejecuta, utilizando la función `mb_strlen()` o `strlen()` para determinar la longitud de la cadena. Antes de PHP 5.3, el código anterior debía escribirse de la siguiente manera:

```
public function mi_strlen ($string)
{
    if (!function_exists('mb_strlen')) {
        return strlen($string);
    }

    if (false === $encoding = mb_detect_encoding($string)) {
        return strlen($string);
    }

    return mb_strlen($string, $encoding);
}

public function renderException($e, $output)
```

```
{
    // ...

    $len = mi_strlen($title);
}
```

El código interno de una función anónima no tiene acceso a ninguna variable externa. Todas las variables que necesite el código se deben pasar mediante la palabra reservada `use`:

```
public function renderException($e, $output)
{
    $strlen = function ($string) use ($output) {
        if (!function_exists('mb_strlen')) {
            $output->print(strlen($string));
        }

        if (false === $encoding = mb_detect_encoding($string)) {
            $output->print(strlen($string));
        }

        $output->print(mb_strlen($string, $encoding));
    };

    // ...
}
```

1.4.2 Namespaces

Según la Wikipedia, un *namespace* o espacio de nombres (https://es.wikipedia.org/wiki/Espacio_de_nombres) es *"un contenedor abstracto que agrupa de forma lógica varios símbolos e identificadores"*. En la práctica, los *namespaces* se utilizan para estructurar mejor el código de la aplicación. Todas las clases de Symfony utilizan los *namespaces* y por tanto, es imprescindible entenderlos bien antes de programar una aplicación Symfony.

Antes de que existieran los *namespaces*, las aplicaciones debían ser cuidadosas al elegir el nombre de sus clases, ya que dos o más clases diferentes no podían tener el mismo nombre. Si la aplicación contenía cientos de clases, como es habitual en los *frameworks*, el resultado eran clases con nombres larguísimos para evitar colisiones.

Gracias a los *namespaces* dos o más clases de una misma aplicación pueden tener el mismo nombre. El único requisito es que sus *namespaces* sean diferentes, de forma que la aplicación sepa en todo momento cuál se está utilizando.

Los siguientes ejemplos utilizan clases reales de la aplicación que se desarrolla en los próximos capítulos. Por el momento no trates de entender por qué las clases se llaman de esa manera y se encuentran en esos directorios.

Imagina que dispones de una clase PHP llamada `Oferta.php` que se encuentra en el directorio `src/AppBundle/Entity/` de tu proyecto. Para que Symfony pueda encontrar esa clase, es obligatorio que incluya el siguiente *namespace* al principio del todo:

```
<?php
namespace AppBundle\Entity;

// ...
```

En las aplicaciones Symfony, el *namespace* coincide con la ruta del archivo a partir del directorio `src/` del proyecto. El *namespace* utiliza la barra invertida (`\`) como separador, sin importar el sistema operativo que utilices. Además, el *namespace* nunca incluye el propio nombre de la clase.

Imagina ahora que dispones de otra clase llamada `DefaultController.php` que se encuentra en el directorio `src/AppBundle/Controller/` del proyecto. Para utilizar la anterior clase `Oferta.php` dentro de `DefaultController.php` debes escribir lo siguiente:

```
<?php
namespace AppBundle\Controller;

class DefaultController {
    $oferta = new \AppBundle\Entity\Oferta();
    // ...
}
```

Como las clases de PHP ya no tienen nombres únicos, debes escribir el *namespace* completo cuando quieres hacer referencia a una clase. Si utilizas muchas veces una clase esto es muy aburrido, así que puedes importarla una sola vez con la instrucción `use`:

```
<?php
namespace AppBundle\Controller;

// no añadas al sufijo '.php' al importar una clase con 'use'
use AppBundle\Entity\Oferta;

class DefaultController {
    $oferta = new Oferta();
    // ...
}
```

Las instrucciones `use` le dicen a PHP las clases que vas a usar en tu código para que así no tengas que importarlas cada vez. En el ejemplo anterior se importa una sola clase, pero si vas a utilizar muchas clases definidas bajo el mismo *namespace*, también puedes importar el *namespace* común a todas ellas. Por ejemplo, si dentro de `AppBundle\Entity` defines muchas clases, puedes hacer lo siguiente:

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

```
use AppBundle\Entity;

class DefaultController {
    $oferta = new \Entity\Oferta();
    // ...
}
```

En las aplicaciones Symfony lo habitual es **importar todas las clases individualmente**, incluso cuando se encuentran en el mismo directorio. Por eso en la mayoría de clases de Symfony verás al principio muchas instrucciones `use` para importar las clases utilizadas.

Una ventaja adicional de los *namespaces* es que se puede modificar el nombre de la clase al importarla. Para ello, indica el nuevo nombre con la palabra reservada `as` en la instrucción `use`:

```
<?php
namespace AppBundle\Controller;

use AppBundle\Entity\Oferta as Offer;

class DefaultController {
    $oferta = new Offer();
    // ...
}
```

Por último, tus clases también pueden tener el mismo nombre que las clases nativas de PHP. En estos casos, para referirse a la clase nativa de PHP, añade una barra `\` por delante:

```
<?php
use AppBundle\Utils\DateTime;
// ...

$fecha = new DateTime();  // Nuestra propia clase
$fecha = new \DateTime(); // La clase nativa de PHP
```

TRUCO Si utilizas PHPStorm, las instrucciones `use` se añaden automáticamente cuando utilizas una clase que no has importado previamente. Además de ahorrar mucho tiempo, esto hace que cometas menos errores al aprender Symfony.

1.4.3 Anotaciones

Las anotaciones son un mecanismo muy utilizado en lenguajes de programación como Java. Aunque PHP todavía no soporta anotaciones (al menos hasta la versión PHP 7.0), las aplicaciones Symfony pueden hacer uso de ellas gracias a una librería desarrollada por un proyecto externo llamado Doctrine.

Técnicamente las anotaciones no son más que comentarios incluidos en el propio código fuente de la aplicación. A diferencia de los comentarios normales, las anotaciones no sólo no se ignoran, sino que se tienen en cuenta y pueden influir en la ejecución del código.

El siguiente código muestra un ejemplo sencillo de cómo se utilizan las anotaciones en Symfony:

```
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

// ...

/**
 * @ORM\Column(type="string")
 * @Assert\NotBlank()
 */
protected $nombre;
```

Justo encima de la declaración de la propiedad `$nombre` se añade un comentario de varias líneas. La única diferencia entre un comentario normal y un comentario que incluye anotaciones, es que la apertura del comentario debe tener dos asteriscos (`/**`) en vez de uno solo (`/*`). Si sólo utilizas un asterisco, la aplicación ignorará completamente las anotaciones y perderás mucho tiempo tratando de descubrir por qué no funcionan como deberían.

En este ejemplo, la anotación `@ORM\Column(type="string")` define el tipo de columna de base de datos en la que se guarda este valor. Y la anotación `@Assert\NotBlank()` asegura que esta propiedad no se pueda dejar vacía.

Si nunca has utilizado anotaciones seguramente te estás preguntando por qué se define esta información tan importante en un simple comentario, en vez de incluirla en el propio código de la aplicación. La gran ventaja de las anotaciones es que permiten modificar el comportamiento de la aplicación sin tener que recurrir a archivos de configuración y sin tener que añadir más líneas de código. Se trata por tanto de conseguir un código más conciso y fácil de mantener.

Symfony recomienda utilizar anotaciones en sus aplicaciones siempre que sea posible, pero **no es obligatorio hacerlo**. Si por cualquier motivo no te gustan las anotaciones, puedes definir la configuración usando otros formatos (XML, YAML, PHP). Symfony nunca te va a obligar a usar un formato determinado.

1.5 Introducción a YAML

Los archivos de configuración de Symfony se pueden escribir en PHP, en XML o en YAML. Desde el punto de vista del rendimiento no hay ninguna diferencia entre los tres, ya que todos ellos se transforman a PHP antes de ejecutar la aplicación.

El formato PHP es la forma más complicada de escribir la configuración de la aplicación, pero es la que más flexibilidad proporciona. El formato XML es el más largo de escribir, pero su contenido se puede validar antes de ejecutar la aplicación gracias a los esquemas definidos con XSD.

YAML es probablemente el formato más equilibrado, ya que es mucho más conciso que XML y es bastante flexible. Su gran desventaja es que no se puede validar automáticamente, por lo que la mayoría de los errores sólo puedes descubrirlos al ejecutar la aplicación.

Los archivos de configuración de la aplicación desarrollada en este libro utilizan el formato YAML. Si no conoces YAML, a continuación se explica resumidamente la sintaxis imprescindible para programadores Symfony.

1.5.1 Sintaxis

Los archivos de configuración de Symfony sólo utilizan una parte muy pequeña de todo el estándar YAML (http://yaml.org) , por lo que es muy sencillo aprender su sintaxis. Los cinco conceptos básicos que debes de conocer son los siguientes:

1. Nunca incluyas un tabulador en un archivo YAML. Para indentar la información, utiliza siempre la barra espaciadora.
2. La información se indica mediante pares `clave: valor`. Si la `clave` o el `valor` tienen espacios en blanco, se encierran con comillas simples o dobles indistintamente.
3. La jerarquía de la información se establece escribiendo cuatro espacios en blanco por delante del par `clave: valor`
4. Los arrays *normales* se indican con los corchetes `[` y `]` y los arrays asociativos con las llaves `{` y `}`.
5. Los comentarios se indican prefijando el carácter `#` por delante de cada una de sus líneas.

Observa las primeras líneas del archivo de configuración de la seguridad de Symfony (`app/config/security.yml`):

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]

    providers:
        in_memory:
            users:
                user:  { password: userpass, roles: [ 'ROLE_USER' ] }
                admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }
```

La palabra `security` de la primera línea es la clave principal de este archivo YAML, a partir de la cual se definen el resto de claves y valores. Observa cómo las claves de segundo nivel (`encoders`, `role_hierarchy` y `providers`) van precedidas por cuatro espacios en blanco. Esta es la forma de establecer la jerarquía en un archivo YAML.

Dentro de la clave `role_hierarchy`, el array de valores de la clave `ROLE_SUPER_ADMIN` se define utilizando la notación tradicional de corchetes:

```
security:
    role_hierarchy:
```

```
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Si lo prefieres, existe una notación alternativa para los arrays que consiste en prefijar con un guión medio - todos los valores del array:

```
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN:
            - ROLE_USER
            - ROLE_ADMIN
            - ROLE_ALLOWED_TO_SWITCH
```

Los arrays (tanto *normales* como asociativos) se pueden mezclar entre sí y con otros valores normales, como por ejemplo en las siguientes líneas del archivo anterior:

```
providers:
    in_memory:
        users:
            user:  { password: userpass, roles: ['ROLE_USER'] }
            admin: { password: adminpass, roles: ['ROLE_ADMIN'] }
```

Al procesar el archivo YAML, Symfony convierte las líneas anteriores en el siguiente array PHP:

```
array(
   'security' => array(
      'providers' => array(
         'in_memory' => array(
            'users' => array(
               'user'  => array(
                  'password' => 'userpass',
                  'roles'    => array('ROLE_USER')
               ),

               'admin' => array(
                  'password' => 'adminpass',
                  'roles'    => array('ROLE_ADMIN')
               )
            )
         )
      )
   )
)
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 2

El proyecto

La mejor manera de evaluar un *framework* consiste en utilizarlo para desarrollar aplicaciones reales. Por eso este libro explica Symfony programando una aplicación web de ejemplo pero completa. La aplicación se llama *Cupon* y es un clon inspirado por el sitio *Groupon.com*.

Cupon es un sitio web que publica ofertas de productos con un gran descuento. Cada día se publica una nueva oferta en todas las ciudades de la aplicación. Si antes de que expire la oferta se apuntan un determinado número de personas, la oferta es válida para cualquier usuario. Si no se alcanza el umbral mínimo necesario de personas interesadas, la oferta se anula.

La aplicación *Cupon* dispone de su propio repositorio de código público en Github: https://github.com/javiereguiluz/Cupon/tree/2.8. Resulta muy recomendable que instales la aplicación en tu ordenador para seguir más fácilmente el desarrollo del libro.

2.1 Funcionamiento detallado de la aplicación

La aplicación Cupon se divide en tres partes:

- *Frontend* o *"parte frontal"*, que es el sitio web público al que acceden los usuarios que quieren consultar y comprar las ofertas.
- *Extranet*, que es una parte restringida a la que sólo pueden acceder las tiendas para publicar sus ofertas y comprobar las ventas producidas.
- *Backend* o *"parte de administración"*, a la que sólo pueden acceder los administradores del sitio y en la que pueden crear, consultar o modificar cualquier información sobre ofertas, tiendas y usuarios.

2.1.1 El *frontend*

Para que el *frontend* funcione correctamente, siempre tiene que estar seleccionada una ciudad. Así se puede mostrar la oferta del día, las ofertas recientes, etc. Para determinar cuál es la *ciudad activa* en cada momento, se utiliza la siguiente lógica:

1. La ciudad activa siempre se incluye como parte de la URL de cualquier página de la aplicación.
2. Si al sitio accede un usuario anónimo por primera vez, la aplicación escoge la ciudad por defecto indicada mediante un archivo de configuración.
3. Si un usuario registrado se *loguea* en el sitio, la ciudad activa se cambia por la ciudad asociada al usuario.

4. En cualquier momento, cualquier usuario (anónimo o registrado) puede cambiar la ciudad activa en la aplicación mediante una lista desplegable de ciudades que se incluye en la parte superior de todas las páginas.

Básicamente, el *frontend* se compone de las siguientes siete páginas:

- **Portada**: es el punto de entrada natural al sitio web y también la página que se muestra al pinchar la opción *Oferta del día* en el menú de navegación. Su contenido coincide en gran parte con la página de detalle de la oferta del día en la ciudad activa.
- **Página de detalle de una oferta**: muestra la información básica de la oferta (descripción, foto y precio), la información básica de la tienda donde se puede comprar, una cuenta atrás que muestra el tiempo que falta para que expire, las compras realizadas hasta ese momento y el número mínimo de compras necesarias para activar la oferta.
- **Ofertas recientes**: se accede desde el menú principal y muestra las cinco ofertas más recientes publicadas en la ciudad activa. Si una oferta todavía se puede comprar, se incluye el botón *Comprar* y la cuenta atrás. Si no, se muestra la fecha en la que expiró la oferta.
- **Mis Ofertas**: se accede desde el menú principal y sólo funciona para los usuarios *logueados*. Muestra las últimas ofertas compradas por el usuario.
- **Portada de tienda**: se muestra al pinchar el nombre de una tienda en cualquier página. Incluye la información básica de la tienda y un listado con sus últimas ofertas publicadas.
- **Página de registro**: se muestra al pinchar en el botón *Regístrate* y muestra un formulario vacío con todos los campos que hay que rellenar para registrarse en la aplicación.
- **Página de perfil**: sólo pueden acceder a ella los usuarios *logueados*. Muestra un formulario con toda la información pública del usuario y permite modificar cualquier dato.

Por último, cuando un usuario anónimo pincha en *Mis ofertas* o en el botón de *Comprar* de alguna oferta, se le redirige a una página especial en la que puede *loguearse* o iniciar el proceso de registro.

2.1.2 La *extranet*

Para poder acceder a cualquier página de la *extranet*, es necesario proporcionar un nombre de usuario y contraseña que pertenezcan a un usuario de tipo `tienda`. Además, la *ciudad activa* se establece automáticamente en función de la ciudad a la que pertenezca la tienda *logueada*.

Las cinco páginas que forman la extranet son las siguientes:

- **Portada**: muestra un listado con las ofertas más recientes publicadas por la tienda. Incluye enlaces para modificar las ofertas y para ver el listado de sus ventas.
- **Mis datos**: muestra un formulario con toda la información de la tienda y permite modificar cualquier dato.
- **Página para añadir oferta**: se muestra al pinchar en el botón *Añadir oferta*. Muestra un formulario vacío con todos los campos que debe rellenar la tienda para publicar una oferta.

- **Página para modificar oferta**: sólo se muestra para aquellas ofertas que todavía no han sido revisadas por los administradores del sitio. Muestra y permite modificar cualquier información de la oferta, incluyendo su foto.
- **Página de ventas**: sólo está disponible para aquellas ofertas publicadas y de las que se haya realizado al menos una venta. Muestra la información básica del comprador y la fecha y hora de la compra.

2.1.3 El *backend*

El *backend* permite gestionar la información de las ofertas, las tiendas, los usuarios y las ciudades de la aplicación. Se trata de la zona más restringida de la aplicación, a la que sólo pueden acceder los usuarios que tengan un permiso especial de tipo administrador. Como los administradores pueden modificar cualquier información, en la parte superior de todas las páginas del *backend* se muestra una lista desplegable para seleccionar la *ciudad activa*.

2.1.4 Límites prácticos

Aunque la aplicación que se desarrolla es lo más completa posible, resulta necesario definir varios límites prácticos debido a las limitaciones de espacio de este libro:

- Cada día sólo se puede publicar una oferta en cada ciudad.
- Una misma oferta sólo puede estar activa en una única ciudad.
- Las tiendas no se pueden registrar libremente en el sitio web, ya que se supone que deben pasar por un proceso manual de verificación.
- Las tiendas sólo pueden publicar ofertas en su ciudad.
- El precio de las ofertas sólo se indica en euros y no se tienen en cuenta los impuestos al consumo.
- Los usuarios se pueden registrar libremente a través del sitio web, sin tener que validar el registro mediante correo electrónico.
- Cuando un usuario compra una oferta, su coste se carga instantáneamente en su tarjeta de crédito, sin tener que pasar por una pasarela de pago.

2.2 Wireframes

Tras definir el funcionamiento de la aplicación, el siguiente paso consiste en decidir la estructura y contenidos de cada página. Para ello se presentan a continuación los *wireframes* de las principales páginas del sitio web.

Un *wireframe* describe los contenidos de una página sin fijarse en su diseño gráfico. Lo único que importa en un *wireframe* son los contenidos que se incluyen, su tamaño y su posición. Como el aspecto gráfico es irrelevante en esta fase, los *wireframes* siempre se diseñan en blanco y negro y con bloques rectangulares de contenidos.

Figura 2.1 Wireframe de la portada

Figura 2.2 Wireframe de la página de detalle de una oferta

Figura 2.3 Wireframe de la página de ofertas recientes de una ciudad

Figura 2.4 Wireframe de la página de tus últimas compras

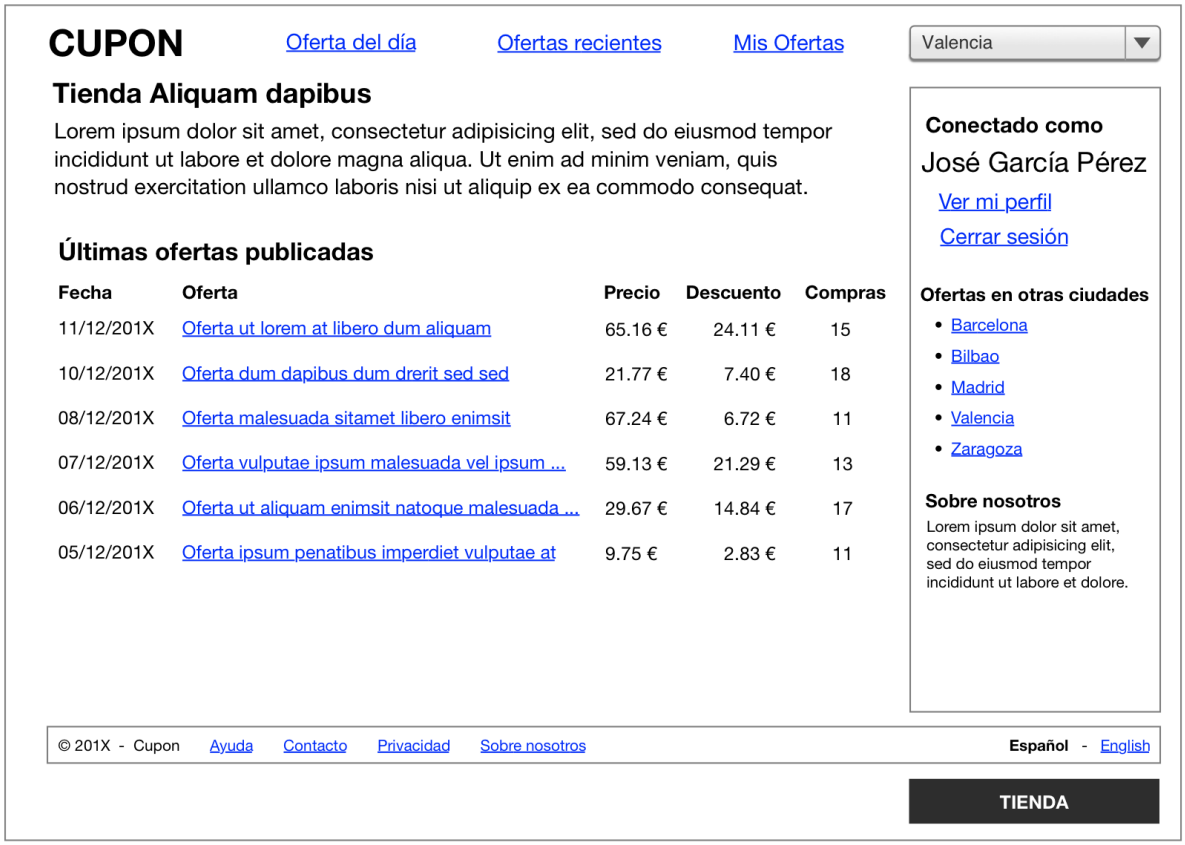

Figura 2.5 Wireframe de la página con información de una tienda

2.3 La base de datos

A partir de los requisitos funcionales definidos anteriormente, la información de la aplicación se almacena en una base de datos relacional con las siguientes cinco tablas:

- `Ciudad`, almacena la información de una ciudad.
- `Oferta`, almacena toda la información del producto o servicio que se oferta y lleva la cuenta de las ventas producidas.
- `Tienda`, almacena los datos de los establecimientos que publican las ofertas.
- `Usuario`, almacena el perfil completo de los usuarios registrados del sitio.
- `Venta`, almacena la información básica de cada venta producida.

Las relaciones entre las tablas son las siguientes:

- `Ciudad` - `Oferta`: `1-n`, una oferta sólo se puede publicar en `1` ciudad, pero una ciudad puede tener `n` ofertas publicadas.
- `Ciudad` - `Tienda`: `1-n`, una tienda sólo se puede asociar con `1` ciudad, pero una ciudad puede tener `n` tiendas asociadas.
- `Ciudad` - `Usuario`: `1-n`, un usuario sólo se puede asociar con `1` ciudad, pero una ciudad puede tener `n` usuarios asociados.

- `Tienda` - `Oferta`: `1-n`, una oferta sólo se puede publicar en `1` tienda, pero una tienda puede publicar `n` ofertas.
- `Oferta` - `Usuario`: `n-n`, un usuario puede comprar `n` ofertas y una oferta puede tener `n` compradores. Esta relación se establece a través de la tabla intermedia `Venta`, que a su vez está relacionada `1-n` con las ofertas y los usuarios.

A continuación se muestran todos los campos de información de cada tabla y sus tipos:

Tabla *Ciudad*

Columna	Tipo	Comentarios
`id`	`int`	clave primaria
`nombre`	`varchar(100)`	
`slug`	`varchar(100)`	

Tabla *Oferta*

Columna	Tipo	Comentarios
`id`	`int`	clave primaria
`nombre`	`varchar(255)`	
`slug`	`varchar(255)`	
`descripcion`	`text`	
`condiciones`	`text`	
`rutaFoto`	`varchar(255)`	sólo se guarda la ruta de la foto
`precio`	`decimal(10,2)`	en euros
`descuento`	`decimal(10,2)`	en euros, no en porcentaje
`fechaPublicacion`	`datetime`	
`fechaExpiracion`	`datetime`	
`compras`	`int`	contador de las ventas realizadas
`umbral`	`int`	compras mínimas para que se active la oferta
`revisada`	`boolean`	
`ciudad_id`	`int`	relación `1:n`
`tienda_id`	`int`	relación `1:n`

Tabla *Tienda*

Columna	Tipo	Comentarios
`id`	`int`	clave primaria

Columna	Tipo	Comentarios
nombre	varchar(100)	
slug	varchar(100)	
login	varchar(10)	nombre de usuario para acceder a la extranet
password	varchar(255)	
descripcion	text	
direccion	text	
ciudad_id	int	relación 1:n

Tabla *Usuario*

Columna	Tipo	Comentarios
id	int	clave primaria
nombre	varchar(100)	
apellidos	varchar(255)	
email	varchar(255)	utilizado también como *login*
password	varchar(255)	
direccion	text	
permiteEmail	boolean	indica si el usuario permite comunicaciones publicitarias
fechaAlta	datetime	
fechaNacimiento	datetime	
dni	varchar(9)	documento nacional de identidad
numeroTarjeta	varchar(20)	número de la tarjeta de crédito
ciudad_id	int	relación 1:n

Tabla *Venta*

Columna	Tipo	Comentarios
fecha	datetime	
oferta_id	int	relación 1:n, parte de la clave primaria compuesta
usuario_id	int	relación 1:n, parte de la clave primaria compuesta

2.4 Aplicando la filosofía de Symfony

Antes de empezar a programar una aplicación con cualquier *framework*, es necesario pensar cómo encajar las funcionalidades a desarrollar con la filosofía de trabajo del *framework*.

Symfony no es ninguna excepción, pero al menos es el framework PHP más "desacoplado", ya que permite separar el código de tu aplicación y el código que debes escribir para integrarla con Symfony. De hecho, la propia documentación oficial de Symfony desaconseja "acoplarse" al *framework* para crear mejores aplicaciones.

En las siguientes secciones se explica cómo adaptar todas las funcionalidades y *wireframes* anteriores a la filosofía de trabajo de Symfony. El orden recomendado consiste en definir primero las entidades, después los *bundles* y por último el enrutamiento.

Si estás empezando con Symfony, es posible que no entiendas bien alguno de los conceptos explicados en las próximas secciones. No te preocupes demasiado, ya que conceptos como *bundle* son difíciles de explicar en abstracto, pero te resultarán intuitivos cuando hayas desarrollado alguna aplicación con Symfony.

2.5 Entidades

El código PHP de las aplicaciones Symfony no interactúa directamente con las bases de datos. Por eso la información no se gestiona con sentencias SQL sino mediante objetos PHP. Estos objetos se denominan técnicamente *entidades*. Definir las *entidades* de la aplicación consiste en crear clases PHP que representen la estructura de las tablas asociadas de la base de datos.

Aunque se trata de lo primero que debes hacer al planificar la aplicación, no podrás hacerlo hasta que no llegues al capítulo 5 (página 63), donde se explica detalladamente el funcionamiento de las bases de datos en las aplicaciones Symfony.

2.6 Bundles

Los *bundles* son uno de los conceptos clave del funcionamiento de Symfony. Técnicamente, un *bundle* es un directorio que contiene todo tipo de archivos dentro una estructura jerarquizada de directorios.

Los *bundles* son similares a los *plugins* de otras aplicaciones y tecnologías. De hecho, normalmente los *bundles* se utilizan para añadir en tu aplicación funcionalidades desarrolladas por terceros. La gran diferencia es que el propio código de las aplicaciones Symfony está dividido en *bundles*.

Los *bundles* de las aplicaciones Symfony suelen contener clases PHP y archivos web (JavaScript, CSS e imágenes). No obstante, no existe ninguna restricción sobre lo que puedes incluir dentro de un *bundle*. Tampoco existen límites técnicos sobre el tamaño que puede llegar a tener un *bundle*.

Algunos programadores prefieren almacenar todo el código de la aplicación en un único *bundle* para simplificar la gestión del código. Otros programadores prefieren usar tantos *bundles* como *divisiones lógicas* tenga la aplicación. Así por ejemplo, la aplicación se podría dividir en tres *bundles*: FrontendBundle, ExtranetBundle y BackendBundle. También se podría dividir en cinco bundles: CiudadBundle, OfertaBundle, TiendaBundle, UsuarioBundle y BackendBundle.

En las versiones anteriores de este libro, la aplicación se divide en esos cinco *bundles*. El resultado es un código muy modular, pero innecesariamente complicado. En esta nueva edición del libro se

sigue la nueva "buena práctica" recomendada por Symfony: **crear un único bundle llamado AppBundle**.

No te obsesiones con tomar una decisión *perfecta* sobre cuántos *bundles* utilizas para dividir tu aplicación. Ésta va a seguir funcionando bien sea cual sea su número de *bundles*. Si tienes dudas, sigue la recomendación de crear un único *bundle* llamado AppBundle y ya lo dividirás más adelante si es necesario.

2.7 Enrutamiento

Definir todas las rutas posibles del sitio web es el último paso de la planificación previa al desarrollo de la aplicación. En Symfony cada ruta debe tener un nombre único y preferiblemente muy corto, para que el código de las plantillas sea más conciso. Además, cuando una ruta contiene partes que varían de una página a otra (como por ejemplo el nombre de una ciudad o de una oferta), estas se indican mediante variables encerradas entre `{` y `}`. A continuación se listan las diferentes rutas de la aplicación:

Frontend

Nombre	URL
portada	/{ciudad}
oferta	/{ciudad}/ofertas/{slug}
comprar	/{ciudad}/ofertas/{slug}/comprar
tienda_portada	/{ciudad}/tiendas/{tienda}
ciudad_cambiar	/ciudad/cambiar-a-{ciudad}
ciudad_recientes	/{ciudad}/recientes
usuario_registro	/usuario/registro
usuario_perfil	/usuario/perfil
usuario_compras	/usuario/compras
usuario_login	/usuario/login
usuario_login_check	/usuario/login_check
usuario_logout	/usuario/logout
contacto	/contacto
estatica	/sitio/{pagina}

Más adelante, cuando se traduzca el *frontend* a varios idiomas, todas las rutas anteriores incluirán el código del idioma seleccionado. Así, la ruta `/{ciudad}` será `/es/{ciudad}`, `/en/{ciudad}`, `/fr/{ciudad}`, etc.

Extranet

Nombre	URL
extranet_portada	/extranet
extranet_oferta_nueva	/extranet/oferta/nueva
extranet_oferta_editar	/extranet/oferta/editar/{id}
extranet_oferta_ventas	/extranet/oferta/ventas/{slug}
extranet_perfil	/extranet/perfil
extranet_login	/extranet/login
extranet_login_check	/extranet/login_check
extranet_logout	/extranet/logout

Las rutas del *backend* se generan automáticamente con las herramientas que se explicarán más adelante, así que no hay que definirlas explícitamente.

Esta página se ha dejado vacía a propósito

CAPÍTULO 3

Instalando y configurando Symfony

El *framework* Symfony requiere para su funcionamiento de varias librerías PHP. Algunas de estas librerías han sido creadas por el propio proyecto Symfony y se llaman **Componentes Symfony**. Otras son librerías desarrolladas por terceros, como **Monolog** (para gestionar los mensaje de log), **Doctrine** (para gestionar las bases de datos), etc.

En la práctica, una aplicación Symfony se compone de dos elementos principales: tu propio código fuente (normalmente se guarda en `src/`) y el código de todas esas librerías que necesita Symfony (normalmente se guardan en `vendor/`). Así que "instalar Symfony" consiste en crear una aplicación PHP vacía e instalar las librerías necesarias dentro del directorio `vendor/`.

3.1 El instalador de Symfony

El proyecto Symfony ha publicado una herramienta llamada **Instalador de Symfony** que simplifica al máximo la creación de nuevas aplicaciones Symfony. Se trata de una pequeña aplicación creada con PHP 5.4 y que solamente hay que instalar una vez en tu ordenador.

```
# Si utilizas Linux o macOS, ejecuta los siguientes comandos:
$ sudo mkdir -p /usr/local/bin
$ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
$ sudo chmod a+x /usr/local/bin/symfony

# Si usas Windows, ejecuta el siguiente comando:
c:\> php -r "readfile('https://symfony.com/installer');" > symfony
```

En Linux y macOS, los comandos anteriores crean un comando global llamado `symfony`. En Windows, mueve el archivo `symfony` a algún directorio que se encuentre dentro de la variable de entorno `PATH` para poder ejecutarlo sin tener que indicar su ruta completa.

3.2 Creando la aplicación Symfony

Una vez disponible el instalador de Symfony, puedes crear aplicaciones Symfony ejecutando un solo comando sencillo:

```
# Si utilizas Linux o macOS, ejecuta los siguientes comandos:
$ cd proyectos/
$ symfony new cupon 2.8
```

```
# Si utilizas Windows, ejecuta los siguientes comandos:
c:\> cd proyectos\
c:\proyectos\> symfony new cupon 2.8
```

El comando `symfony new` crea una nueva aplicación Symfony vacía en el directorio indicado como argumento (`cupon` en este caso). El tercer argumento (`2.8` en este ejemplo) indica la versión de Symfony a instalar. Si no indicas la versión, se instalará la versión más reciente de Symfony disponible en ese momento:

```
# utilizar la versión más reciente de Symfony, sea cual sea:
$ symfony new cupon

# utilizar la versión más reciente en una determinada "rama" de desarrollo:
$ symfony new cupon 3.1

# utilizar una versión exacta de Symfony:
$ symfony new cupon 2.5.6

# utilizar la versión más reciente de Symfony que tenga soporte muy largo
# (en inglés, se denomina versión "Long Term Support" o LTS)
$ symfony new cupon lts
```

3.2.1 Creando la aplicación Symfony con Composer

Si por cualquier motivo no puedes utilizar el instalador de Symfony, es posible crear la aplicación Symfony con Composer (asegúrate de instalar primero Composer tal y como se explica en el primer capítulo de este libro):

```
# Si utilizas Linux o macOS, ejecuta los siguientes comandos:
$ composer create-project symfony/framework-standard-edition cupon 2.8.*

# Si utilizas Windows, ejecuta los siguientes comandos:
C:\> composer create-project symfony/framework-standard-edition cupon 2.8.*
```

La instalación de Symfony con Composer puede tardar hasta varios minutos (con el instalador son solo unos segundos) y el proceso es más complicado, ya que Composer te obliga a definir el valor de varias opciones de configuración. Por eso se recomienda utilizar el instalador de Symfony siempre que sea posible.

3.3 Actualizando la aplicación Symfony

El código fuente del *framework* Symfony se actualiza cada día con mejoras y correcciones de errores, por lo que es recomendable que actualices Symfony habitualmente. Gracias a Composer actualizar Symfony es realmente sencillo:

```
# Si utilizas Linux o macOS, ejecuta los siguientes comandos:
$ cd proyectos/cupon/
```

```
$ composer update

# Si utilizas Windows, ejecuta los siguientes comandos:
C:\> cd proyectos\cupon\
C:\proyectos\cupon\> composer update
```

El comando `composer update` lee la configuración del archivo `composer.json` y actualiza todas las versiones de todas las librerías, siempre que cumplan las restricciones impuestas por tu aplicación. Después, actualiza el contenido del archivo `composer.lock`, que es el que guarda la versión exacta que se ha instalado para cada dependencia.

TRUCO Además de las versiones menores (ej. 2.8.4, 3.1.5) que se publican cada mes, Symfony publica dos versiones principales (ej. 2.8, 3.2) cada año; en mayo y noviembre. Visita https://symfony.com/roadmap para conocer todos los detalles sobre el proceso de lanzamiento de versiones de Symfony.

3.4 Instalado una aplicación Symfony existente

Si trabajas en equipo, normalmente no crearás aplicaciones Symfony desde cero sino que instalarás aplicaciones creadas por otras personas. En este caso, lo habitual es descargar el código fuente de la aplicación desde un repositorio compartido e instalar después las librerías de Symfony en el directorio `vendor/`.

A modo de ejemplo, se va a instalar la aplicación *Cupon* desde su repositorio en GitHub: https://github.com/javiereguiluz/Cupon. Para ello:

1. Crea un directorio para guardar el código de la aplicación (`mkdir cupon`)
2. Descarga la aplicación clonando con Git su repositorio oficial (`git clone https://github.com/javiereguiluz/Cupon.git cupon`)
3. Accede al directorio de la aplicación (`cd cupon`)
4. Selecciona la rama correspondiente a Symfony 2.8 (`git checkout 2.8`)
5. Descarga las dependencias de Symfony (`composer install`)

El comando `composer install` es necesario porque en el repositorio compartido nunca se suben los contenidos del directorio `vendor/` (ocupa demasiado sitio y es mejor dejar que cada programador se instale las librerías en su ordenador).

Si al repositorio solamente subes el archivo `composer.json`, el comando `composer install` instala las versiones más recientes posibles de cada dependencia. Si también subes el archivo `composer.lock` al repositorio, Composer instala exactamente las versiones indicadas en ese archivo.

Cuando se trabaja en equipo, se recomienda subir tanto `composer.json` como `composer.lock` al repositorio para que todos los miembros del equipo usen exactamente las mismas versiones de todas las dependencias.

3.5 Comprobando la instalación de Symfony

Tanto si has creado una aplicación Symfony nueva como si has instalado una aplicación existente, antes de empezar a programar debes asegurarte de que tu ordenador cumple con los requisitos técnicos para que Symfony funcione bien.

Para ello, entra en el directorio del proyecto (`cupon/` en este ejemplo) y ejecuta el siguiente comando:

```
$ cd proyectos/cupon/
$ php app/check.php
```

El script `check.php` muestra por consola una lista de requisitos obligatorios (*Mandatory requirements*) y otra de requisitos deseables (*Optional checks*) para ejecutar Symfony. No sigas adelante si incumples alguno de los requisitos obligatorios. Cuando ejecutes la aplicación en el servidor de producción, asegúrate también de cumplir todos los requisitos deseables.

Después de esta comprobación, ya puedes ejecutar el comando `php app/console`, que es la herramienta de consola principal de Symfony. Si funciona bien, tienes que ver en tu consola algo como lo siguiente:

```
$ php app/console
Symfony version 2.8.9 - app/dev/debug

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  ...
```

Resulta imprescindible que el comando `php app/console` se ejecute correctamente, ya que Symfony utiliza la línea de comandos para automatizar muchas tareas importantes. Todos los comandos de Symfony se ejecutan mediante `php app/console <nombre-del-comando>`

3.6 Accediendo a la aplicación Symfony

Si dispones de PHP 5.4 o superior, puedes aprovechar el servidor web interno que incluye el propio PHP para acceder a la aplicación Symfony sin tener que configurar ningún servidor web como Apache o Nginx. Simplemente entra en el directorio del proyecto y ejecuta el siguiente comando:

```
$ cd cupon/
$ php app/console server:run

 [OK] Server running on http://127.0.0.1:8000
 // Quit the server with CONTROL-C.
```

Ahora ya puedes abrir un navegador y acceder a la URL `http://127.0.0.1:8000` para probar tu aplicación. Si has creado una aplicación Symfony nueva, verás la página de bienvenida de Symfony:

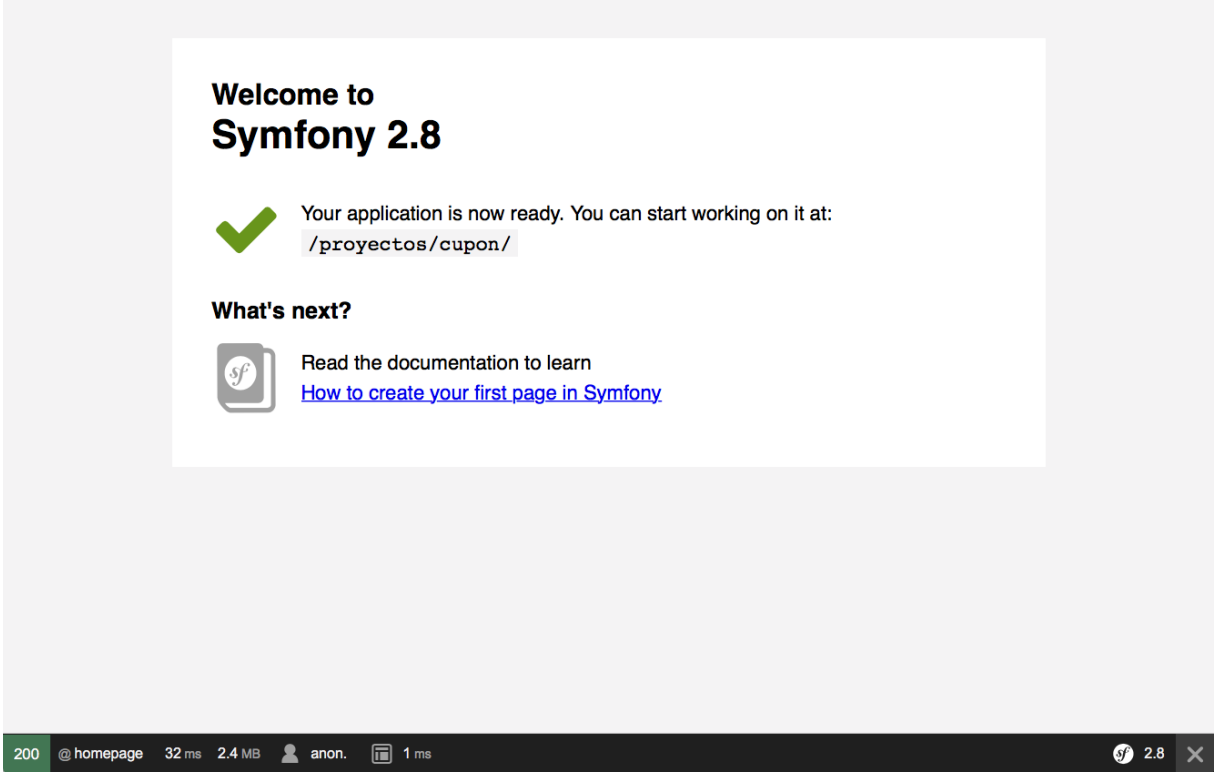

Figura 3.1 Página de bienvenida de Symfony

NOTA Si en vez de la portada de Symfony ves un mensaje de error, es muy posible que se trate de un problema de permisos. Más adelante, en la sección *Configurando los permisos* (página 60) se explica con detalle la causa de este error y cómo solucionarlo.

Utilizar el servidor web de PHP es la forma recomendada de acceder a las aplicaciones Symfony mientras las desarrollas en tu propio ordenador. A partir de ahora, todos los ejemplos de este libro utilizan `127.0.0.1:8000` en sus URL.

Si prefieres utilizar un servidor web *de verdad*, como Apache o Nginx, lee el artículo Configuring a Web Server (http://symfony.com/doc/current/setup/web_server_configuration.html) de la documentación oficial de Symfony para saber cómo configurarlo.

3.7 Estructura de las aplicaciones Symfony

Una vez instalado Symfony, entra en el directorio del proyecto y observa su jerarquía de directorios. Esta jerarquía es común a todos los proyectos Symfony y está compuesta por:

Directorio	Propósito
`app/`	Contiene los archivos de configuración, la caché, los logs y los recursos globales.
`app/config/`	Guarda todos los archivos de configuración de la aplicación.

Directorio	Propósito
`app/cache/`	Contiene todos los archivos cacheados por Symfony (clases, enrutamiento, plantillas, entidades, validación, etc.). Junto con el directorio `app/logs/` es el único en el que Symfony debe tener permisos de escritura.
`app/logs/`	Contiene los archivos de log generados al ejecutar la aplicación. Junto con el directorio `app/cache/` es el único en el que Symfony debe tener permisos de escritura.
`app/Resources/`	Almacena los recursos que se utilizan globalmente en el proyecto (como por ejemplo las plantillas).
`src/`	Guarda todo el código fuente propio de tu aplicación.
`vendor/`	Contiene todo el código fuente de Symfony y de todas las librerías externas.
`web/`	El único directorio público del proyecto. Contiene los archivos web (CSS, JavaScript e imágenes) y los controladores frontales de la aplicación (`app.php` y `app_dev.php`)

Sección 2

Frontend

Esta página se ha dejado vacía a propósito

CAPÍTULO 4

Creando las primeras páginas

4.1 La filosofía de Symfony

La principal crítica que se suele hacer a los frameworks web es que son muy intrusivos, ya que obligan al programador a trabajar de una determinada manera. Symfony ha sido diseñado para evitar este problema y para interferir lo menos posible en el *flujo natural* de las peticiones HTTP.

Si un sitio web está compuesto por páginas estáticas, su funcionamiento es el siguiente:

1. El **usuario** solicita una página mediante su URL
2. El **servidor web** devuelve el contenido de la página que corresponde a la URL

En el caso de los sitios web dinámicos controlados por una aplicación:

1. El **usuario** solicita una página mediante su URL
2. La **aplicación** busca o genera el contenido que corresponde a la URL
3. El **servidor web** devuelve el contenido que le pasa la aplicación

Si la aplicación está programada con Symfony:

1. El **usuario** solicita una página mediante su URL
2. **Symfony** convierte la petición en un objeto PHP de tipo `Request` y se lo pasa a la aplicación
3. La **aplicación** busca o genera el contenido que corresponde a la petición y devuelve a Symfony la respuesta mediante un objeto PHP de tipo `Response`
4. **Symfony** convierte el objeto `Response` en el contenido que se pasa al servidor web (normalmente una página HTML)
5. El **servidor web** devuelve el contenido que le pasa Symfony

Symfony se adapta perfectamente al modelo de petición + respuesta HTTP. La gran diferencia respecto al código PHP tradicional, es que no debes lidiar con funciones como `header()`, `setcookie()`, etc. y con los super arrays `$_GET`, `$_POST`, etc. Symfony convierte toda esa información HTTP en dos objetos PHP: `Request` y `Response`.

Toda la arquitectura de Symfony se ha diseñado para facilitar la creación de un objeto de tipo `Response` a partir de un objeto de tipo `Request`. Fuera de este ámbito, Symfony *desaparece* y deja que sea el programador el que realice su trabajo como quiera.

Si es la primera vez que utilizas un *framework*, después de leer este capítulo puedes tener la sensación de que no son más que una molestia, que te obligan a trabajar mucho y que ni solucionan tantos problemas ni aportan tantas ventajas. La documentación oficial de Symfony dispone de un artículo que explica por qué es mejor usar un framework como Symfony (http://librosweb.es/symfony_2_4/capitulo_2.html) en vez de programar todo a mano con PHP.

Dentro de dos capítulos se explica con todo detalle el funcionamiento interno de Symfony. Por el momento sólo se explica el ejemplo más sencillo posible para mostrar el contenido de una página.

4.2 La primera página

La primera página que se va a desarrollar es la página de ayuda del sitio web, que se muestra cuando el usuario accede a `http://127.0.0.1:8000/ayuda`. Para ello, es necesario hacer lo siguiente:

1. Crear el código PHP necesario para mostrar el contenido de la página de ayuda.
2. Asociar de alguna manera la URL `/ayuda` con el código anterior.

En Symfony, el código PHP anterior se implementa como un **método** de una **clase** PHP. La clase se llama **controlador** y el método se llama **acción**. Por otra parte, la asociación entre URL y métodos PHP se hace mediante las **rutas** del sistema de enrutamiento.

En la práctica, todo este flujo de trabajo es mucho más fácil de lo que parece. Así que para crear esta primera página, copia y pega el siguiente código PHP en el archivo `src/AppBundle/Controller/DefaultController.php`:

```
<?php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController
{
    /**
     * @Route("/ayuda")
     */
    public function ayudaAction()
    {
        return new Response('Página de ayuda');
    }
}
```

Si ahora accedes a `http://127.0.0.1:8000/ayuda`, verás en el navegador el mensaje *Página de ayuda*. Enhorabuena, acabas de programar tu primera página con Symfony! Si ves un mensaje de error o una pantalla en blanco, lee la sección *Configurando los permisos* (página 60) de este mismo capítulo. Si te funciona bien, sigue leyendo para comprender qué ha sucedido internamente en Symfony para que puedas ver ese mensaje.

En primer lugar, los **controladores** se definen por convención como clases PHP cuyo nombre acaba en `Controller` y se guardan dentro del directorio `Controller/` del *bundle*. Puedes crear tantos controladores como necesites y Symfony siempre crea por defecto un controlador llamado `DefaultController`.

Las **acciones**, que son los métodos PHP que se ejecutan para responder a las peticiones de los usuarios, se definen como métodos públicos cuyo nombre acaba en `action`. Puedes crear tantas acciones como necesites en un controlador, aunque si creas muchas acciones, puede ser mejor dividirlas en varios controladores.

Las **rutas** se definen como anotaciones de las **acciones**. En este caso, la anotación `@Route("/ayuda")` es muy simple y solo indica que el método `ayudaAction()` está asociado con la URL `/ayuda`. Observa como `@Route` es en realidad una clase PHP y por eso para usarla hay que importarla mediante la instrucción `use` correspondiente.

La gran ventaja de las anotaciones es que son muy concisas y que puedes ver fácilmente la ruta de cualquier acción sin tener que abrir un archivo de configuración separado. De todas formas, si no te gustan las anotaciones, Symfony también te permite definir las rutas en los formatos YAML, XML y PHP.

Las **acciones** de los controladores siempre deben devolver un objeto de tipo `Response`. Si cambias el código anterior por algo como `return 'Página de ayuda';` Symfony mostrará un mensaje de error. Así que primero importa la clase `Response` con el `use` correspondiente (si utilizas el editor PHPStorm, se importa automáticamente) y luego pásale como argumento el mensaje a mostrar.

NOTA En este caso sencillo, tener que instanciar la clase `Response` en vez de devolver una simple cadena de texto puede parecer demasiado complicado. En efecto lo es. Pero ten en cuenta que en las aplicaciones web reales nunca devuelves una cadena de texto simple como respuesta a una URL, sino que devuelves toda una página HTML, un contenido codificado como JSON, etc.

4.2.1 Respondiendo con plantillas

Las aplicaciones web reales nunca devuelven cadenas de texto como respuesta a una petición de una URL. Lo normal es devolver toda una página HTML. Para ello, modifica de nuevo el archivo `DefaultController.php` pegando el siguiente contenido:

```
<?php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;
```

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/ayuda")
     */
    public function ayudaAction()
    {
        return $this->render('sitio/ayuda.html.twig');
    }
}
```

Si ahora actualizas la página de tu navegador (`http://127.0.0.1:8000/app_dev.php/ayuda`), ya no verás el contenido *Página de ayuda* sino que se mostrará un error indicando que Symfony no ha podido encontrar la plantilla `ayuda.html.twig`.

Las páginas de las aplicaciones dinámicas se crean mediante plantillas. Estas no son más que los *moldes* con los que se *fabrican* las páginas HTML que realmente se entregan a los usuarios. En este ejemplo resulta absurdo utilizar una plantilla, ya que la página de ayuda siempre es igual para cualquier usuario. Pero si piensas en la página de detalle de una oferta o de una tienda, está claro que la página siempre tiene el mismo aspecto y estructura, pero sus contenidos varían, así que es necesario utilizar plantillas para crear la página específica de cada oferta o tienda.

En Symfony las plantillas normalmente tienen dos extensiones. La primera es el lenguaje con el que se crean sus contenidos. Casi siempre se utiliza el lenguaje HTML, pero también es posible crear contenido XML, JavaScript, JSON, CSS, etc.

La segunda extensión indica el lenguaje con el que se añade la programación a las plantillas. Aunque puedes crear las plantillas con PHP, Symfony recomienda utilizar Twig (http://twig.sensiolabs.org/) . Se trata de un lenguaje de plantillas moderno, seguro, rápido y con el que puedes crear plantillas concisas y muy fáciles de mantener. En los próximos capítulos se explicará con detalle cómo crear plantillas con Twig.

Volviendo al mensaje de error mostrado por Symfony, el motivo es que Symfony no ha sido capaz de encontrar la plantilla `sitio/ayuda.html.twig`. Symfony recomienda por defecto almacenar las plantillas de la aplicación en el directorio `app/Resources/views/`.

Así que para solucionar el error que muestra Symfony, simplemente debes crear el archivo `app/Resources/views/sitio/ayuda.html.twig`. Añade cualquier contenido a ese archivo, guarda los cambios y recarga el navegador. El error de Symfony desaparece y en su lugar estarás viendo el mismo contenido que has añadido al archivo de la plantilla.

4.2.2 El controlador base de Symfony

Tal y como se explicó en las secciones anteriores, las **acciones** de Symfony deben devolver un objeto de tipo `Response` (que Symfony convierte después en una cadena de texto con la respuesta HTTP que se entrega al servidor web).

Sin embargo, en el ejemplo anterior la respuesta de la acción es:

```
return $this->render('sitio/ayuda.html.twig');
```

¿Qué es el método `render()`? ¿Cómo puede funcionar ese código si no has definido en el método `render()` en la clase de tu controlador? El método `render()` es uno de los **atajos** que Symfony define para las tareas más comunes que se realizan en los controladores, como por ejemplo renderizar una plantilla.

Si te fijas en la parte superior del código de `DefaultController`, verás que ahora extiende de una clase llamada `Controller` (y que hay que importar con la instrucción `use`). Esta clase es la que contiene todos los atajos definidos por Symfony y es común que los controladores hereden de ella (aunque no es obligatorio hacerlo).

Internamente, el atajo `render()` busca la plantilla que le indicas, la transforma en el código HTML utilizando las variables que le pasas (en este caso sencillo no se le ha pasado ninguna). Después, crea un objeto de tipo `Respose()` y añade como contenido el resultado de renderizar la plantilla. Así que `$this->render(...)` es equivalente a `new Response('<html>...</html>')`.

4.2.3 Sirviendo todas las páginas del sitio

Además de la página de ayuda, el sitio web que se está desarrollando tiene otras páginas como "Privacidad" y "Sobre nosotros". Siguiendo la misma lógica explicada anteriormente, podrías crear una nueva acción para cada página:

```
class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/privacidad")
     */
    public function privacidadAction()
    {
        return $this->render('sitio/privacidad.html.twig');
    }
}
```

Aunque esta solución funciona correctamente, es muy poco eficiente. Además, sería inviable si el sitio tuviera muchas páginas. La solución es sencilla y consiste en reutilizar la misma acción para servir muchas páginas diferentes.

En primer lugar, crea las tres plantillas de las páginas del sitio con cualquier contenido de prueba:

```
app/Resources/views/sitio/ayuda.html.twig
app/Resources/views/sitio/privacidad.html.twig
app/Resources/views/sitio/sobre_nosotros.html.twig
```

A continuación, modifica el controlador `DefaultController` renombrando la acción `ayudaAction()` a `paginaAction()`:

```
// src/AppBundle/Controller/DefaultController.php

// Antes
public function ayudaAction()

// Después
public function paginaAction()
```

Después, cambia el nombre de la plantilla que se renderiza para que no sea siempre `ayuda` sino que sea el valor que contiene una variable llamada `$nombrePagina`:

```
// src/AppBundle/Controller/DefaultController.php

// Antes
public function paginaAction()
{
    return $this->render('sitio/ayuda.html.twig');
}

// Después
public function paginaAction()
{
    return $this->render('sitio/'.$nombrePagina.'.html.twig');
}
```

La última parte consiste en obtener el valor de la variable `$nombrePagina` que sea adecuado en cada caso. Cuando el usuario quiera ver la página de ayuda, esta variable debe valer `ayuda`, cuando quiera ver la página de privacidad, la variable debe ser `privacidad`, etc.

NOTA Las variables de tipo `$nombrePagina` normalmente se llaman `$slug` en las aplicaciones web reales. El término inglés *"slug"* se refiere a una cadena de texto transformada para que sea seguro incluirla en la URL: la cadena se pasa a minúsculas, se eliminan los espacios en blanco, los acentos, los caracteres problemáticos, etc.

Para ello, en primer lugar se va a hacer un pequeño cambio en las URL del sitio web. En vez de acceder directamente a la URL `/ayuda`, ahora se va a acceder a `/sitio/ayuda`. Así que las tres URL que se deben servir con la acción `paginaAction()` son:

```
/sitio/ayuda
/sitio/privacidad
/sitio/sobre_nosotros
```

Examinando las URL es obvio que todas ellas tienen una parte fija (`/sitio`) y luego otra parte variable que es precisamente el nombre de la página que se quiere ver. Este razonamiento es justo el que debes hacer al definir las rutas de las aplicaciones Symfony. Después, convierte el resultado en la anotación `@Route` correspondiente. En este caso, el resultado es:

```
// src/AppBundle/Controller/DefaultController.php

// Antes
/**
 * @Route("/sitio/ayuda")
 */
public function paginaAction()
{
    return $this->render('sitio/'.$nombrePagina.'.html.twig');
}

// Después
/**
 * @Route("/sitio/{nombrePagina}")
 */
public function paginaAction()
{
    return $this->render('sitio/'.$nombrePagina.'.html.twig');
}
```

Cuando una ruta encierra alguna de sus partes con `{` y `}`, esa parte puede tomar cualquier valor, que se guarda en una variable con el mismo nombre que se indica en la ruta (en este caso, `nombrePagina`).

En este momento, si el usuario accede por ejemplo a `/sitio/privacidad`, el sistema de enrutamiento de Symfony considera que esta es la ruta que se ha solicitado y crea una variable llamada `$nombrePagina` con el valor `privacidad`. Sin embargo, el código PHP de la acción no puede acceder mágicamente a esta variable. De hecho, si recargas la página verás un mensaje de error diciendo que la variable `$nombrePagina` no existe.

La solución que propone Symfony es sencilla y elegante. Para pasar variables desde las rutas a las acciones, añádelas como argumentos de la acción:

```
// src/AppBundle/Controller/DefaultController.php

// Antes
/**
 * @Route("/sitio/{nombrePagina}")
 */
public function paginaAction()
{
    return $this->render('sitio/'.$nombrePagina.'.html.twig');
}

// Después
/**
 * @Route("/sitio/{nombrePagina}")
 */
public function paginaAction($nombrePagina)
```

```
{
    return $this->render('sitio/'.$nombrePagina.'.html.twig');
}
```

Con todo esto, el código completo del controlador `DefaultController` ahora es el siguiente:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/sitio/{nombrePagina}")
     */
    public function paginaAction($nombrePagina)
    {
        return $this->render('sitio/'.$nombrePagina.'.html.twig');
    }
}
```

Prueba ahora a acceder a cualquier página del sitio y deberías verlas sin ningún error: `/sitio/ayuda`, `/sitio/privacidad` y `/sitio/sobre_nosotros`.

4.2.4 Configurando el enrutamiento

En las secciones anteriores apenas se han visto las funcionalidades básicas de las rutas de Symfony. En realidad, el sistema de enrutamiento es muy potente y permite realizar acciones muy avanzadas.

4.2.4.1 Nombres de rutas

Se recomienda asignar un nombre único a cada ruta de la aplicación mediante la opción `name`:

```
// Antes
/**
 * @Route("/sitio/{nombrePagina}")
 */

// Después
/**
 * @Route("/sitio/{nombrePagina}", name="pagina")
 */
```

En este ejemplo sencillo, asignar un nombre a la ruta es totalmente innecesario. Sin embargo, en las aplicaciones web reales es imprescindible y a partir de ahora, todas las rutas creadas en este libro tendrán un nombre. El motivo es que el sistema de enrutamiento funciona en las dos direcciones: asocia URL con acciones pero también es capaz de generar URL a partir del nombre de una ruta.

De esta manera, puedes generar enlaces entre las diferentes acciones de la aplicación, tal y como se explicará en los próximos capítulos. Como el nombre de las rutas se utiliza una y otra vez en las plantillas, es aconsejable que sean cortos aunque fáciles de entender (ej. `pagina`).

4.2.4.2 Valores por defecto

Las variables definidas en las rutas pueden tener valores por defecto, de manera que no sea obligatorio incluirlos en las rutas. Por ejemplo, si quieres que al acceder a la URL `/sitio` se muestre la página de ayuda, puedes hacer que la variable `{nombrePagina}` de la ruta tenga como valor por defecto `ayuda`.

Para ello, define la opción `defaults` e indica como pares `"variable"="valor"` el valor por defecto de tantas variables como quieras:

```
// Antes
/**
 * @Route("/sitio/{nombrePagina}", name="pagina")
 */

// Después
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     "defaults"={ "nombrePagina" = "ayuda" }
 *     name="pagina"
 * )
 */
```

Alternativamente, puedes definir los valores por defecto directamente en los argumentos de las acciones:

```
// Antes
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     "defaults"={ "nombrePagina" = "ayuda" }
 *     name="pagina"
 * )
 */
public function paginaAction($nombrePagina)
{
    // ...
}

// Después
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     name="pagina"
```

```
     * )
     */
    public function paginaAction($nombrePagina = 'ayuda')
    {
        // ...
    }
```

4.2.4.3 Restricciones

Uno de los aspectos a mejorar de la acción `paginaAction()` desarrollada anteriormente consiste en controlar mejor el nombre de la página solicitado. Ahora mismo el usuario podría acceder a cualquier URL de tipo `/sitio/esto-no- existe` y la aplicación trataría de buscar la plantilla `esto-no-existe`, lo que provocaría un error en la aplicación.

Aunque este comportamiento no va a suponer ningún problema de seguridad, puede ser una buena idea restringir los posibles valores que la variable `nombrePagina` puede tomar. Para ello, utiliza la opción `requirements` de la ruta:

```
// Antes
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     "defaults"={ "nombrePagina" = "ayuda" }
 *     name="pagina"
 * )
 */

// Después
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     "defaults"={ "nombrePagina" = "ayuda" },
 *     "requirements"={ "nombrePagina"="ayuda|privacidad|sobre_nosotros" },
 *     name="pagina"
 * )
 */
```

El valor de `requirements` es un *hash* de pares `"variable"="expresion regular"` con los requisitos que debe cumplir cada variable. En este caso, la expresión regular simplemente define los tres valores posibles. Pero puedes utilizar cualquier expresión regular PHP válida (ej. `"requirements" = { "numeroPagina"="\d+" }`).

4.2.5 Definiendo el enrutamiento en otros formatos

Como se explicó anteriormente, las anotaciones son el formato recomendado por Symfony para definir las rutas porque son muy concisas y muy cómodas para trabajar con ellas en el día a día. Sin embargo, Symfony nunca te obliga a usar un formato de configuración determinado.

Así que las rutas de la aplicación también se pueden definir en archivos de configuración independientes y creados con YAML, XML o PHP. A continuación se muestra cómo definir en YAML la siguiente ruta:

```
/**
 * @Route(
 *     "/sitio/{nombrePagina}",
 *     "requirements"={ "nombrePagina"="ayuda|privacidad|sobre_nosotros" },
 *     name="pagina"
 * )
 */
public function paginaAction($nombrePagina)
{
    // ...
}
```

Las rutas en formato YAML se pueden definir dentro del directorio `Resources/config/` de cada *bundle*, pero también en el archivo global de enrutamiento que se define en `app/config/routing.yml`. Así que para definir la ruta `pagina` en formato YAML, habría que editar este último archivo y añadir lo siguiente:

```
# app/config/routing.yml
pagina:
    path: /sitio/{nombrePagina}
    defaults:
        _controller: 'AppBundle:Default:pagina'
    requirements:
        nombrePagina: ayuda|privacidad|sobre_nosotros
```

La clave YAML de la configuración de la ruta (`pagina` en este caso) se considera el nombre de la ruta. La opción `path` define la URL asociada a esta ruta y la opción `requirements` define las condiciones que deben cumplir los valores de las variables definidas en la ruta. Tanto `path` como `requirements` utilizan el mismo formato explicado para las anotaciones.

La gran diferencia entre YAML y anotaciones se encuentra en la opción `defaults`. En las anotaciones, cada `@Route()` se define encima de su método, por lo que Symfony ya sabe qué ruta está asociada a cada acción. Sin embargo, este archivo YAML es totalmente independiente al controlador `DefaultController`, por lo que de alguna manera hay que indicar a Symfony qué acción está asociada a esta ruta.

Para ello, se utiliza la opción `defaults` para asignar un valor por defecto a una variable especial que utiliza Symfony para guardar el controlador + acción asociado a la ruta. Como se explicará en los próximos capítulos, las variables especiales de enrutamiento comienzan por un guión bajo, por lo que esta variable se llama `_controller`.

El valor de `_controller` sigue una nomenclatura especial, llamada "notación *bundle*" que consta de tres partes:

- La primera parte (`AppBundle`), es el nombre del *bundle* en el que se encuentra el controlador. En este ejemplo hace referencia al directorio `src/AppBundle/`.
- La segunda parte (`Default`), es el nombre de la clase del controlador pero sin el sufijo `Controller`. En este ejemplo el archivo es `src/AppBundle/Controller/DefaultController.php`.
- La tercera parte (`ayuda`), es el nombre de la acción pero sin el sufijo `action`. Así, en este ejemplo el método es `ayudaAction()`.

La notación `bundle:controlador:acción` se utiliza en otras partes de Symfony, así que aunque no utilices rutas YAML, es conveniente que aprendas cómo funciona.

4.3 Configurando los permisos

La principal causa de errores al ejecutar la aplicación o los comandos de Symfony es la mala configuración de los permisos de los directorios `app/cache/` y `app/logs/`, que son los únicos en los que escribe Symfony.

El problema es que los archivos de esos directorios los crean tanto el servidor web como los comandos de consola (por ejemplo al borrar la caché). Como el usuario con el que se ejecuta el servidor web suele ser diferente al usuario con el que se ejecutan los comandos, se producen problemas por ejemplo cuando un comando de consola quiere escribir en el archivo de log creado por el servidor web.

A continuación se indican las posibles soluciones más comunes, pero también puedes consultar la guía publicada en el sitio symfony.es sobre cómo solucionar el problema de los permisos de Symfony (http://symfony.es/documentacion/como-solucionar-el-problema-de-los-permisos-de-Symfony/) .

4.3.1 Configurando los permisos en Windows

Accede a las propiedades de los directorios `app/cache/` y `app/logs/` y en la pestaña *Seguridad* cambia los permisos para todos los usuarios. Si esta solución no te funciona, utiliza la **solución 4** que se explica en la siguiente sección.

4.3.2 Configurando los permisos en Linux o macOS

Solución 1. Usar el mismo usuario para el servidor y la consola

Abre el archivo de configuración de tu servidor web (por ejemplo `httpd.conf`) y modifica el valor de las directivas `User` y `Group` para que coincidan con las del usuario con el que ejecutas los comandos en la consola. Después de reiniciar el servidor, ya no tendrás que preocuparte nunca jamás por los permisos de los proyectos Symfony.

Si aplicas esta solución en el servidor de producción, asegúrate de que el usuario tenga sus privilegios restringidos, de manera que se minimicen los problemas de seguridad en caso de que el servidor se vea comprometido.

Solución 2. Definir una ACL con chmod +a (sólo macOS)

En macOS el comando `chmod` soporta la opción `+a` para definir una ACL sobre el directorio. Para aplicar esta solución, ejecuta los siguientes comandos, que incluso detectan automáticamente el usuario con el que se ejecuta el servidor web:

```
$ rm -rf app/cache/*
$ rm -rf app/logs/*

$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]g
inx' | grep -v root | head -1 | cut -d\  -f1`
$ sudo chmod -R +a "$HTTPDUSER allow delete,write,append,file_inherit,director
y_inherit" var
$ sudo chmod -R +a "`whoami` allow delete,write,append,file_inherit,directory_in
herit" var
```

Solución 3. Definir una ACL con setfacl (sólo Linux/BSD)

Si tu distribución de Linux no soporta `chmod +a`, en su lugar seguramente puedes hacer uso del comando `setfacl` de la siguiente manera:

```
$ rm -rf app/cache/*
$ rm -rf app/logs/*

$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]g
inx' | grep -v root | head -1 | cut -d\  -f1`
# si estos comandos no funcionan, añade la opción `-n`
$ sudo setfacl -R -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX var
$ sudo setfacl -dR -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX var
```

Solución 4. Usar umask() al crear los archivos

Si no te funciona ninguna de las soluciones anteriores, puedes usar la función `umask()` de PHP para cambiar los permisos con los que se crean los archivos de cache y de log. Para ello, añade la siguiente línea al principio del todo de los archivos `bin/console`, `web/app.php` y `web/app_dev.php`:

```
// Esto hace que los permisos de los archivos sean 0775
umask(0002);

// Si no te funciona lo anterior, prueba con lo siguiente para
// hacer que los permisos de los archivos sean 0777
umask(0000);
```

Al margen de la solución que utilices, no olvides comprobar que los cambios han surtido efecto. Para ello, prueba a borrar la caché de la aplicación mediante los siguientes comandos:

```
$ php app/console cache:clear --env=dev
$ php app/console cache:clear --env=prod
```

Si se produce algún error, borra manualmente todos los contenidos del archivo `app/cache/` y vuelve a ejecutar el comando. Mientras no llegues a los capítulos más avanzados del libro, cuando se produzca algún error extraño en Symfony, lo primero que debes hacer es probar a borrar la caché con el comando `cache:clear`.

4.4 Configurando *la barra del final* en las URL

El problema de *la barra del final* (*"trailing slash"*) en las URL de los sitios y aplicaciones web es uno de esos problemas que resisten admirablemente el paso del tiempo. En las secciones anteriores, el patrón de la URL de las páginas se configuró como `/sitio/{pagina}`. Esto significa que por ejemplo la URL `/sitio/ayuda` funciona bien, pero la URL `/sitio/ayuda/` produce un error de tipo 404 con el siguiente mensaje: *"No route found for GET /sitio/ayuda/"*.

Para evitar este problema, la solución más sencilla consiste en añadir una barra al final de todas las rutas:

```
// Antes
/**
 * @Route("/sitio/{nombrePagina}")
 */
public function paginaAction($nombrePagina)
{
    // ...
}

// Después
/**
 * @Route("/sitio/{nombrePagina}/")
 */
public function paginaAction($nombrePagina)
{
    // ...
}
```

Con esta nueva configuración, cuando el usuario accede a `/sitio/ayuda/`, se ejecuta directamente la acción `paginaAction()`. Mientras que cuando el usuario accede a `/sitio/ayuda` (sin la barra del final) Symfony redirige automáticamente a la URL `/sitio/ayuda/`, por lo que también se ejecuta la acción `paginaAction()`. Esta redirección se realiza con una respuesta de tipo `301` (*Moved Permanently*).

CAPÍTULO 5

La base de datos

Las aplicaciones Symfony no gestionan su información accediendo directamente a la base de datos. Crean, modifican y borran información mediante objetos PHP, en vez de crear y ejecutar sentencias SQL. Esto es posible gracias a unas librerías externas llamadas ORM u *Object-Relational Mapping*.

Symfony ha elegido el proyecto Doctrine (http://www.doctrine-project.org/) como su ORM oficial. Por tanto, toda la documentación oficial de Symfony utiliza Doctrine y también se han publicado varios *bundles* para mejorar la integración de Symfony con Doctrine.

Las siguientes secciones de este capítulo explican Doctrine desde el punto de vista de los programadores Symfony. Para desarrollar aplicaciones muy complejas, será necesario que profundices en el estudio de Doctrine a través de su documentación oficial (http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/index.html) .

Aunque la aplicación desarrollada en este libro utiliza una base de datos relacional, Doctrine también soporta MongoDB y CouchDB. A pesar de que su funcionamiento es muy diferente, la mayoría de ejemplos y conceptos que se explican a continuación se pueden adaptar fácilmente.

5.1 Entidades

Se denomina *entidades* a los objetos PHP utilizados para manipular la información de la base de datos. Generalmente cada tabla de la base de datos se representa mediante una entidad. No obstante, en ocasiones Doctrine crea tablas adicionales para representar la relación entre dos entidades.

Las entidades se definen mediante clases PHP normales y corrientes. A diferencia de otros ORM, Doctrine no obliga a que estas clases hereden de otra clase especial y tampoco impone ninguna restricción en cómo debes nombrar a estas clases. Por eso estas clases se pueden crear a mano, aunque Symfony incluye comandos para generarlas automáticamente.

Crear varias clases PHP para manipular la información parece más trabajo que escribir simplemente las sentencias SQL necesarias. En la práctica, comprobarás que manipular la información mediante objetos es mucho más productivo. Además, tus aplicaciones Symfony funcionarán con cualquier base de datos (Oracle, MySQL, PostgreSQL, SQL Server, SQLite, etc.) sin tener que hacer cambios en el código.

La aplicación *Cupon* guarda su información en la base de datos mediante cinco tablas, así que se van a crear cinco entidades para representarlas. Por convención, las entidades se crean en el directorio `Entity/` del *bundle* (que debes crear a mano porque inicialmente no existe).

5.1.1 Creando la entidad Ciudad

La entidad `Ciudad` es la más sencilla de todas, ya que la tabla a la que representa solamente contiene tres columnas de información: `id` (clave primaria cuyo valor se autoincrementa), `nombre` (el nombre de la ciudad) y `slug` (el nombre de la ciudad preparado para incluirlo en las URL). Como es tan sencilla, esta entidad se puede crear completamente a mano.

Primero crea una clase llamada `Ciudad` en el directorio `Entity/` del *bundle* `AppBundle`:

```
<?php
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

class Ciudad
{
}
```

A continuación, añade tres propiedades en la clase para representar a las tres columnas de la tabla:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

class Ciudad
{
    protected $id;
    protected $nombre;
    protected $slug;
}
```

NOTA Se recomienda que las propiedades de las entidades de Doctrine sean `protected` o `private` para controlar más fácilmente cómo se modifican sus valores. Además, en las versiones anteriores de Doctrine, las propiedades no podían ser `public` porque entonces no funcionaba el mecanismo de *lazy loading* que se explica más adelante en este capítulo.

Por el momento, `Ciudad.php` simplemente es una clase PHP normal. Para *transformarla* en una entidad, sólo tienes que añadir la anotación `@ORM\Entity`:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
{
    // ...
}
```

Las anotaciones de Doctrine se crean con la clase `Doctrine\ORM\Mapping`. Para que el código de las entidades sea muy conciso, importa esta clase con la instrucción `use Doctrine\ORM\Mapping as ORM;` Así podrás añadir fácilmente las anotaciones de Doctrine con el prefijo `@ORM\`.

La anotación `@ORM\Entity` hace que Doctrine guarde los datos de este objeto en una tabla llamada igual que la clase PHP (`Ciudad` en este caso). Puedes cambiar el nombre de la tabla añadiendo la anotación `@ORM\Table` y estableciendo el nombre en su atributo `name`:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="ProyectoCupon_Ciudad")
 */
class Ciudad
{
    // ...
}
```

A continuación, añade más anotaciones para que Doctrine sepa qué tipo de información almacena cada propiedad. Así podrá crear el tipo de columna adecuado en la tabla de la base de datos:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
{
  /** @ORM\Column(type="integer") */
  protected $id;

  /** @ORM\Column(type="string", length=100) */
  protected $nombre;

  /** @ORM\Column(type="string", length=100) */
  protected $slug;
}
```

Las características de cada propiedad se definen con la anotación `@ORM\Column()`. Por defecto Doctrine supone que las propiedades son cadenas de texto de una longitud de 255 caracteres. Si esto no es lo más adecuado para tu aplicación, cambia el tipo mediante el atributo `type`.

Los tipos de datos disponibles en las entidades no son ni los de SQL ni los de PHP. Doctrine define sus propios tipos de datos, que después convierte al tipo adecuado en función de la base de datos utilizada. La siguiente tabla muestra todos los tipos de datos disponibles:

Tipo de Doctrine	Tipo equivalente SQL	Tipo equivalente PHP
`smallint`	`SMALLINT`	`integer`
`integer`	`INT`	`integer`
`bigint`	`BIGINT`	`string`
`float`	`FLOAT`	`double`
`decimal`	`DECIMAL`	`double`
`date`	`DATETIME`	`DateTime`
`time`	`TIME`	`DateTime`
`datetime`	`DATETIME` o `TIMESTAMP`	`DateTime`
`datetimez`	`DATETIME`	`DateTime` (con el huso horario)
`boolean`	`BOOLEAN`	`boolean`
`string`	`VARCHAR`	`string`
`guid`	`VARCHAR`	`string`
`text`	`CLOB`	`string`
`object`	`CLOB`	`Object`
`array`	`CLOB`	`Array`
`simple_array`	`TEXT`	`Array`
`json_array`	`TEXT`	`Array`
`binary`	`BINARY`	-
`blob`	`BLOB`	-

Doctrine define los siguientes atributos para cada propiedad de la entidad:

- `type`: indica el tipo de dato de la columna creada en la tabla en la base de datos. Su valor por defecto es `string`.
- `name`: establece el nombre de la columna en la tabla. Su valor por defecto coincide con el nombre de la propiedad.
- `length`: la longitud máxima que puede tener el contenido de la propiedad. Sólo está definido para las propiedades de tipo texto y su valor por defecto es `255`.
- `unique`: indica si el valor de la columna debe ser único en toda la tabla. Su valor por defecto es `false`.

- `nullable`: indica si esta columna permite guardar `null` como valor. Su valor por defecto es `false`.

Además, para las propiedades de tipo `decimal` puedes definir los siguientes atributos:

- `precision`: es la precisión o número de dígitos significativos que se almacenan. En otras palabras, cuántos dígitos (contando los decimales) se pueden guardar para el número. Su valor por defecto es `0`.
- `scale`: es la escala o número de dígitos que se pueden almacenar después del símbolo decimal. En otras palabras, cuántos decimales se pueden guardar para el número. Su valor por defecto es `0`.

Todas las entidades deben definir una clave primaria, sea esta simple o compuesta. En la entidad `Ciudad` la clave primaria es la propiedad `id`. Márcala como clave primaria añadiendo la anotación `@ORM\Id`:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    protected $id;

    // ...
}
```

El código anterior le indica a Doctrine que la clave primaria es `$id`, pero no establece su valor automáticamente. A menos que quieras asignar manualmente el valor de las claves primarias, añade también la anotación `@ORM\GeneratedValue`:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
```

```
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;

    // ...
}
```

Sin ningún otro atributo, la anotación `@ORM\GeneratedValue` genera las claves primarias utilizando la estrategia `AUTO_INCREMENT` en MySQL y SQLite, `IDENTITY` en MSSQL y `SEQUENCE` en Oracle y PostgreSQL. Puedes modificar la estrategia, estableciendo el atributo `strategy` (`@ORM\GeneratedValue(strategy="...")`) y utilizando los valores `AUTO`, `SEQUENCE`, `IDENTITY`, `TABLE` o `NONE` tal y como se explica en el manual de Doctrine.

Como las propiedades de la entidad son `protected`, para que Doctrine pueda manipular su valor es necesario añadir los métodos *getter* y *setter* correspondientes:

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
{
    // ...

    public function getId()
    {
        return $this->id;
    }

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

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

    public function setSlug($slug)
    {
```

```
        $this->slug = $slug;
    }

    public function getSlug()
    {
        return $this->slug;
    }
}
```

Doctrine se encarga de establecer el valor de la clave primaria de la entidad, por lo que no es necesario que añadas el método `setId()`. Los métodos `getXXX()` y `setXXX()` de este ejemplo son muy básicos, pero como las entidades son clases PHP normales, puedes añadir en cada uno tanta lógica como sea necesaria.

> **NOTA** Se recomienda utilizar la notación *camelCase* para el nombre de las propiedades (ej. `fechaNacimiento` en vez de `fecha_nacimiento`) porque así los *getters* y *setters* se simplifican (ej. `getFechaNacimiento()`) y no tendrás que pensar cuál es su formato (¿`getfecha_nacimiento()`?, ¿`getFecha_nacimiento()`?, etc.)

Por último, para completar la entidad puedes añadir también el método mágico `__toString()` para decirle a PHP cómo convertir esta entidad en una cadena de texto. Esto es especialmente útil cuando trabajas con entidades relacionadas, cuyos valores se muestran por ejemplo en una lista desplegable (como sucederá con la entidad `Ciudad` más adelante):

```
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Ciudad
{
    // ...

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

5.1.2 Creando la entidad Tienda

En primer lugar, crea una clase vacía llamada `Tienda` en el directorio `Entity/`:

```
<?php
// src/AppBundle/Entity/Tienda.php
namespace AppBundle\Entity;
```

```
class Tienda
{
}
```

Después, añade las propiedades necesarias para almacenar la información en la base de datos:

```
// src/AppBundle/Entity/Tienda.php
namespace AppBundle\Entity;

class Tienda
{
    protected $id;
    protected $nombre;
    protected $slug;
    protected $login;
    protected $password;
    protected $descripcion;
    protected $direccion;
    protected $ciudad;
}
```

Para convertir la clase PHP en una entidad, añade las anotaciones correspondientes en la clase y en las propiedades:

```
// src/AppBundle/Entity/Tienda.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/** @ORM\Entity */
class Tienda
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    protected $id;

    /** @ORM\Column(type="string", length=100) */
    protected $nombre;

    /** @ORM\Column(type="string", length=100) */
    protected $slug;

    /** @ORM\Column(type="string", length=10) */
    protected $login;
```

```
    /** @ORM\Column(type="string") */
    protected $password;

    /** @ORM\Column(type="text") */
    protected $descripcion;

    /** @ORM\Column(type="text") */
    protected $direccion;

    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad") */
    protected $ciudad;
}
```

Todas las propiedades salvo `$ciudad` son muy sencillas y se pueden entender siguiendo la explicación de la entidad `Ciudad` anterior. La propiedad `$ciudad` es muy diferente porque se trata de una **clave externa** que relaciona una tabla con otra, es decir, asocia la entidad `Tienda` con la entidad `Ciudad`.

La relación entre ciudades y tiendas se definió como `1-n` en el capítulo dos. Desde el punto de vista de la tienda la relación es `n-1` (muchas tiendas pueden estar asociadas con una ciudad, pero una tienda sólo puede estar asociada con una ciudad). Este tipo de relación se denomina *many-to-one* o *muchos-a-uno* y se define con la anotación `@ORM\ManyToOne`:

```
/** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad") */
protected $ciudad;
```

El único atributo obligatorio es `targetEntity` que indica el *namespace* completo de la entidad con la que se establece la relación. Aunque en la entidad `Tienda` la propiedad se llama `$ciudad`, en la tabla `Tienda` de la base de datos Doctrine crea una columna llamada `ciudad_id` relacionada con la columna `id` de la tabla `Ciudad`. La conversión entre `ciudad` y `ciudad_id` es transparente para el programador, ya que en tu código siempre utilizarás el valor `$ciudad` y será Doctrine el que lo convierta a `ciudad_id` para guardar o buscar información en la base de datos.

Considerar que la columna se llama `<propiedad>_id` y está relacionada con la columna `id` de otra tabla es el comportamiento por defecto de Doctrine, pero puedes cambiarlo con la anotación `@ORM\JoinColumn`. El siguiente código muestra dos ejemplos equivalentes:

```
/** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad") */
protected $ciudad;

/**
 * @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad")
 * @ORM\JoinColumn(name="ciudad_id", referencedColumnName="id")
 */
protected $ciudad;
```

Doctrine soporta las relaciones uno-a-uno, muchos-a-uno, uno-a-muchos, muchos-a-muchos, relaciones autoreferentes, etc. El manual oficial de Doctrine explica los detalles de cada relación (http://www.doctrine-project.org/docs/orm/current/en/reference/association-mapping.html) .

Como se sabe, el último paso consiste en añadir los *getters* y *setters* para todas las propiedades. Como esta tarea pronto se convierte en algo tedioso, Symfony incluye un comando para añadir los *getters* y *setters* básicos automáticamente. Para ello, ejecuta el siguiente comando:

```
$ php app/console generate:doctrine:entities AppBundle
```

Si ahora vuelves al archivo `src/AppBundle/Entity/Tienda.php`, verás que su contenido ha sido modificado para incluir todos los *getters* y *setters*:

```php
// src/AppBundle/Entity/Tienda.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/** @ORM\Entity */
class Tienda
{
    // ... propiedades

    public function getId()
    {
        return $this->id;
    }

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

        return $this;
    }

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

    // ...

    public function setCiudad(\AppBundle\Entity\Ciudad $ciudad)
    {
        $this->ciudad = $ciudad;

        return $this;
    }
```

```
    public function getCiudad()
    {
        return $this->ciudad;
    }
}
```

NOTA Los *setters* generados por Doctrine incluyen al final la instrucción `return $this;` para permitir el uso de *"interfaces fluídas"*, como por ejemplo: `$tienda->setNombre(...)->setCiudad(...);`.

Observa con atención el *setter* de la propiedad `$ciudad`. En la base de datos esta propiedad se guarda como `ciudad_id` y por tanto, sólo se guarda el valor del `id` de la ciudad. Sin embargo, al trabajar con los objetos de las entidades, tendrás que pasar el objeto `Ciudad` entero, no el valor de su propiedad `id` (como se explicará más adelante). Recuerda que al utilizar un ORM tu código siempre trabaja con objetos PHP. Las conversiones para gestionar la información de la base de datos es algo que realiza Doctrine internamente.

Aunque el comando `generate:doctrine:entities` ha sido muy probado y es fiable, cualquier comando que modifique el contenido de un archivo es susceptible de cometer errores. Así que al ejecutar el comando se genera automáticamente una copia de seguridad de la entidad original en el archivo `Tienda.php~`. Después de comprobar que los cambios se han realizado correctamente, deberías borrar esta copia de seguridad, ya que tiene un nombre que se confunde fácilmente con el de la entidad original. También puedes evitar la creación de esta copia de seguridad añadiendo la opción `--no-backup` al ejecutar el comando.

Para completar la entidad `Tienda`, no olvides añadir el método mágico `__toString()`:

```
// src/AppBundle/Entity/Tienda.php

/** @ORM\Entity */
class Tienda
{
    // ...

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

5.1.3 Creando la entidad Oferta

Crear las entidades es algo que sólo se hace una vez en cada proyecto, pero es una tarea tediosa y propensa a cometer errores. Por eso Symfony también incluye el comando `doctrine:generate:entity` para generar automáticamente entidades completas. Este comando es interactivo, por lo que se basa en plantearte preguntas y esperar tus respuestas. Cuando una

pregunta muestra una respuesta por defecto entre corchetes, puedes pulsar la tecla `Enter` para seleccionarla:

```
$ php app/console doctrine:generate:entity

  Welcome to the Doctrine entity generator

  This command helps you generate Doctrine entities.

  First, you need to give the entity name you want to generate.
  You must use the shortcut notation like AcmeBlogBundle:Post.

  The Entity shortcut name: AppBundle:Oferta
```

El *Entity shortcut name* es el nombre corto de la entidad, que se forma concatenando el nombre del *bundle* en el que se encuentra y el nombre de la clase de la entidad. Para generar una entidad `Oferta.php` en el *bundle* `AppBundle`, indica el nombre corto `AppBundle:Oferta`.

```
Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation) [annotation]: <Enter>
```

Las anotaciones son el formato recomendado para definir las entidades, pero Doctrine también permite definir esta información en archivos XML, YAML y PHP. Para seleccionar las anotaciones como formato, simplemente pulsa `Enter`. Después de indicar esta información básica, el comando solicita una por una la información de todas las propiedades.

No es necesario que crees la propiedad `$id` que actúa de clave primaria de la entidad porque el comando la genera automáticamente. Así que empieza por la primera propiedad de `Oferta`, que es `$nombre`:

```
New field name (press <return> to stop adding fields): nombre
Field type [string]: <Enter>
Field length [255]: <Enter>
```

En primer lugar indica el nombre de la propiedad (en este caso, `nombre`). Después, indica el tipo de información que se guarda en la propiedad. Como por defecto el tipo es `string`, puedes pulsar `Enter` para seleccionarlo. Dependiendo del tipo de dato seleccionado, el comando puede plantear más preguntas. Para el tipo `string` por ejemplo se pide la longitud máxima de la cadena de texto. Para seleccionar el valor por defecto `255`, pulsa `Enter`.

A continuación se crean el resto de propiedades de la entidad:

```
New field name (press <return> to stop adding fields): slug
Field type [string]: <Enter>
Field length [255]: <Enter>
```

```
New field name (press <return> to stop adding fields): descripcion
Field type [string]: text

New field name (press <return> to stop adding fields): condiciones
Field type [string]: text

New field name (press <return> to stop adding fields): rutaFoto
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): precio
Field type [string]: decimal

New field name (press <return> to stop adding fields): descuento
Field type [string]: decimal

New field name (press <return> to stop adding fields): fechaPublicacion
Field type [string]: datetime

New field name (press <return> to stop adding fields): fechaExpiracion
Field type [string]: datetime

New field name (press <return> to stop adding fields): compras
Field type [string]: integer

New field name (press <return> to stop adding fields): umbral
Field type [string]: integer

New field name (press <return> to stop adding fields): revisada
Field type [string]: boolean

New field name (press <return> to stop adding fields): ciudad
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): tienda
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields):
```

El comando `doctrine:generate:entity` no permite indicar el nombre de una entidad como tipo de dato (como los que necesitan las propiedades `$ciudad` y `$tienda`). En este caso, simplemente utiliza el tipo por defecto (`string`) y corrige después a mano el código generado.

Para terminar de añadir propiedades, pulsa `Enter` como nombre de la propiedad. A continuación el comando pregunta si quieres generar un repositorio vacío. Por el momento, y hasta que más adelante no se explique la utilidad de los repositorios, contesta que `no`:

```
Do you want to generate an empty repository class [no]? no
```

El comando ya dispone de toda la información necesaria, por lo que se muestra el resumen de las próximas acciones a realizar y se pide la confirmación de que realmente se quiere generar la entidad:

```
Summary before generation

You are going to generate a "AppBundle:Oferta" Doctrine entity
using the "annotation" format.

Do you confirm generation [yes]? yes
```

La entidad ya ha sido generada en el archivo `src/AppBundle/Entity/Oferta.php`. Si el directorio `Entity/` no existía, el comando también lo genera. Antes de dar por concluida la entidad, corrige las propiedades `$ciudad` y `$tienda`:

```
// src/AppBundle/Entity/Oferta.php

// Antes
class Oferta
{
    // ...

    /** @ORM\Column(name="ciudad", type="string", length=255) */
    private $ciudad;

    /** @ORM\Column(name="tienda", type="string", length=255) */
    private $tienda;
}

// Ahora
class Oferta
{
    // ...

    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad") */
    private $ciudad;

    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Tienda") */
    private $tienda;
}
```

Actualiza también los *getters* y *setters* generados:

```
// src/AppBundle/Entity/Oferta.php

class Oferta
{
    // ...

    public function setCiudad(\AppBundle\Entity\Ciudad $ciudad)
    {
        $this->ciudad = $ciudad;
    }

    public function getCiudad()
    {
        return $this->ciudad;
    }

    public function setTienda(\AppBundle\Entity\Tienda $tienda)
    {
        $this->tienda = $tienda;
    }

    public function getTienda()
    {
        return $this->tienda;
    }
}
```

Y como siempre, no olvides añadir el método mágico `__toString()`:

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

5.1.4 Creando la entidad Usuario

La entidad `Usuario` tiene muchas propiedades, así que se utiliza el comando `doctrine:generate:entity` para generarla automáticamente:

```
$ php app/console doctrine:generate:entity

# ...

The Entity shortcut name: AppBundle:Usuario
Configuration format (yml, xml, php, or annotation) [annotation]: <Enter>

New field name (press <return> to stop adding fields): nombre
```

```
Field type [string]: <Enter>
Field length [255]: 100

New field name (press <return> to stop adding fields): apellidos
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): email
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): password
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): direccion
Field type [string]: text

New field name (press <return> to stop adding fields): permiteEmail
Field type [string]: boolean

New field name (press <return> to stop adding fields): fechaAlta
Field type [string]: datetime

New field name (press <return> to stop adding fields): fechaNacimiento
Field type [string]: datetime

New field name (press <return> to stop adding fields): dni
Field type [string]: <Enter>
Field length [255]: 9

New field name (press <return> to stop adding fields): numeroTarjeta
Field type [string]: <Enter>
Field length [255]: 20

New field name (press <return> to stop adding fields): ciudad
Field type [string]: <Enter>
Field length [255]: <Enter>

New field name (press <return> to stop adding fields): <Enter>

Do you want to generate an empty repository class [no]? no
Do you confirm generation [yes]? yes
```

Si el comando se ejecuta correctamente, se habrá creado la entidad en el archivo `src/AppBundle/Entity/Usuario.php`. Para finalizar la entidad, corrige la propiedad `$ciudad` y su *setter*:

```
// src/AppBundle/Entity/Usuario.php

// Antes
class Usuario
{
    // ...

    /** @ORM\Column(name="ciudad", type="string", length=255) */
    private $ciudad;

    public function setCiudad($ciudad)
    {
        $this->ciudad = $ciudad;
    }
}

// Ahora
use AppBundle\Entity\Ciudad;

class Usuario
{
    // ...

    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Ciudad") */
    private $ciudad;

    public function setCiudad(Ciudad $ciudad)
    {
        $this->ciudad = $ciudad;
    }
}
```

Y como siempre, no olvides añadir el método mágico `__toString()`:

```
public function __toString()
{
    return $this->getNombre().' '.$this->getApellidos();
}
```

Una última mejora que puede resultar muy útil es establecer automáticamente la fecha de alta del usuario. Como las entidades son simples clases PHP, esto es tan fácil como añadir un constructor a la clase para establecer el valor de la propiedad `$fechaAlta`:

```
// src/AppBundle/Entity/Usuario.php

class Usuario
{
    // ...
```

```
    public function __construct()
    {
        $this->fechaAlta = new \DateTime();
    }
}
```

5.1.5 Creando la entidad Venta

La última entidad de la aplicación es muy especial y por eso se crea manualmente. En realidad, la entidad `Venta` representa a una tabla intermedia entre las ofertas y los usuarios. Si no se guardara la fecha de cada venta, no sería necesario definir esta entidad, ya que Doctrine crearía automáticamente la tabla que relaciona la entidad `Oferta` con la entidad `Usuario`. Esta es una de las ventajas de utilizar relaciones *muchos-a-muchos* de Doctrine.

Crea un archivo llamado `Venta.php` en el directorio `Entity/` y añade el siguiente código:

```
<?php
// src/AppBundle/Entity/Venta.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/** @ORM\Entity */
class Venta
{
    /** @ORM\Column(type="datetime") */
    protected $fecha;

    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="OfertaBundle\Entity\Oferta")
     */
    protected $oferta;

    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="AppBundle\Entity\Usuario")
     */
    protected $usuario;

    public function setFecha($fecha)
    {
        $this->fecha = $fecha;
    }

    public function getFecha()
    {
        return $this->fecha;
    }
```

```
    public function setOferta(\AppBundle\Entity\Oferta $oferta)
    {
        $this->oferta = $oferta;
    }

    public function getOferta()
    {
        return $this->oferta;
    }

    public function setUsuario(\AppBundle\Entity\Usuario $usuario)
    {
        $this->usuario = $usuario;
    }

    public function getUsuario()
    {
        return $this->usuario;
    }
}
```

Observa cómo en este caso la entidad no tiene la típica clave primaria llamada `id`, sino que se define una clave primaria compuesta formada por la oferta y el usuario. Para crear claves compuestas, añade la anotación `@ORM\Id` en todas las propiedades que forman la clave.

Además de las alternativas explicadas en las secciones anteriores, existe una última forma de crear entidades. Este método sólo es útil cuando la entidad es muy sencilla, ya que se basa en pasar toda la información de la entidad directamente al comando `doctrine:generate:entity`, como muestra el siguiente ejemplo:

```
$ php app/console doctrine:generate:entity --entity="AppBundle:NombreEntidad"
  --fields="nombre:string(100) apellidos:string(100) email:string(255) fechaAlt
a:datetime"
```

5.2 Creando y configurando la base de datos

5.2.1 Usuarios y permisos

Una buena práctica para el uso de bases de datos en aplicaciones web consiste en crear un usuario específico para cada aplicación. Si tu base de datos es por ejemplo MySQL, puedes crear un usuario con el siguiente comando:

```
$ mysql -u root

mysql> CREATE USER 'usuario'@'localhost' IDENTIFIED BY 'contraseña';
```

Así, para crear un usuario llamado `cupon` y cuya contraseña sea también `cupon` ejecuta el comando: `CREATE USER 'cupon'@'localhost' IDENTIFIED BY 'cupon';`

Después de crear el usuario, otórgale los permisos adecuados sobre la base de datos de la aplicación. No hace falta que crees la base de datos, pero sí que debes decidir su nombre. Si esta base de datos se llama también `cupon`, el comando sería el siguiente:

```
mysql> GRANT ALL ON cupon.* TO 'cupon'@'localhost';
```

5.2.2 Configurando el acceso a base de datos

La única información requerida por Symfony para acceder a la base de datos es el nombre y contraseña del usuario y el nombre de la base de datos. Como estas credenciales cambian de un ordenador a otro, configura sus valores como parámetros en el archivo `app/config/parameters.yml`:

```
parameters:
    database_driver:   pdo_mysql
    database_host:     localhost
    database_port:     ~
    database_name:     cupon
    database_user:     cupon
    database_password: cupon

    # ...
```

Si te fijas en el archivo `app/config/config.yml`, verás las siguientes opciones de configuración bajo la clave `dbal` de `doctrine`:

```
# app/config/config.yml
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        port:     %database_port%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
        charset:  UTF8
```

Las opciones cuyo nombre se encierra entre `%`, toman su valor de las opciones del mismo nombre definidas en el archivo `parameters.yml`.

5.2.3 Creando la base de datos y sus tablas

Una vez configurados los datos de acceso, ya puedes crear la base de datos vacía directamente con el siguiente comando de Symfony:

```
$ php app/console doctrine:database:create

Created database for connection named cupon
```

A continuación, crea toda la estructura de tablas de la base de datos (también llamada *esquema*):

```
$ php app/console doctrine:schema:create
```

Si en vez de crear las tablas, solamente quieres ver las sentencias SQL que se van a ejecutar, añade la opción `--dump-sql`:

```
$ php app/console doctrine:schema:create --dump-sql

CREATE TABLE Ciudad (id INT AUTO_INCREMENT NOT NULL, nombre VARCHAR(100)
NOT NULL, slug VARCHAR(100) NOT NULL, PRIMARY KEY(id)) ENGINE = InnoDB;
CREATE TABLE Usuario ...
```

Durante el desarrollo de la aplicación suele ser habitual añadir o eliminar propiedades de las entidades. En este caso no es necesario que elimines la base de datos y vuelvas a crearla. Basta utilizar el comando `doctrine:schema:update` para actualizar la estructura de tablas a la última definición de las entidades:

```
// Ver las sentencias SQL que se ejecutarían para la actualización
$ php app/console doctrine:schema:update --dump-sql

// Ejecutar las sentencias SQL anteriores
$ php app/console doctrine:schema:update --force
```

5.3 El *Entity Manager*

La manipulación de la información de Doctrine (buscar, crear, modificar y borrar registros en las tablas) se realiza a través de un objeto especial llamado *Entity Manager*. Si has trabajado con herramientas como Hibernate (http://www.hibernate.org/) de Java, la mayoría de conceptos te resultarán familiares.

5.3.1 Obteniendo el *entity manager*

Symfony crea este objeto automáticamente y lo pone a tu disposición a través del *"contenedor de inyección de dependencias"*, un concepto clave del funcionamiento de Symfony que se explica detalladamente en el apéndice B (página 505) (no leas este apéndice todavía porque es demasiado pronto para hacerlo).

Por el momento, sólo debes saber que para obtener el *entity manager* dentro de un controlador, debes utilizar la siguiente instrucción:

```
class DefaultController extends Controller
{
    public function portadaAction($ciudad)
```

```
    {
        // ...

        $em = $this->getDoctrine()->getManager();
    }
}
```

Resulta muy habitual recoger el *entity manager* en una variable llamada `$em`, por lo que ese código lo verás en la mayoría de aplicaciones y tutoriales de Symfony.

NOTA En las versiones anteriores de Doctrine, la instrucción utilizada para obtener el *entity manager* era `$this->getDoctrine()->getEntityManager()`. Este código sigue funcionando, pero ha sido declarado obsoleto por Doctrine y por tanto, dejará de funcionar en las próximas versiones.

5.3.2 Buscando información

Una vez obtenido el *entity manager*, ya puedes buscar información en la base de datos sin utilizar sentencias SQL. El siguiente ejemplo muestra cómo buscar los datos de la oferta cuya propiedad `id` vale `1`:

```
$em = $this->getDoctrine()->getManager();

$oferta = $em->find('AppBundle:Oferta', 1);
// la siguiente instrucción es equivalente:
// $oferta = $em->find('AppBundle\Entity\Oferta', 1);

// $oferta ya contiene toda la información de la oferta
$precio = $oferta->getPrecio();
```

El método `find()` busca entidades mediante su clave primaria. El primer argumento es el nombre de la entidad que se busca (indicado con la *notación bundle*: `<nombre bundle>:<nombre entidad>`) aunque también puedes usar el *namespace* completo de la entidad. El segundo argumento es el valor de la clave primaria del registro que se está buscando.

Una de las principales ventajas del *entity manager* es que puede reducir drásticamente el número de consultas a la base de datos, ya que guarda en memoria los resultados de las consultas que ya han sido realizadas. Por eso, si ahora vuelves a buscar la oferta cuyo `id` es `1`, Doctrine te devuelve instantáneamente el mismo resultado que antes, sin tener que hacer una nueva consulta.

El método `find()` anterior es en realidad un atajo del verdadero método para realizar consultas:

```
$em = $this->getDoctrine()->getManager();
$oferta = $em->getRepository('AppBundle:Oferta')->find(1);
```

Para obtener todas las entidades de un tipo (es decir, todas las filas de una tabla) se emplea el método `findAll()`. Así, la variable `$ofertas` del siguiente ejemplo es un array con todos los objetos de tipo `Oferta` de la aplicación:

```
$em = $this->getDoctrine()->getManager();
$ofertas = $em->getRepository('AppBundle:Oferta')->findAll();
```

Para realizar consultas con condiciones más complejas, se emplean los métodos `findBy()` (para obtener muchos resultados) y `findOneBy()` (para obtener sólo un resultado):

```
$em = $this->getDoctrine()->getManager();

// Encontrar todas las ofertas revisadas
$ofertasRevisadas = $em->getRepository('AppBundle:Oferta')->findBy(array(
    'revisada' => true
));

// Encontrar la ciudad de Vitoria-Gasteiz
$ciudad = $em->getRepository('AppBundle:Ciudad')->findOneBy(array(
    'slug' => 'vitoria-gasteiz'
));

// Encontrar usuarios de Vitoria-Gasteiz que permitan el envío de emails
$usuarios = $em->getRepository('AppBundle:Usuario')->findBy(array(
    'ciudad' => $ciudad->getId(),
    'permiteEmail' => true,
));
```

Las condiciones de los métodos `findBy()` y `findOneBy()` se indican mediante un array asociativo de propiedades y valores. Observa como en el último ejemplo se buscan usuarios a partir de la ciudad a la que pertenecen. Aunque en la base de datos la columna se llama `ciudad_id`, la propiedad de la entidad se llama `$ciudad` y por eso se utiliza `ciudad` en el criterio de búsqueda. En cualquier caso, para realizar la búsqueda sí que hay que utilizar el valor de la propiedad `id` (es decir, `$ciudad->getId()`) y por eso no se pasa la entidad entera (`$ciudad`).

El método `findBy()` de Doctrine admite otros tres argumentos opcionales: el criterio por el que se ordenan los resultados, el máximo número de resultados devueltos y la posición a partir de la que se devuelven resultados. Los dos últimos argumentos están pensados para la paginación de las consultas que devuelven muchos resultados:

```
// devuelve los 10 usuarios que primero se dieron de alta en el sitio web
$losMasVeteranos = $em->getRepository('AppBundle:Usuario')->findBy(
    array(), // criterio de búsqueda: vacío = devuelve todos
    array('fechaAlta' => 'ASC'), // ordenado de más antiguo a más moderno
    10,  // 10 resultados
    0    // empezando en la posición 0
);

// devuelve la cuarta página de resultados de la búsqueda que
// encuentra a los usuarios suscritos al boletín de noticias y
// ordenados primero alfabéticamente y después por edad
$usuarios = $em->getRepository('AppBundle:Usuario')->findBy(
    array('permiteEmail' => true),
```

```
    array('nombre' => 'ASC', 'fechaNacimiento' => 'DESC'),
    10,  // 10 resultados
    30   // empezando en la posición 30
);
```

Imagina que ahora quieres mostrar el nombre y apellidos de todos los usuarios devueltos por el *entity manager*:

```
// $usuarios es la variable del listado de código anterior
$usuarios = ...

// Mostrar el nombre de los usuarios encontrados
foreach ($usuarios as $usuario) {
    echo $usuario->getNombre().' '.$usuario->getApellidos();
}
```

El código anterior es trivial, pero no lo es tanto si en vez del nombre quieres mostrar el nombre de la ciudad en la que reside cada usuario:

```
// $usuarios es la variable del listado de código anterior
$usuarios = ...

// Mostrar el nombre de la ciudad de cada usuario
foreach ($usuarios as $usuario) {
    echo $usuario->getCiudad()->getNombre();
}
```

¿Funciona correctamente el código anterior? Aunque te resulte extraño, la respuesta es afirmativa. En la base de datos, la tabla de los usuarios sólo contiene el atributo `id` de la ciudad. De hecho, la columna se llama `ciudad_id`, aunque en la entidad la propiedad se llama `$ciudad`. Así que, ¿cómo es posible que la instrucción `$usuario->getCiudad()` devuelva el objeto de la entidad `Ciudad` relacionada con el usuario?

La clave para entender cómo funciona el código anterior es la característica *lazy loading* de Doctrine. Este comportamiento hace que Doctrine busque automáticamente en la base de datos cualquier información que solicites y no esté disponible. La instrucción `$usuario->getCiudad()` por ejemplo hace que Doctrine realice una consulta a la base de datos para obtener todos los datos de la ciudad cuyo atributo `id` se guarda en la tabla de los usuarios.

El *lazy loading* es completamente transparente para el programador y parece una idea brillante. Sin embargo, puede penalizar seriamente el rendimiento de la aplicación. El bucle `foreach()` del código anterior podría realizar cientos de consultas a la base de datos, una por cada ciudad cuyos datos no se hayan buscado anteriormente. Más adelante se explica cómo solucionar los problemas causados por el *lazy loading*.

Los métodos *oficiales* para realizar búsquedas con el *entity manager* son `find()`, `findAll()`, `findBy()` y `findOneBy()`. No obstante, Doctrine aprovecha el método mágico `__call()` para permitir atajos de búsqueda para todas las propiedades de la entidad:

```
$em = $this->getDoctrine()->getManager();

// Encontrar todas las ofertas revisadas
$ofertas = $em->getRepository('AppBundle:Oferta')
              ->findBy(array('revisada' => true));

// Misma búsqueda, pero utilizando el atajo
$ofertas = $em->getRepository('AppBundle:Oferta')
              ->findByRevisada(true);


// Encontrar la ciudad de Vitoria-Gasteiz
$ciudad = $em->getRepository('AppBundle:Ciudad')
             ->findOneBy(array('slug' => 'vitoria-gasteiz'));

// Misma búsqueda, pero utilizando el atajo
$ciudad = $em->getRepository('AppBundle:Ciudad')
             ->findOneBySlug('vitoria-gasteiz');
```

Los nombres de los métodos de los atajos siempre empiezan por `findBy` o `findOneBy` y acaban por el nombre de la propiedad utilizando la *notación CamelCase*: `codigo_postal` se transforma en `findByCodigoPostal()`, `fechaPublicacion` se transforma en `findByFechaPublicacion()`, etc.

5.3.3 Creando, modificando o borrando información

Buscar información es la operación más habitual en las aplicaciones web, pero crear, modificar y borrar información son las operaciones más importantes. Todas estas operaciones se realizan también a través del *entity manager*.

El siguiente ejemplo muestra cómo crear una entidad de tipo `Oferta` y guardarla después en la base de datos:

```
use AppBundle\Entity\Oferta;

$oferta = new Oferta();

$oferta->setNombre('Lorem Ipsum ...');
$oferta->setPrecio(10.99);
// ... completar todas las propiedades ...

$em = $this->getDoctrine()->getManager();
$em->persist($oferta);
$em->flush();
```

Para crear una oferta, primero instancia la clase correspondiente a la entidad `Oferta`. Después, utiliza los *setters* de la entidad (o si lo prefieres, su constructor) para rellenar toda la información de la oferta. Para guardarla en la base de datos, obtén el *entity manager* e invoca su método `persist()` pasando como parámetro la entidad que se quiere guardar.

El método `persist()` marca la entidad como *persistente*, pero no la guarda en la base de datos (no se ejecuta la sentencia `INSERT` de SQL). Esta es otra de las claves del funcionamiento de Doctrine. Para reducir el número de sentencias SQL ejecutadas, Doctrine no ejecuta ninguna sentencia de modificación de información hasta que se solicita explícitamente invocando el método `flush()`.

Si en el código anterior no ejecutas la sentencia `flush()`, puedes seguir utilizando el objeto `$oferta`, pero sus datos no se han guardado en la base de datos y por tanto, no aparecerá en los resultados de las búsquedas. Piensa en el método `flush()` como si fuera un comando llamado *Guardar cambios*. Hasta que no lo ejecutas, la base de datos no se modifica.

Modificar los datos de una entidad es muy similar a crearla. La diferencia es que, en vez de partir de una entidad vacía, se parte de una entidad existente obtenida mediante alguna búsqueda:

```
$em = $this->getDoctrine()->getManager();
$oferta = $em->getRepository('AppBundle:Oferta')->find(3);

$oferta->setPrecio($oferta->getPrecio() + 5);
// no es necesario ejecutar $em->persist($oferta); porque la
// entidad ya existía en la base de datos antes de modificarla
$em->flush();
```

Por último, para borrar una entidad se emplea el método `remove()`. El siguiente código muestra cómo buscar a un usuario específico para borrarlo de la aplicación:

```
$em = $this->getDoctrine()->getManager();

$usuario = $em->getRepository('AppBundle:Usuario')->findOneByDni('1234567L');

$em->remove($usuario);
$em->flush();
```

Técnicamente, el funcionamiento de Doctrine descrito anteriormente y la obligación de utilizar el método `flush()` para guardar los cambios, se conoce como patrón *Unit of Work*. Según la definición de Martin Fowler (http://martinfowler.com/eaaCatalog/unitOfWork.html) este patrón consiste en *"mantener una lista de todos los objetos modificados durante una transacción y coordinar la persistencia de esos cambios y la resolución de los posibles problemas de concurrencia"*.

5.4 Archivos de datos o *fixtures*

Las aplicaciones web suelen necesitar algunos datos iniciales para funcionar correctamente. En el caso de la aplicación *Cupon*, es imprescindible por ejemplo que en la base de datos existan varias ciudades. Estos datos iniciales se cargan mediante los *fixtures* o archivos de datos.

Además de los datos imprescindibles, cuando se desarrolla la aplicación son necesarios datos falsos para probarla. La aplicación *Cupon* por ejemplo no se puede probar a menos que existan tiendas, ofertas y usuarios falsos. Estos datos de prueba también se cargan mediante los *fixtures* o archivos de datos.

5.4.1 Instalando los *fixtures* de Doctrine

Symfony no incluye soporte nativo para *fixtures*, por lo que es necesario instalar un *bundle* específico desarrollado por el proyecto Doctrine y llamado DoctrineFixturesBundle (http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html) .

Como esta es la primera vez que se instala un *bundle* de terceros, se va a explicar el proceso detalladamente.

Paso 1. Instalar el *bundle* en la aplicación.

Gracias al uso de Composer (explicado en el capítulo 1 de este libro), descargar una versión de DoctrineFixturesBundle que sea compatible con la aplicación es tan sencillo como ejecutar un comando de consola:

```
$ cd proyectos\cupon
$ composer require --dev doctrine/doctrine-fixtures-bundle
```

La opción `--dev` indica a la aplicación que esta dependencia solo hace falta en el entorno donde se desarrolla esta aplicación. Por tanto, este bundle no se instalará en el servidor de producción.

Paso 2. Activar el *bundle* en la aplicación.

Además de descargar el *bundle*, es necesario activarlo en la aplicación antes de empezar a usarlo. Para ello, abre el archivo `app/AppKernel.php` y añade la siguiente línea relacionada con `DoctrineFixturesBundle`:

```
// app/AppKernel.php

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            // ...
        );

        if (in_array($this->getEnvironment(), array('dev', 'test'))) {
            // ...
            $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBund
le();
        }

        // ...
    }
}
```

La activación de los *bundles* es automática para todos los que incluye Symfony por defecto, pero tienes que hacerlo manualmente para el resto de *bundles* de terceros que instales.

En este caso, el *bundle* solo hace falta en el entorno de desarrollo, por lo que se activa dentro de `if (in_array($this->getEnvironment(), array('dev', 'test')))`. De esta manera, el *bundle* no se activará cuando la aplicación se ejecute en el entorno de producción, para no perjudicar su rendimiento de ninguna manera.

Para asegurarte de que la instalación ha sido correcta, ejecuta el siguiente comando para listar todos los comandos de la aplicación y comprueba que se muestra el nuevo comando `doctrine:fixtures:load`:

```
$ php app/console
```

5.4.2 Creando el primer archivo de datos

Doctrine define los *fixtures* mediante clases PHP que por convención, se guardan en el directorio `DataFixtures/ORM/` de cada *bundle*. Si los colocas en otro directorio, tendrás que indicar siempre la ruta de los archivos en los comandos que cargan los datos. Puedes crear tantos *fixtures* como quieras para cada *bundle* y el nombre de los archivos de datos lo puedes elegir libremente.

El primer archivo de datos que se va a crear es el que carga la información de las ciudades, por ser el más sencillo. Crea los directorios `DataFixtures/ORM/` dentro del *bundle* `AppBundle` y añade dentro un archivo llamado `Ciudades.php` con el siguiente código:

```
// src/AppBundle/DataFixtures/ORM/Ciudades.php
namespace AppBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\Ciudad;

class Ciudades implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $ciudades = array(
            array('nombre' => 'Madrid', 'slug' => 'madrid'),
            array('nombre' => 'Barcelona', 'slug' => 'barcelona'),
            // ...
        );

        foreach ($ciudades as $ciudad) {
            $entidad = new Ciudad();

            $entidad->setNombre($ciudad['nombre']);
            $entidad->setSlug($ciudad['slug']);

            $manager->persist($entidad);
        }
```

```
            $manager->flush();
        }
    }
```

Para cargar todas las ciudades en la aplicación, ejecuta el siguiente comando:

```
$ php app/console doctrine:fixtures:load
  > purging database
  > loading AppBundle\DataFixtures\ORM\ciudades
```

Observa ahora el contenido de la tabla `Ciudad` de la base de datos. Verás que toda la información de las ciudades definidas en el archivo *fixtures* se ha guardado en la tabla. Además de esto, es importante reseñar la primera línea que muestra la ejecución del comando: *purging database*. En efecto, el comando `doctrine:fixtures:load` **borra todos los contenidos de la base de datos** antes de cargar los nuevos contenidos. Obviamente debes tener mucho cuidado con este comando en el servidor de producción de la aplicación.

Aunque este comando borra toda la información de la base de datos, no reinicializa ni el valor de las claves `id` ni el de cualquier otra columna de tipo `auto_increment`. Para hacer que estos valores vuelvan a empezar en `1`, añade la opción `--purge-with-truncate` de modo que Doctrine borre las tablas con `TRUNCATE` en vez de con `DELETE`:

```
$ php app/console doctrine:fixtures:load --purge-with-truncate
```

Para no borrar los datos existentes, puedes emplear la opción `--append`, que hace que los nuevos datos se añadan a los ya existentes:

```
$ php app/console doctrine:fixtures:load --append
  > loading AppBundle\DataFixtures\ORM\ciudades
```

Como el comando `doctrine:fixtures:load` es bastante largo de escribir, puedes utilizar los atajos de la consola de Symfony. Siempre que el texto introducido se pueda interpretar de forma única, puedes escribir cualquier parte del nombre del comando en vez de su nombre completo. Así por ejemplo, todos los comandos siguientes son equivalentes a `doctrine:fixtures:load`:

```
$ php app/console doctrine:fixtures:loa
$ php app/console doctrine:fixtures:lo
$ php app/console doctrine:fixtures:l
$ php app/console doctrine:fixt:l
$ php app/console doctrine:fix:l
$ php app/console d:fixtures:load
$ php app/console do:fi:lo
$ php app/console d:f:l
```

La estructura de todos los archivos de datos es siempre la misma:

```
namespace <mi-aplicacion>\<mi-bundle>\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;

class <mi-clase> implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // ...
    }
}
```

El código para crear las entidades se incluye dentro del método `load()`. A este método se le pasa como primer parámetro una variable llamada `$manager` que es el objeto del *entity manager*. El código de este primer *fixture* es realmente sencillo, ya que sólo crea una entidad de tipo `Ciudad`, establece su información con los *setters* y la guarda en la base de datos:

```
// src/AppBundle/DataFixtures/ORM/Ciudades.php
// ...
public function load(ObjectManager $manager)
{
    $ciudades = ...

    foreach ($ciudades as $ciudad) {
        $entidad = new Ciudad();

        $entidad->setNombre($ciudad['nombre']);
        $entidad->setSlug($ciudad['slug']);

        $manager->persist($entidad);
    }

    $manager->flush();
}
```

Recuerda que el método `flush()` hace que se escriban los cambios en la base de datos. Así que para mejorar el rendimiento, es mejor que dentro del bucle `foreach` sólo incluyas la llamada al método `persist()`. Después, fuera del bucle se llama al método `flush()` para insertar a la vez todas las ciudades creadas anteriormente.

A pesar de ser un archivo de datos muy sencillo, es posible mejorar su código. El campo `slug` de la entidad guarda el *nombre seguro* de la ciudad. Este *nombre seguro* no contiene espacios en blanco, acentos, eñes o cualquier otro símbolo potencialmente problemático para las URL. Este tipo de campos son esenciales para que la aplicación tenga *URL limpias*.

Calcular el *slug* de una cadena de texto es un procedimiento que se puede automatizar, así que no tiene sentido establecerlo manualmente. El siguiente código muestra una función que calcula el *slug* de una cadena de texto:

```
static public function getSlug($cadena, $separador = '-')
{
    // Código copiado de http://cubiq.org/the-perfect-php-clean-url-generator
    $slug = iconv('UTF-8', 'ASCII//TRANSLIT', $cadena);
    $slug = preg_replace("/[^a-zA-Z0-9\/_|+ -]/", '', $slug);
    $slug = strtolower(trim($slug, $separador));
    $slug = preg_replace("/[\/_|+ -]+/", $separador, $slug);

    return $slug;
}
```

Para utilizarla en cualquier lugar de la aplicación, añade la función anterior a una clase llamada `Slugger.php` dentro del directorio `src/AppBundle/Util/`:

```
// src/AppBundle/Util/Slugger.php
namespace AppBundle\Util;

class Slugger
{
    static public function getSlug($cadena, $separador = '-')
    {
        // ...
    }
}
```

Cuando quieras calcular el *slug* de una cadena dentro de alguna clase, sólo tienes que importar la clase `Slugger` e invocar el método `getSlug()`:

```
use AppBundle\Util\Slugger;

class DefaultController extends Controller
{
    public function portadaAction()
    {
        $titulo = ...
        $slug = Slugger::getSlug($titulo);

        // ...
    }
}
```

Después de definir esta clase, puedes incluso modificar el código de las entidades para que establezcan los *slug* automáticamente:

```php
// src/AppBundle/Entity/Ciudad.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use AppBundle\Util\Slugger;

/** @ORM\Entity */
class Ciudad
{
    // ...

    public function setNombre($nombre)
    {
        $this->nombre = $nombre;
        $this->slug = Slugger::getSlug($nombre);
    }
}
```

Cada vez que establezcas el nombre de una ciudad con el método `setNombre()`, el *slug* de la entidad se actualizará automáticamente. Así, el archivo de datos de las ciudades se puede simplificar a lo siguiente:

```php
// src/AppBundle/DataFixtures/ORM/Ciudades.php
namespace AppBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\Ciudad;

class Ciudades implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $ciudades = array(
            array('nombre' => 'Madrid'),
            array('nombre' => 'Barcelona'),
            // ...
        );

        foreach ($ciudades as $ciudad) {
            $entidad = new Ciudad();
            $entidad->setNombre($ciudad['nombre']);
            $manager->persist($entidad);
        }

        $manager->flush();
    }
}
```

Actualiza también el código de las entidades `Oferta` y `Tienda` para calcular sus *slugs* automáticamente. El resto de capítulos de este libro suponen que está disponible el método `getSlug()` en el lugar explicado anteriormente. Resulta muy recomendable añadir este método y también actualizar el código de las entidades para poder seguir el código de los próximos capítulos.

5.4.3 Creando el resto de archivos de datos

El listado de ciudades es la única información realmente necesaria para que la aplicación funcione bien. Aun así, para poder probar la aplicación sería interesante contar con muchas ofertas, tiendas y usuarios falsos. No se incluye a continuación el código de todos los *fixtures* por razones de espacio y por ser muy similares al archivo de datos explicado anteriormente. El archivo de ofertas por ejemplo tiene el siguiente aspecto:

```
// src/AppBundle/DataFixtures/ORM/Ofertas.php
namespace AppBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\Oferta;

class Ofertas implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 400; $i++) {
            $entidad = new Oferta();

            $entidad->setNombre('Oferta '.$i);
            $entidad->setPrecio(rand(1, 100));
            $entidad->setFechaPublicacion(new \DateTime());
            // ...

            $manager->persist($entidad);
        }

        $manager->flush();
    }
}
```

Puedes conseguir el código completo de todos los *fixtures* en el repositorio público de la aplicación *Cupon* en https://github.com/javiereguiluz/Cupon

El comando `doctrine:fixtures:load` también permite cargar archivos individuales indicando su ruta con la opción `--fixtures`, lo que es muy útil cuando se necesita cargar *fixtures* específicos (por ejemplo, para la ejecución de test unitarios):

```
$ php app/console doctrine:fixtures:load --fixtures=src/AppBundle/DataFixtures/ORM
```

La opción `--fixtures` es una opción *múltiple*, lo que significa que puedes utilizarla varias veces en el mismo comando y se interpreta como si se hubieran indicado todos los valores a la vez en una única opción:

```
php app/console doctrine:fixtures:load --fixtures=... --fixtures=... --fixture
s=...
```

5.4.4 Ordenando los archivos de datos

En ocasiones, es necesario que los *fixtures* se carguen en un orden determinado. Por defecto, el comando `doctrine:fixtures:load` busca los archivos de datos *bundle* por *bundle* y los carga por orden alfabético. En el caso de la aplicación *Cupon*, el orden adecuado es cargar primero las ciudades, después las tiendas y las ofertas y por último, los usuarios.

Resulta sorprendente, pero para que un archivo de datos indique el orden en el que se carga, hay que modificar la clase de la que hereda (`AbstractFixture`), la interfaz que implementa (`OrderedFixtureInterface`) y hay que añadir un método (`getOrder()`). Este es el código completo del *esqueleto* de un archivo de datos que establece el orden en el que se carga:

```
// src/AppBundle/DataFixtures/ORM/Ciudades.php
namespace AppBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\Ciudad;

class Ciudades extends AbstractFixture implements OrderedFixtureInterface
{
    public function getOrder()
    {
        return 1;
    }

    public function load(ObjectManager $manager)
    {
        // ...
    }
}
```

El método `getOrder()` devuelve un número que indica la posición de este archivo de datos respecto a los demás, por lo que cuanto más bajo sea el número, antes se carga el archivo.

5.4.5 Cargando los datos de prueba de *Cupon*

En el repositorio de la aplicación *Cupon* puedes encontrar todos sus *fixtures* o archivos de datos, por lo que no es necesario que los crees a mano.

Si estás desarrollando la aplicación desde cero a medida que lees el libro, tienes que cargar los *fixtures* básicos, que no hacen uso del componente de seguridad que se explica más adelante. Para ello, ejecuta el siguiente comando:

```
$ php app/console doctrine:fixtures:load --fixtures=app/Resources
```

Este comando sólo carga los *fixtures* del archivo `Basico.php` que se encuentra en el directorio `app/Resources/`.

Por el contrario, si estás probando la aplicación *Cupon* terminada, puedes cargar los datos de prueba completos con el siguiente comando:

```
$ php app/console doctrine:fixtures:load
```

Este comando sólo funciona si has configurado bien la seguridad de la aplicación (tal y como se explica en los próximos capítulos) ya que en uno de los *fixtures* se codifican las contraseñas de los usuarios utilizando el componente de seguridad.

5.5 Alternativas para generar el modelo

Las secciones anteriores explican la forma recomendada de trabajar con bases de datos en Symfony. No obstante, según las circunstancias particulares de cada proyecto, puede resultarte más cómodo utilizar alguno de los métodos alternativos que se muestran a continuación.

5.5.1 Configurando las entidades con YAML o XML

Si no te sientes cómodo configurando la información de las entidades mediante anotaciones, puedes hacerlo en formato YAML e incluso XML. Para ello, crea un archivo YAML para cada entidad de la aplicación. El nombre del archivo coincide con el nombre de la entidad y su extensión debe ser `.orm.yml`. Dentro de cada *bundle* estos archivos de configuración se guardan en el directorio `Resources/config/doctrine/`, que debes crear manualmente.

```
# src/AppBundle/Resources/config/doctrine/Ciudad.orm.yml
AppBundle\Entity\Ciudad:
    type: entity
    # la siguiente opción es opcional y define el repositorio
    # asociado con esta entidad, tal y como se explicará más adelante
    repositoryClass: AppBundle\Repository\CiudadRepository
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        nombre:
            type:   string
            length: 100
        slug:
            type:   string
            length: 100
```

Una vez creados los archivos de configuración, puedes generar las clases de las entidades mediante el comando `doctrine:generate:entities`:

```
// Genera todas las clases de todos los bundles
$ php app/console doctrine:generate:entities

// Genera sólo las clases del bundle AppBundle
$ php app/console doctrine:generate:entities AppBundle

// Genera sólo la clase de la entidad Ciudad del bundle AppBundle
$ php app/console doctrine:generate:entities AppBundle:Ciudad
```

Igualmente, la configuración de la entidad se puede definir mediante archivos XML. Aunque son mucho más largos de escribir que los archivos YAML, cuentan con la ventaja de que su contenido se puede validar a medida que se escribe. Estos archivos también se guardan en el directorio `Resources/config/doctrine/` del *bundle* y su nombre coincide con el nombre de cada entidad, pero su extensión ahora es `.orm.xml`.

```
<!-- src/AppBundle/Resources/config/doctrine/Ciudad.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mappin
g"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mappi
ng http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <!-- el atributo repository-class es opcional y define el repositorio asocia
do
         con esta entidad, tal y como se explicará más adelante -->
    <entity name="AppBundle\Entity\Ciudad" table="ciudad"
            repository-class="AppBundle\Repository\CiudadRepository">
        <id name="id" type="integer" column="id">
            <generator strategy="AUTO" />
        </id>
        <field name="nombre" column="nombre" type="string" length="100" />
        <field name="slug" column="slug" type="string" length="100" />
    </entity>
</doctrine-mapping>
```

Después de definir los archivos de configuración, las clases de la entidad se generan con el mismo comando `doctrine:generate:entities` explicado anteriormente.

Por último, una limitación muy importante que siempre debes tener en cuenta al utilizar formatos alternativos es que **en un mismo *bundle* no se pueden mezclar formatos diferentes para definir las entidades**.

5.5.2 Ingeniería inversa

Algunos proyectos web obligan a utilizar una base de datos existente y que no se puede modificar. En estos casos no es necesario crear las entidades con las herramientas explicadas en las secciones anteriores. Lo mejor es crear las entidades directamente a partir de la base de datos mediante las herramientas de ingeniería inversa que incluye Doctrine.

En primer lugar, configura la información de acceso a la base de datos mediante el archivo `app/config/parameters.yml` (o, si lo prefieres, en el archivo `app/config/config.yml`). Después, ejecuta el siguiente comando para transformar la estructura completa de tablas de la base de datos en las clases PHP de las entidades:

```
$ php app/console doctrine:mapping:import AppBundle annotation

Importing mapping information from "default" entity manager
  > writing .../AppBundle/Entity/Ciudad.php
  > writing .../AppBundle/Entity/Oferta.php
  > writing .../AppBundle/Entity/Tienda.php
  // ...
```

El primer argumento del comando es el nombre del *bundle* en el que se guardan todos los archivos generados automáticamente. El segundo argumento indica el formato utilizado para definir la información de cada entidad. El valor `annotation` hace que la información se defina mediante anotaciones. Por esa razón el comando anterior genera una clase PHP por cada tabla de la base de datos e incluye todas las anotaciones necesarias para reflejar la estructura de tablas y el tipo de columnas.

Si el formato indicado es `yml` o `xml`, no se generan las clases PHP sino los archivos de configuración YAML o XML que se guardan en el directorio `Resources/config/doctrine` del *bundle*.

Para completar las entidades, añade los *getters* y *setters* con el siguiente comando:

```
$ php app/console doctrine:generate:entities AppBundle
```

Generar las entidades mediante ingeniería inversa es un proceso sencillo y que funciona muy bien, ya que Doctrine tiene en cuenta la información de las tablas, columnas, índices, claves primarias y externas. No obstante, el manual de Doctrine asegura que cuando el esquema de la base de datos es muy complejo, solamente se puede obtener entre el 70 y el 80% de la información. Conceptos como la herencia de entidades, las asociaciones inversas, los eventos y otra información avanzada tendrás que configurarla manualmente.

Esta página se ha dejado vacía a propósito

CAPÍTULO 6

Creando la portada

El capítulo 4 (página 49) introdujo el funcionamiento interno de Symfony creando las páginas básicas del sitio web. Se explicaron brevemente los conceptos de controladores, acciones y rutas y se mostró en la práctica el flujo de trabajo de Symfony. En este capítulo se explica más detalladamente el comportamiento interno de Symfony y para ello se desarrolla la portada de la aplicación, que es una página compleja y dinámica.

Recordando el funcionamiento de la aplicación explicado en el capítulo 2 (página 29), la portada *"es el punto de entrada natural al sitio web y también la página que se muestra al pinchar la opción Oferta Del Día en el menú de navegación. Si accede un usuario registrado y logueado, se muestra la oferta del día de su ciudad. Si accede un usuario anónimo, se muestra la oferta del día de la ciudad por defecto"*. Como todavía no se va a trabajar con usuarios registrados, nos centraremos exclusivamente en mostrar la oferta del día de la ciudad por defecto.

6.1 Arquitectura MVC

Symfony basa su funcionamiento interno en la famosa arquitectura Modelo - Vista - Controlador (MVC) (http://es.wikipedia.org/wiki/Modelo_Vista_Controlador) , utilizada por la mayoría de *frameworks* web. No obstante, según su creador Fabien Potencier: *"Symfony no es un framework MVC. Symfony sólo proporciona herramientas para la parte del Controlador y de la Vista y no para la parte del Modelo"* (fuente: http://fabien.potencier.org/article/49/what-is-Symfony).

En cualquier caso, resulta esencial conocer cómo se aplican los principios fundamentales de la arquitectura MVC a las aplicaciones Symfony. Observa el siguiente esquema simplificado del funcionamiento interno de Symfony (al final de este capítulo se muestra un esquema mucho más técnico y detallado):

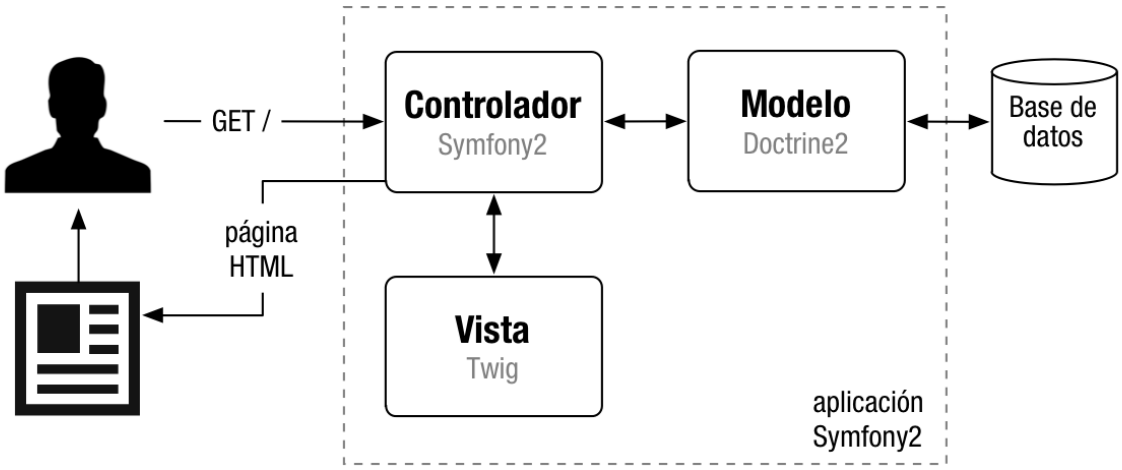

Figura 6.1 Esquema simplificado de la arquitectura interna de Symfony

Cuando el usuario solicita ver la portada del sitio, internamente sucede lo siguiente:

1. El sistema de enrutamiento determina qué **Controlador** está asociado con la página de la portada.
2. Symfony ejecuta el **Controlador** asociado a la portada. Un *controlador* no es más que una clase PHP en la que puedes ejecutar cualquier código que quieras.
3. El **Controlador** solicita al **Modelo** los datos de la oferta del día. El *modelo* no es más que una clase PHP especializada en obtener información, normalmente de una base de datos (en este caso, el modelo está formado por las entidades de Doctrine).
4. Con los datos devueltos por el **Modelo**, el **Controlador** solicita a la **Vista** que cree una página mediante una plantilla y que inserte los datos del **Modelo**.
5. El **Controlador** entrega al servidor la página creada por la **Vista**.

A pesar de que puedes llegar a hacer cosas muy complejas con Symfony, el funcionamiento interno siempre es el mismo: 1) el **Controlador** da órdenes, 2) el **Modelo** busca la información que se le pide, 3) la **Vista** crea páginas con plantillas y variables.

6.2 El enrutamiento

El sistema de enrutamiento asocia las URL y los controladores. Más en concreto, determina qué código PHP, llamado **acción**, se ejecuta cuando el usuario solicita una determinada URL.

El archivo principal de enrutamiento de Symfony es `app/config/routing.yml`. Aunque puedes definir las rutas directamente en ese archivo, normalmente se importan desde otros lugares. Por defecto su contenido es el siguiente:

```
# app/config/routing.yml
app:
```

```
    resource: "@AppBundle/Controller/"
    type:     annotation
```

Esta configuración le dice a Symfony que entre en el directorio `Controller/` del *bundle* AppBundle y busque todas las rutas que hayan sido definido como anotaciones en las clases que pueda haber dentro de ese directorio.

Los controladores se buscan alfabéticamente (ej. las rutas de `OfertaController` se cargan antes que las de `UsuarioController`) y dentro de cada controlador se añaden las rutas en el mismo orden en el que se han definido los métodos. Si necesitas cargar las rutas en un orden determinado, puedes importar los controladores individualmente desde el archivo `app/config/routing.yml`. Para reordenar las rutas dentro de un controlador, reordena sus métodos.

Si lo prefieres, puedes definir las rutas como archivos YAML, XML o PHP dentro de cada *bundle*. Por convención, estos archivos se guardan en el directorio `Resources/config/` del *bundle* y se importarían de la siguiente manera:

```
# app/config/routing.yml
# ...
oferta:
    resource: "@OfertaBundle/Resources/config/routing.yml"
```

En el caso de la portada de la aplicación, como se trata de una ruta muy especial, se podría definir directamente en el archivo `app/config/routing.yml`. Sin embargo, otra práctica muy común consiste en definirla directamente en el controlador `DefaultController`:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="portada")
     */
    public function portadaAction()
    {
        // ...
    }
}
```

6.3 El controlador

Los *controladores* contienen el código PHP que se ejecuta para responder a las peticiones de los usuarios. Técnicamente un **controlador** es una clase PHP que contiene uno o más metodos llamados **acciones**. No obstante, en la práctica se utiliza la palabra **controlador** también para referirse a las **acciones** individuales (ej. *"el controlador de la portada", en vez de *"la acción de la portada"*).

En las aplicaciones bien diseñadas, las acciones tienen muy poco código, ya que solo se encargan de llamar a otras partes de la aplicación que son las que realmente hacen el trabajo (buscar información en una base de datos, enviar un email, etc.)

Las aplicaciones Symfony incluyen por defecto un controlador llamado `DefaultController` dentro del *bundle* AppBundle. Este es el controlador ideal para definir las acciones genéricas de la aplicación (como la portada o las páginas estéticas), pero si no lo vas a utilizar, puedes borrarlo y crear tus propios controladores.

En el caso de la portada, lo único que debe hacer este controlador es buscar la oferta del día en la ciudad por defecto y después pasar los datos a la plantilla de la portada. Así que en primer lugar obtén el objeto del *entity manager* necesario para hacer consultas a la base de datos:

```
// src/AppBundle/Controller/DefaultController.php
// ...

/**
 * @Route("/", name="portada")
 */
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();
}
```

Para buscar la oferta del día, puedes hacer uso del método `findOneBy()` de Doctrine estableciendo como condiciones que la oferta sea de una determinada ciudad y que se haya publicado hoy:

```
// src/AppBundle/Controller/DefaultController.php
// ...

/**
 * @Route("/", name="portada")
 */
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();
    $oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(array(
        'ciudad' => 1,
        'fechaPublicacion' => new \DateTime('today')
    ));
}
```

Recuerda que, aunque la propiedad de la entidad `Oferta` se llame `$ciudad`, para realizar las búsquedas siempre hay que indicar el valor de su atributo `id`, ya que en la tabla de la base de datos se guarda como `ciudad_id`. No olvides tampoco la barra `\` por delante de la clase `DateTime`, obligatorio debido al uso de los *namespaces* en el código fuente de Symfony, tal y como se explicó en la sección de *namespaces* (página 22) del capítulo 1.

El objeto `$oferta` ya contiene los datos de la oferta del día en la ciudad indicada. El trabajo del controlador ya ha finalizado, por lo que sólo resta indicar a la vista qué plantilla debe utilizar para generar la página que se entrega al usuario.

Para generar una página de respuesta a partir de una plantilla, utiliza el método `render()` que se define en el controlador base del que hereda `DefaultController`. El primer argumento indica la plantilla a utilizar y el segundo argumento opcional es un array con todas las variables que se pasan a la plantilla:

```
// src/AppBundle/Controller/DefaultController.php
// ...

/**
 * @Route("/", name="portada")
 */
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();
    $oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(array(
        'ciudad' => 1,
        'fechaPublicacion' => new \DateTime('today')
    ));

    return $this->render('portada.html.twig', array(
        'oferta' => $oferta
    ));
}
```

El código del controlador ya está completo. Al usuario se le devolverá la página que Symfony genere mediante la plantilla `portada.html.twig` (guardada en el directorio `app/Resources/views/`) y la variable llamada `oferta` que contiene todos los datos de la oferta del día.

6.4 La plantilla

Todas las plantillas desarrolladas en este libro utilizan Twig como lenguaje de plantillas. Si no lo has hecho todavía, lee el apéndice A (página 451) en el que se explica detalladamente toda su sintaxis. De lo contrario no entenderás nada de lo que se desarrolla a continuación.

El controlador de la portada renderiza la plantilla `portada.html.twig`, que se corresponde con el archivo `app/Resources/views/portada.html.twig`. Crea ese archivo y añade el siguiente contenido HTML:

```
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Portada | Cupon</title>
</head>
```

```
<body id="portada"><div id="contenedor">
    <header>
        <h1><a href="#">CUPON</a></h1>
        <nav>
            <ul>
                <li><a href="#">Oferta del día</a></li>
                <li><a href="#">Ofertas recientes</a></li>
                <li><a href="#">Mis ofertas</a></li>
            </ul>
        </nav>
    </header>

    <article class="oferta">
        <section class="descripcion">
            <h1><a href="#"> ## NOMBRE DE LA OFERTA ## </a></h1>
             ## DESCRIPCIÓN DE LA OFERTA ##
            <a class="boton" href="#">Comprar</a>
        </section>

        <section class="galeria">
            <img alt="Fotografía de la oferta" src="#">
            <p class="precio">## PRECIO ## &euro;
               <span>## DESCUENTO ##</span></p>
            <p><strong>Condiciones:</strong> ## CONDICIONES ##</p>
        </section>

        <section class="estado">
            <div class="tiempo">
                <strong>Faltan</strong>: ## FECHA DE EXPIRACIÓN ##
            </div>

            <div class="compras">
                <strong>Compras</strong>: ## COMPRAS TOTALES ##
            </div>

            <div class="faltan">
                {# Si las compras no llegan al umbral mínimo establecido #}
                Faltan <strong>## NN ## compras</strong> <br/>
                para activar la oferta

                {# Si las compras superan el umbral mínimo establecido #}
                <strong>Oferta activada</strong> por superar las
                <strong>## NN ##</strong> compras necesarias
            </div>
        </section>

        <section class="direccion">
```

```
            <h2>Disfruta de la oferta en</h2>
            <p>
                <a href="#">## NOMBRE DE LA TIENDA ##</a>
                ## DIRECCION DE LA TIENDA ##
            </p>
        </section>

        <section class="tienda">
            <h2>Sobre la tienda</h2>
            ## DESCRIPCION DE LA TIENDA ##
        </section>
    </article>

    <aside>
        ## FORMULARIO DE LOGIN ##

        <section id="nosotros">
            <h2>Sobre nosotros</h2>
            <p>Lorem ipsum dolor sit amet...</p>
        </section>
    </aside>

    <footer>
        &copy; 201X - Cupon
        <a href="#">Ayuda</a>
        <a href="#">Contacto</a>
        <a href="#">Privacidad</a>
        <a href="#">Sobre nosotros</a>
    </footer>
</div></body>

</html>
```

La plantilla de la portada por el momento sólo contiene código HTML, que es el punto de partida habitual antes de transformarla en una plantilla Twig. En primer lugar, modifica el elemento `<header>` para añadir enlaces a la portada del sitio:

```
<header>
    <h1><a href="{{ path('portada') }}">CUPON</a></h1>
    <nav>
        <ul>
            <li><a href="{{ path('portada') }}">Oferta del día</a></li>
            ...
        </ul>
    </nav>
</header>
```

Los enlaces de las plantillas Twig se incluyen mediante la función `path()` a la que se le pasa como primer argumento el nombre de la ruta. De esta forma, los enlaces de la plantilla se generan diná-

micamente en el mismo instante que se *renderiza* la plantilla. La gran ventaja respecto a escribir a mano las URL es que si modificas la configuración de las rutas, todos los enlaces de la aplicación se actualizan instantáneamente y sin tener que hacer ningún cambio.

Los enlaces del pie de página (elemento `<footer>` de la página) no son tan sencillos, ya que todos utilizan la misma ruta llamada `pagina` pero cada uno apunta a una página diferente. La solución consiste en pasar una variable con el nombre de la página como segundo parámetro de la función `path()` de Twig:

```
<footer>
    &copy; {{ 'now'|date('Y') }} - Cupon
    <a href="{{ path('pagina', { nombrePagina: 'ayuda' }) }}">
        Ayuda
    </a>
    ...
    <a href="{{ path('pagina', { nombrePagina: 'privacidad' }) }}">
        Privacidad
    </a>
    <a href="{{ path('pagina', { nombrePagina: 'sobre-nosotros' }) }}">
        Sobre nosotros
    </a>
</footer>
```

Cuando se *renderice* la plantilla, Symfony utilizará el patrón de la ruta (`/sitio/{nombrePagina}`) y sustituirá todas sus variables por los valores que se pasan como segundo argumento de la función `path()`. Así la primera ruta será `/sitio/ayuda`, la segunda `/sitio/contacto`, etc.

Observa cómo el código anterior hace uso de un truco muy interesante para obtener el año actual (`'now'|date('Y')`). El filtro `date()` admite cualquier valor que también sea válido para la clase `DateTime` de PHP, lo que incluye cualquier cadena de texto en inglés que se pueda interpretar como una fecha (`now`, `today`, `yesterday`, `tomorrow`, `now - 2 hours`, `today - 1 year`, `tomorrow + 3 weeks`, etc.)

A continuación, completa toda la información sobre la oferta. Para ello, recuerda que desde el controlador se pasa a la plantilla una variable llamada `oferta` que contiene todos los datos de la oferta del día en una entidad de tipo `Oferta`. La primera parte de la plantilla es muy sencilla porque sólo muestra el valor de algunas propiedades de la oferta. Para ello, se utiliza la notación `variable.propiedad`:

```
<section class="descripcion">
    <h1><a href="#">{{ oferta.nombre }}</a></h1>
    {{ oferta.descripcion }}
    <a class="boton" href="#">Comprar</a>
</section>
```

La siguiente sección de código incluye también la ruta de la imagen de la oferta:

```
<section class="galeria">
    <img alt="Fotografía de la oferta" src="{{ asset('uploads/images/' ~ ofert
```

```
a.rutaFoto) }}">

    <p class="precio">{{ oferta.precio }} &euro; <span>{{ oferta.descuento }}</s
pan></p>

    <p><strong>Condiciones:</strong> {{ oferta.condiciones }}</p>
</section>
```

La función `asset()` de Twig genera la URL pública del elemento que se le pasa (imagen, archivo CSS o JavaScript). En la mayoría de proyectos Symfony la carpeta pública es `web/` (el valor por defecto que utiliza Symfony) por lo que las URL generadas por `asset()` son del tipo `/web/...`

Utilizar la función `asset()` de Twig en vez enlazar los archivos directamente permite que la aplicación sea más flexible. La función `asset()` tiene en cuenta la configuración de Symfony al generar las URL. De esta manera podrás mover los archivos CSS y JS añadiéndoles algún prefijo a su URL, podrás versionarlos añadiéndole algún parámetro en sus URL, etc. y todo ello simplemente modificando alguna opción del archivo `app/config/config.yml` en vez de tener que modificar todas las plantillas.

Como las tiendas podrán crear ofertas desde la extranet, es mejor guardar las fotos dentro del directorio `web/uploads/` del proyecto, que es el lugar preparado para que los usuarios puedan subir contenidos. Para separar las imágenes del resto de contenidos, crea el directorio `web/uploads/images/`.

Por otra parte, en la entidad (es decir, en la base de datos) no se almacena la ruta completa de cada foto, sino solamente el nombre del archivo (`foto1.jpg`, `foto-aSdfE.jpg`, etc.) Así que para obtener la ruta relativa de la oferta, se concatena la cadena de texto `uploads/images/` y el valor de la propiedad `oferta.rutaFoto`.

El siguiente fragmento de la plantilla muestra fechas y define variables propias:

```
<section class="estado">
    <div class="tiempo">
        <strong>Faltan</strong>: {{ oferta.fechaExpiracion|date }}
    </div>

    <div class="compras">
        <strong>Compras</strong>: {{ oferta.compras }}
    </div>

    <div class="faltan">
        {% set faltan = oferta.umbral - oferta.compras %}
        {% if faltan > 0 %}
            Faltan <strong>{{ faltan }} compras</strong> <br/>
            para activar la oferta
        {% else %}
            <strong>Oferta activada</strong> por superar las
            <strong>{{ oferta.umbral }}</strong> compras necesarias
```

```
        {% endif %}
    </div>
</section>
```

La propiedad `fechaExpiracion` de la entidad guarda la fecha como un objeto de tipo `DateTime`. Si utilizas la instrucción `{{ oferta.fechaExpiracion }}` Twig muestra un error porque no sabe convertir el valor `DateTime` en una cadena de texto. Así que siempre que muestres una fecha en una plantilla, no olvides añadir el filtro `|date`.

Después, la plantilla debe tomar una decisión: si se han producido más compras del mínimo necesario, la oferta se activa para todos los usuarios. Si no, se muestran cuántas compras faltan para que se active la oferta. Las decisiones se toman con la estructura de control `if..else` pero para hacer más conciso el código, también se crea una variable llamada `faltan`:

```
{% set faltan = oferta.umbral - oferta.compras %}

{% if faltan > 0 %}
    Faltan {{ faltan }} compras para activar la oferta
{% else %}
    Oferta activada por superar las {{ oferta.umbral }} compras necesarias
{% endif %}
```

La última parte de la plantilla es una de las más interesantes:

```
<section class="direccion">
    <h2>Disfruta de la oferta en</h2>
    <p>
        <a href="#">{{ oferta.tienda.nombre }}</a>
        {{ oferta.tienda.direccion }}
    </p>
</section>

<section class="tienda">
    <h2>Sobre la tienda</h2>
    {{ oferta.tienda.descripcion }}
</section>
```

En la tabla `Oferta` de la base de datos existe una columna llamada `tienda_id` que sólo guarda el atributo `id` de la tienda asociada a la oferta. Sin embargo, en la entidad `Oferta` la propiedad se llama `tienda` y Doctrine hace que su valor sea la entidad `Tienda` completa asociada a la `Oferta`.

La instrucción `{{ oferta.tienda }}` hace que todos los datos de la tienda estén disponibles en la plantilla, por lo que para mostrar su nombre, simplemente se debe indicar `{{ oferta.tienda.nombre }}` y su dirección se obtendría como `{{ oferta.tienda.direccion }}`

Este comportamiento es posible gracias al *lazy loading* de Doctrine, que busca automáticamente cualquier información no disponible. Recuerda que su desventaja es que puede aumentar rápidamente el número de consultas a la base de datos. Al incluir `{{ oferta.tienda }}` en la plantilla, Doctrine realiza una consulta a la base de datos para obtener la información de la tienda. Así que

la portada requiere dos consultas: la primera es la que realiza el controlador y la otra es la que Doctrine realiza automáticamente debido a la plantilla.

La plantilla ya está completa, por lo que puedes probar la página accediendo con tu navegador a la URL `http://127.0.0.1:8000/app_dev.php` Si en la base de datos existe una oferta que cumpla las condiciones de búsqueda del controlador, verás una página con todos los datos de la oferta. Si no, el objeto `$oferta` del controlador será `null` y por eso verás un error de Twig (*Item "nombre" for "" does not exist ...*)

6.5 Entornos de ejecución

Una aplicación de Symfony no sólo se compone del código fuente incluido en el directorio `src/`, sino que también se define mediante los archivos de configuración que controlan su comportamiento. Un entorno es la suma del código fuente y los archivos de configuración con los que se ejecuta la aplicación en un momento dado.

Accede con tu navegador a la URL `http://127.0.0.1:8000/app_dev.php` y después accede a `http://127.0.0.1:8000/app.php`. Aunque la aplicación es la misma y el código fuente es idéntico en los dos casos, la portada se ve diferente. Estas diferencias son mucho mayores en las páginas de error: accede a `http://127.0.0.1:8000/app_dev.php/esto-no-existe` y después accede a `http://127.0.0.1:8000/app.php/esto-no-existe`. Es la misma página, la misma aplicación, el mismo código, pero no se parecen en nada.

Cuando la URL de una página incluye `app_dev.php`, la aplicación se ejecuta en el entorno `dev` o *entorno de desarrollo*. Este entorno está pensado para que lo usen los programadores mientras desarrollan la aplicación. Ejecutar así la aplicación es mucho más lento, pero todas las páginas incluyen información útil para los programadores (sobre todo, las páginas de error).

De la misma forma, cuando la URL de una página incluye `app.php`, la aplicación se ejecuta en el entorno `prod` o *entorno de producción*. Este es el entorno en el que se debe ejecutar la aplicación en el servidor de producción, de ahí su nombre. Las páginas no incluyen ningún tipo de información útil para los programadores (y tampoco para los usuarios maliciosos), pero a cambio la aplicación se ejecuta lo más rápido posible.

Mientras desarrollas la aplicación, se recomienda que utilices siempre el entorno de desarrollo. Cuando acabes una funcionalidad completa, pruébala en el entorno de producción para comprobar que todo funciona bien. Symfony incluye por defecto un tercer entorno llamado `test` que no se utiliza en el navegador (no existe el archivo `app_test.php`) pero que es imprescindible para los tests unitarios y funcionales (como se explica en el capítulo 11 (página 283)).

6.5.1 Archivos de configuración

Las diferencias en el comportamiento de la aplicación según el entorno de ejecución se controlan mediante los archivos de configuración del directorio `app/config/`. Hasta ahora, sólo se ha utilizado el archivo de *configuración general* `app/config/config.yml`, pero el funcionamiento de Symfony es mucho más avanzado.

Si observas el código de los archivos `web/app.php` o `web/app_dev.php` verás lo siguiente:

```
// web/app.php
// ...
$kernel = new AppKernel('prod', false);
// ...

// web/app_dev.php
// ...
$kernel = new AppKernel('dev', true);
// ...
```

Y si ahora observas el código de la clase `app/AppKernel.php`:

```
// app/AppKernel.php

// ...
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
}
```

Así que, en realidad, cuando ejecutas la aplicación en el entorno de desarrollo, se carga el archivo de configuración `app/config/config_dev.yml` y en producción se carga `app/config/config_prod.yml`. No obstante, como las diferencias de configuración entre entornos son importantes pero escasas, Symfony define un tercer archivo llamado `app/config/config.yml` que recoge toda la configuración común de los dos entornos.

La primera instrucción del archivo de configuración de desarrollo y de producción consiste en importar este archivo de configuración común:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }
# ...

# app/config/config_prod.yml
imports:
    - { resource: config.yml }
# ...
```

Mediante la clave `imports` puedes incluir recursos (`resource`) que se encuentren en cualquier directorio, aunque estén escritos en otros formatos (YAML, PHP, XML o INI). Después de la importación, puedes añadir nuevas opciones de configuración o puedes redefinir el valor de cualquier opción importada.

Observa por ejemplo las siguientes líneas del archivo `app/config/config.yml`:

```
# app/config/config.yml

# ...
```

```
framework:
    router: { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

La opción `router` bajo la clave `framework` indica que las rutas de la aplicación se definen en el archivo `app/config/routing.yml` (el valor `%kernel.root_dir%` equivale al directorio donde se encuentra la clase `AppKernel.php`, que casi siempre es `app/`).

A continuación, observa las siguientes líneas del archivo `app/config/config_dev.yml`:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: "%kernel.root_dir%/config/routing_dev.yml" }
    profiler: { only_exceptions: false }

web_profiler:
    toolbar: true
    intercept_redirects: false

# ...
```

La opción `imports` hace que todas las opciones del archivo `/app/config/config.yml` se añadan en este archivo, incluyendo la opción `router`. Sin embargo, en el entorno de desarrollo son necesarias algunas rutas más que en el resto de entornos. Para ello, después de la importación se añade una opción con el mismo nombre que la del archivo original (`router`) y se establece un nuevo valor (`app/config/routing_dev.yml`).

Así que en todos los entornos se utiliza el archivo `app/config/routing.yml` pero en el entorno de desarrollo se emplea `app/config/routing_dev.yml`. Esta característica se puede aprovechar por ejemplo para utilizar bases de datos diferentes según se ejecute la aplicación en desarrollo o producción. Para ello sólo debes añadir en el archivo `app/config/config_prod.yml` los datos de conexión de la base de datos de producción y utilizar los datos del archivo `app/config/config.yml` en el resto de entornos.

Saltar de un archivo a otro y distribuir la configuración en muchos archivos diferentes puede resultarte confuso si es la primera vez que lo ves. Sin embargo, esta es una de las claves de la flexibilidad que ofrece Symfony.

6.5.2 Comandos de consola

Elegir el entorno bajo el que se ejecuta la aplicación no sólo es importante cuando se accede a ella a través de un navegador. Al ejecutar los comandos de consola de Symfony resulta esencial indicar el entorno bajo el que se ejecuta y por tanto, la configuración utilizada por cada comando.

El nombre del entorno de ejecución se indica mediante la opción `--env`. Cuando no utilizas esta opción, se sobreentiende que has seleccionado el entorno de desarrollo (`dev`):

```
// Estos dos comandos son equivalentes
$ php app/console doctrine:fixtures:load
$ php app/console doctrine:fixtures:load --env=dev

// El comando se ejecuta en producción, lo que
// permite por ejemplo utilizar otra base de datos
$ php app/console doctrine:fixtures:load --env=prod
```

6.5.3 La caché

Symfony genera cientos de archivos de caché en el directorio `app/cache/` para mejorar el rendimiento de las aplicaciones. Cada entorno cuenta con su propia caché, localizada en un directorio con el mismo nombre del entorno (`app/cache/dev/`, `app/cache/prod/`, etc.)

Tal y como se explica en los próximos capítulos, algunas operaciones requieren borrar la caché para regenerarla con nueva información. Esta tarea se realiza con el comando `cache:clear`:

```
// Limpiar la caché de desarrollo
$ php app/console cache:clear
$ php app/console cache:clear --env=dev

// Limpiar la caché de producción
$ php app/console cache:clear --env=prod
```

El comando para borrar la caché define otra opción muy interesante llamada `--no-warmup`. Por defecto, el comando `cache:clear` no sólo borra todos los contenidos de la caché, sino que también regenera los cientos de archivos que forman la caché (archivos de Doctrine, de Twig, de anotaciones, del sistema de enrutamiento, etc.)

Este comportamiento es ideal para las aplicaciones que están en producción, ya que cuando el primer usuario acceda al sitio web, toda la caché ya estará llena de los archivos necesarios. La gran desventaja es que este proceso consume un tiempo no despreciable que debes esperar antes de ejecutar nuevamente la aplicación.

Añadiendo la opción `--no-warmup`, el comando `cache:clear` solamente borra los contenidos de la caché, por lo que su ejecución es casi instantánea. Después, la primera vez que accedas a la aplicación se genera la caché de todos los archivos necesarios para ejecutar esa página.

Si quieres agilizar el desarrollo de la aplicación, no olvides utilizar los siguientes comandos para borrar la caché:

```
// Limpiar la caché de desarrollo
$ php app/console cache:clear --no-warmup --env=dev
```

```
// Limpiar la caché de producción
$ php app/console cache:clear --env=prod
```

Si se produce algún error al ejecutar los comandos anteriores, repasa el apartado de configuración de permisos (página 60) del capítulo 4.

6.6 Depurando errores

Cometer errores es inevitable al desarrollar una aplicación, por lo que Symfony te ayuda a encontrar lo más fácil y rápido posible la causa del error gracias a las siguientes herramientas:

- **Barra de depuración web**, que se incluye en la parte inferior de todas las páginas que se ejecutan en el entorno de desarrollo. Muestra de un vistazo mucha información importante sobre la página actual.
- **El *profiler***, que muestra toda la información de configuración y de ejecución de la página. Como se guardan todos los datos históricos, permite realizar el *profiling* o análisis del rendimiento de la aplicación a lo largo del tiempo.
- **Los archivos de log**, que guardan toda la información sobre la ejecución interna de la aplicación y que permiten realizar un análisis exhaustivo en los casos que así lo requieran.

Antes de explicar el uso de cada una de estas herramientas, es importante introducir la opción `debug` de Symfony. Si abres cualquiera de los archivos `web/app_*.php`, verás que al instanciar la clase `app/AppKernel.php` se le pasan dos parámetros:

```
// web/app_prod.php

// ...
$kernel = new AppKernel('prod', false);
```

El primer parámetro es el nombre del entorno de ejecución. El segundo parámetro es el valor de la opción `debug`, que indica si la aplicación se debe ejecutar en el *modo debug*, también conocido como *modo depuración*. Cuando la opción `debug` vale `true`, la aplicación guarda todos los mensajes de log, los errores muestran toda su información por pantalla, la caché se regenera en cada petición, etc.

Aunque la opción `debug` es completamente independiente del entorno bajo el que se ejecuta la aplicación, normalmente se asocia el valor `true` con el entorno de desarrollo y el valor `false` con el entorno de producción. Pero si la aplicación por ejemplo no te funciona bien en el entorno de producción, quizás debas establecer la opción `debug` a `true` hasta que encuentres la causa del error.

En el archivo de configuración `app/config/config.yml` verás que se utiliza en muchas opciones el valor `%kernel.debug%` que es precisamente el valor de la opción `debug` de Symfony.

6.6.1 La barra de depuración web

La *barra de depuración web* es la forma más inmediata de visualizar información sobre la ejecución de la página actual. Cuando se encuentra activa, en la parte inferior de todas las páginas verás algo como lo siguiente:

Figura 6.2 Barra de depuración web normal de Symfony

La información incluida en la barra depende del código ejecutado. Por ejemplo, si se realizan consultas a la base de datos, verás un panel con el número de consultas y el tiempo empleado en ellas. Si no se realizan consultas, la barra no incluye ninguna información sobre la base de datos.

6.6.2 El *profiler*

Los paneles que se muestran en la barra de depuración web se pueden pinchar para mostrar información más detallada. Esta información se genera mediante el *profiler*, que incluye toda la información imprescindible para detectar aquellos errores que no logras localizar ni con la barra de depuración web ni con los datos que muestra la página de error del entorno de desarrollo. La siguiente imagen muestra el aspecto inicial del *profiler*:

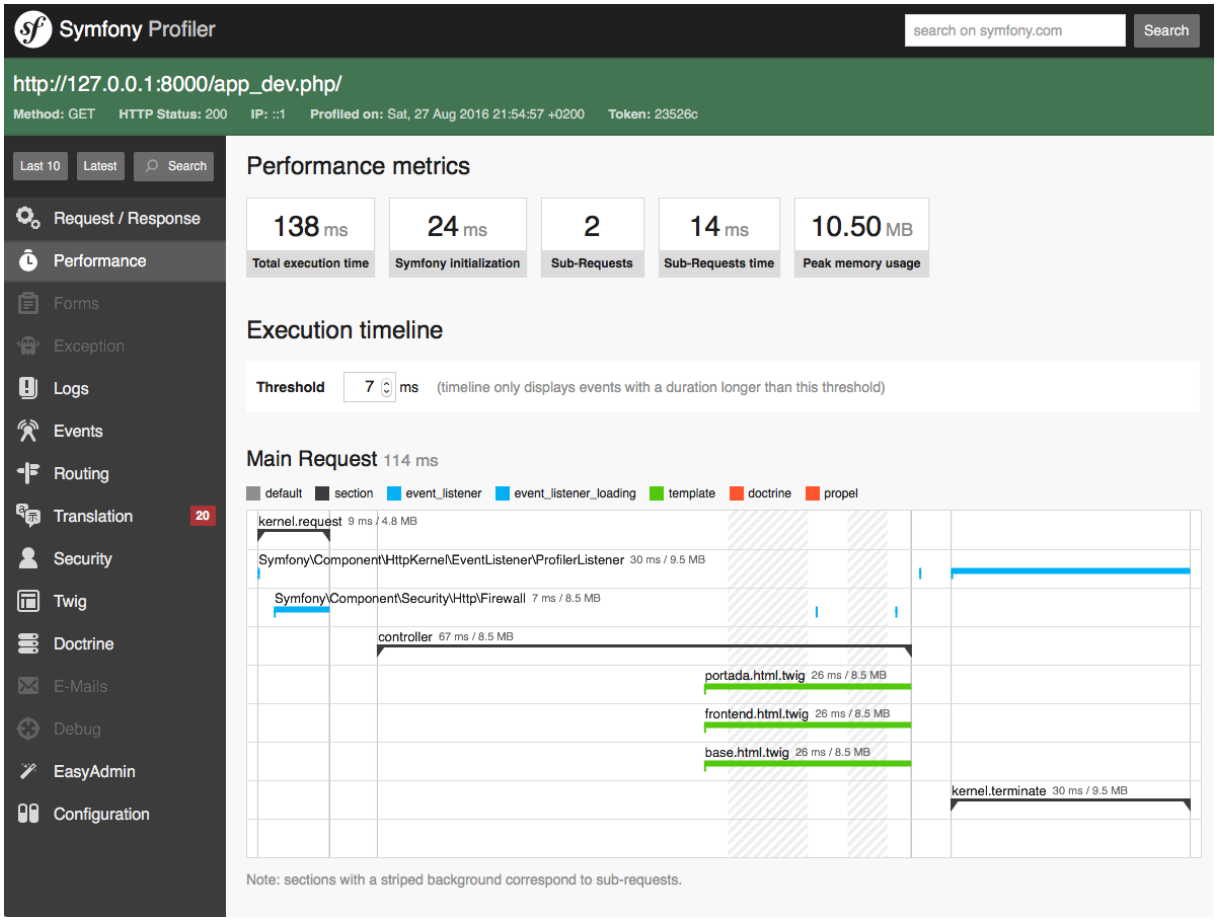

Figura 6.3 Panel principal del profiler de Symfony

NOTA Para poder acceder al *profiler* vía web sus rutas deben estar activadas. Por defecto Symfony sólo las carga en el archivo de enrutamiento del entorno de desarrollo:

```
# app/config/routing_dev.yml
_profiler:
    resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
    prefix:   /_profiler
```

6.6.3 Los archivos de log

La barra de depuración web y el *profiler* son las herramientas más cómodas para el programador porque siempre se encuentran a un click de distancia. Aún así, en ocasiones resulta más rápido consultar un archivo de log con todos los mensajes generados por una determinada petición.

Los archivos de log se guardan por defecto en el directorio `app/logs/`, que es el único, junto a `app/cache/`, en el que Symfony debe tener permisos de escritura. Cada entorno genera su propio archivo de log, llamado exactamente igual que el entorno (`dev.log`, `prod.log`, `test.log`, etc.)

La cantidad y detalle de los mensajes que se guardan en el archivo de log dependen del entorno de ejecución. En el entorno de desarrollo cada petición genera muchos mensajes, mientras que en producción no se escribe ningún mensaje a menos que la petición produzca algún error.

Aunque resulta lógico que en desarrollo se sacrifique el rendimiento para guardar toda la información y en producción se haga lo contrario, Symfony permite modificar este comportamiento completamente. Esto es posible mediante las opciones de configuración de *Monolog* (https://github.com/Seldaek/monolog) una librería de logs muy avanzada incluida en Symfony.

Monolog se configura mediante *handlers*. Cuando un mensaje se envía al sistema de logs, Monolog se lo pasa por orden a cada *handler* configurado. Los *handlers* pueden ignorar el mensaje, procesarlo y permitir que se siga procesando por el resto de *handlers* o procesarlo y detener el procesamiento. Para que un mensaje se entregue al *handler*, su nivel de gravedad debe ser igual o superior al indicado por la opción `level` o `action_level` del *handler*.

Los seis niveles de log definidos por Symfony son los siguientes:

- `debug`, es el nivel más detallado y genera decenas de mensajes para cada petición del usuario.
- `info`, mensajes muy poco importantes, pero con alguna información útil (al menos más útil que la de los mensajes de tipo `debug`).
- `warning`, no son errores, pero son situaciones excepcionales. Este es el nivel *normal* para una aplicación web.
- `error`, errores de cualquier tipo.
- `critical`, errores muy críticos, como por ejemplo un componente no disponible.
- `alert`, la situación es tan grave que se requiere una acción inmediata, como por ejemplo cuando todo el sitio web está caído.

En el entorno de desarrollo, Monolog se configura con un único *handler* que guarda todos los mensajes de tipo `debug` o superior en un archivo llamado `dev.log` y guardado en `app/logs/`:

```
# app/config/config_dev.yml
monolog:
    handlers:
        main:
            type:     stream
            path:     "%kernel.logs_dir%/%kernel.environment%.log"
```

```
            level:    debug
            channels: ["!event"]
```

Si crees que la cantidad de mensajes guardados en el archivo `app/logs/dev.log` es excesiva, sube el nivel mínimo de los mensajes a `info` o `warning`.

Por su parte, el entorno de producción configura un *handler* muy especial llamado `fingers_crossed` que a su vez hace uso de un archivo de log llamado `prod.log` en el directorio `app/logs/`:

```
# app/config/config_prod.yml
monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: error
            handler:      nested
        nested:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
```

Monolog incluye numerosos *handlers* que te permiten enviar los *logs* por email, guardarlos en bases de datos, enviarlos a servicios como Sentry y New Relic, mostrarlos en el navegador Chrome y Firefox, etc. Consulta la referencia completa de handlers (https://github.com/Seldaek/monolog/blob/master/doc/02-handlers-formatters-processors.md) en la documentación oficial de Monolog.

Encadenando varios *handlers* puedes realizar configuraciones muy avanzadas, como la siguiente que envía un email con todos los mensajes de log solamente cuando se produce un error:

```
# app/config/config_prod.yml
monolog:
    handlers:
        mail:
            type:         fingers_crossed
            action_level: critical
            handler:      buffered
        buffered:
            type:         buffer
            handler:      swift
        swift:
            type:         swift_mailer
            from_email:   robot@cupon.com
            to_email:     errores@cupon.com
            subject:      Se ha producido un error
            level:        debug
```

Además de los mensajes de log de Symfony, resulta muy sencillo añadir mensajes de log propios. Dentro de cualquier controlador, utiliza el siguiente código para añadir un mensaje de nivel `info`:

```
// ...
public function portadaAction()
{
    // ...

    $log = $this->get('logger');
    $log->addInfo('Generada la portada en '.$tiempo.' milisegundos');
}
```

Cada uno de los seis niveles dispone de su propio método: `addDebug()`, `addInfo()`, `addWarning()`, `addError()`, `addCritical()`, `addAlert()`. Para que tu código sea más conciso, cada nivel también tiene uno o más métodos abreviados:

- `debug`, método `debug()`
- `info`, método `info()` y método `notice()`
- `warning`, método `warn()`
- `error`, método `err()`
- `critical`, método `crit()`
- `alert`, método `alert()` y método `emerg()`

TRUCO Por defecto todos los archivos de log de Symfony están optimizados para "máquinas" en vez de para humanos. Si quieres, puedes optimizar el archivo `dev.log` para que la información se muestre de manera más amigable y tu productividad aumente. Para ello, instala la utilidad https://github.com/EasyCorp/easy-log-handler

6.6.4 La caché

Las tres herramientas anteriores son suficientes para encontrar la causa de cualquier error. No obstante, algunos errores especialmente problemáticos pueden solucionarse más fácilmente investigando el contenido de los archivos de caché generados por Symfony y sus componentes. Estos son los archivos más importantes que encontrarás en el directorio `app/cache/<nombre-del-entorno>/`:

- `appdevUrlGenerator.php`, esta es la clase PHP que se utiliza para generar las URL de la aplicación (por ejemplo mediante la función `path()` de Twig).
- `appdevUrlMatcher.php`, esta es la clase PHP utilizada para convertir las URL en rutas, es decir, para determinar qué controlador se utiliza para responder a la URL solicitada por el usuario.
- `appDevDebugProjectContainer.php`, esta clase (de varios miles de líneas de código) contiene todos los servicios y parámetros de configuración del contenedor de inyección de dependencias.

También puede ser útil echar un vistazo a los siguientes directorios:

- `doctrine/`, incluye una clase por cada relación entre dos entidades. Sin estas clases, no funciona el *lazy loading* de Doctrine.
- `twig/`, aunque la estructura de carpetas hace muy difícil la depuración, en este directorio se incluyen los archivos PHP que resultan de *compilar* las plantillas Twig. Lo mejor es que cada archivo PHP incluye el contenido original de la plantilla y su transformación, lo que facilita mucho su depuración.

En cualquier caso, recuerda que acceder al directorio `cache/` para investigar el contenido de estos archivos y directorios siempre debe ser el último recurso. Las herramientas anteriores son más que suficientes en casi todos los casos.

6.7 Refactorizando el Controlador

El *controlador* desarrollado en las secciones anteriores funciona bien, pero no es correcto para una aplicación web real. A continuación se refactoriza su código siguiendo las buenas prácticas recomendadas.

6.7.1 Parámetros de configuración

La búsqueda de *la oferta del día* se realiza indicando directamente el valor del atributo `id` de la ciudad por defecto. Lógicamente, este valor por defecto es mejor incluirlo en un archivo de configuración. Los parámetros de configuración globales del proyecto se definen bajo la clave `parameters` del archivo `app/config/config.yml` (debes añadir a mano la clave `parameters`):

```
# app/config/config.yml
parameters:
    app.ciudad_por_defecto: '1'
```

El nombre del parámetro puedes elegirlo libremente, pero se recomienda seguir una estructura de tipo `app.nombre_parametro` para distinguir mejor cuáles son los parámetros definidos por tu propia aplicación.

Para obtener el valor de un parámetro global, utiliza el método `getParameter()` dentro de la acción del controlador:

```
// src/AppBundle/Controller/DefaultController.php
// ...

/**
 * @Route("/", name="portada")
 */
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();
    $oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(array(
        'ciudad' => $this->getParameter('app.ciudad_por_defecto'),
        'fechaPublicacion' => new \DateTime('today')
```

```
        ));

        // ...
    }
```

6.7.2 Redirecciones

Según las especificaciones funcionales de la aplicación, todas las URL deben incluir el *slug* de la ciudad activa. Pero al mismo tiempo, todos los usuarios utilizan la URL `http://127.0.0.1:8000/` para acceder al sitio web. Una posible solución es buscar el nombre de la ciudad en la URL de la portada y si no se encuentra, redireccionar a una portada que incluya la ciudad por defecto de la aplicación.

Para ello, se va a actualizar el controlador `DefaultController` para asociar dos rutas diferentes a la misma acción. En primer lugar, modifica la URL de la ruta `portada` para que incluya el nombre de la ciudad:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", name="portada")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

Como la ruta `portada` ha cambiado, si pruebas ahora la aplicación verás numerosos mensajes de error. El motivo es que la ruta `portada` ahora requiere que se le pase un argumento.

Para evitar tener que actualizar todas las plantillas, puedes definir un valor por defecto para este argumento mediante la opción `defaults` (que admite el uso de parámetros de configuración si los encierras con `%`):

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
```

```
    /**
     * @Route(
     *     "/{ciudad}",
     *     "defaults" = { "ciudad" = "%app.ciudad_por_defecto%" },
     *     name = "portada"
     * )
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

A continuación, añade una nueva anotación `@Route` en el método `portadaAction()` debajo de la anotación existente:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", "defaults"={"ciudad"="%app.ciudad_por_defecto%"}, nam
e="portada")
     * @Route("/")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

No es necesario que definas un nombre a la nueva ruta porque nunca se va a utilizar para generar enlaces en las plantillas. La aplicación siempre enlaza con la portada de alguna ciudad, así que esta ruta solo se utiliza cuando el usuario accede directamente a esta URL desde su navegador.

El siguiente paso consiste en actualizar el código del método `portadaAction()`. Cuando el usuario accede a la URL `/`, el valor del argumento `$ciudad` será `null`. Si se da esta condición, el método debe redirigir a la portada de la ciudad por defecto. El código resultante sería:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", "defaults"={"ciudad"="%app.ciudad_por_defecto%"}, nam
e="portada")
     * @Route("/")
     */
    public function portadaAction($ciudad)
    {
        if (null === $ciudad) {
            return $this->redirectToRoute('portada', array(
                'ciudad' => $this->getParameter('app.ciudad_por_defecto')
            ));
        }

        // ...
    }
}
```

El método `redirectToRoute()` es uno de los atajos definidos en el controlador base (también define `redirect()` para redirigir a URLs). Su código es equivalente a lo siguiente:

```
// redireccionando con un atajo
return $this->redirectToRoute('portada', array('ciudad' => '...'));

// redireccionando con las utilidades de Symfony
use Symfony\Component\HttpFoundation\RedirectResponse;

return new RedirectResponse(
    $this->generateUrl('portada', array('ciudad' => '...'))
);
```

El constructor de la clase `RedirectResponse` toma como argumento la URL a la que se redirecciona. Al igual que sucede en las plantillas, las URL no se escriben a mano sino que se generan automáticamente con el método `generateUrl()` del controlador. Su primer argumento es el nombre de la ruta y el segundo argumento opcional es un array con los parámetros de la ruta.

Si ahora accedes a la portada a través de la URL `http://127.0.0.1:8000/app_dev.php`, serás redirigido automáticamente a la portada de la ciudad por defecto (`http://127.0.0.1:8000/app_dev.php/1` en este ejemplo).

6.7.3 Gestión de errores

El principal error que se puede producir al generar la portada del sitio es que no exista una *oferta del día* para la ciudad solicitada. En un proyecto web real es impensable que los administradores sean tan descuidados como para no definir la *oferta del día* en alguna ciudad. Aún así, el código debe estar preparado para esta circunstancia, así que añade lo siguiente en el controlador:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", "defaults"={"ciudad"="%app.ciudad_por_defecto%"}, nam
e="portada")
     * @Route("/")
     */
    public function portadaAction($ciudad)
    {
        // ...
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(array(
            'ciudad' => $this->getParameter('app.ciudad_por_defecto'),
            'fechaPublicacion' => new \DateTime('today')
        ));

        if (!$oferta) {
            throw $this->createNotFoundException(
                'No se ha encontrado la oferta del día en la ciudad seleccionad
a'
            );
        }

        // ...
    }
}
```

Cuando la consulta a la base de datos no devuelve ningún resultado, la variable `$oferta` vale `null`. Por tanto, se cumple la condición del `if()` y se lanza una excepción de tipo `NotFoundHttpException` creada con el método `createNotFoundException()`. Este método está disponible en todos los controladores que heredan de la clase `Controller`. Aunque puede parecer enrevesado, lanzar esta excepción es la forma de mostrar **las páginas de error 404** en las aplicaciones Symfony.

Después de todos estos cambios, el código del controlador resultante es el siguiente:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", "defaults"={"ciudad"="%app.ciudad_por_defecto%"}, nam
e="portada")
     * @Route("/")
     */
    public function portadaAction($ciudad)
    {
        // ...
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(array(
            'ciudad' => $this->getParameter('app.ciudad_por_defecto'),
            'fechaPublicacion' => new \DateTime('today')
        ));

        if (!$oferta) {
            throw $this->createNotFoundException(
                'No se ha encontrado la oferta del día en la ciudad seleccionad
a'
            );
        }

        return $this->render('portada.html.twig', array(
            'oferta' => $oferta
        ));
    }
}
```

6.7.4 Consultas a la base de datos

Una de las recomendaciones más importantes al desarrollar aplicaciones que siguen la arquitectura *Modelo - Vista - Controlador* consiste en no mezclar el código de cada parte. Esto significa por ejemplo que la vista no debe contener código relacionado con el controlador o que el controlador no debería incluir partes del modelo.

Así que el controlador actual no es del todo correcto, ya que realiza consultas a la base de datos, algo propio del modelo. Un problema añadido de las consultas dentro del controlador es que no se pueden reutilizar en otras acciones. Por tanto, la siguiente sección se centra en refactorizar la parte del modelo.

6.8 Refactorizando el Modelo

Las consultas de Doctrine se realizan a través de un objeto de tipo `EntityRepository` obtenido mediante el método `getRepository()` del *entity manager*:

```
$oferta = $em->getRepository('AppBundle:Oferta')->findOneBy(...);
```

Todas las entidades definidas en Doctrine disponen de su propio repositorio, con el que se pueden realizar las consultas `find()`, `findAll()`, `findBy()` y `findOneBy()`. El inconveniente de estos mé-

todos es que no permiten reutilizar búsquedas complejas en diferentes partes de la aplicación. Para solucionarlo, puedes crear tu propio repositorio para añadir nuevos métodos de búsqueda.

6.8.1 Creando un repositorio propio

Buscar la *oferta del día* en cada ciudad es una búsqueda que se repite en varias partes del código de la aplicación *Cupon*. Así que lo ideal sería poder utilizar el siguiente código en cualquier controlador:

```
$oferta = $em->getRepository('AppBundle:Oferta')->findOfertaDelDia($ciudad);
```

Como `findOfertaDelDia()` no es un método definido en Doctrine, es necesario crearlo en el repositorio propio de las entidades de tipo `Oferta`. Crea un archivo llamado `OfertaRepository.php` en el directorio `Repository/` del *bundle* `AppBundle` y añade el siguiente código:

```
// src/AppBundle/Repository/OfertaRepository.php
namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
{
    public function findOfertaDelDia($ciudad)
    {
    }
}
```

NOTA No es obligatorio que los métodos de un repositorio propio se llamen `findXXX()`. Sin embargo, se trata de una buena práctica recomendada para seguir la misma nomenclatura utilizada por Doctrine.

El método `findOfertaDelDia()` del ejemplo anterior no contiene ningún código porque no es posible hacer la consulta hasta que no se explique más adelante el lenguaje DQL de Doctrine.

Después, modifica el código de la entidad `Oferta` para indicarle que ahora dispone de un repositorio propio. Abre el archivo `Oferta.php` de la entidad y añade el parámetro `repositoryClass`:

```
// src/AppBundle/Entity/Oferta.php

/**
 * @ORM\Entity(repositoryClass="AppBundle\Repository\OfertaRepository")
 */
class Oferta
{
    // ...
}
```

Añadiendo el parámetro `repositoryClass` en la anotación `@ORM\Entity`, la entidad `Oferta` utilizará de ahora en adelante el repositorio propio creado anteriormente. Así que además de los

métodos `find()`, `findAll()`, `findBy()` y `findOneBy()`, ahora también podrás utilizar el método `findOfertaDelDia()` sobre cualquier entidad de tipo `Oferta`.

Crear un repositorio propio es algo tan sencillo y que requiere tan poco esfuerzo, que en las aplicaciones web reales se recomienda crear un repositorio para cada entidad. Así podrás separar correctamente el código que pertenece al modelo y podrás reutilizar consultas en toda la aplicación.

6.8.2 Realizando consultas DQL

Doctrine define un lenguaje propio llamado DQL para realizar consultas a la base de datos. Su sintaxis es similar al lenguaje SQL, pero su funcionamiento es radicalmente distinto. DQL realiza consultas sobre **objetos**, mientras que SQL realiza consultas sobre **tablas**. Si tratas de hacer consultas DQL pensando en el funcionamiento de SQL, no serás capaz de hacer ninguna consulta correcta.

El lenguaje DQL permite realizar consultas de tipo `SELECT`, `UPDATE` y `DELETE`. Las consultas de tipo `INSERT` no están permitidas, ya que toda la nueva información se debe crear a través del método `persist()` del *entity manager*. Observa la siguiente consulta que obtiene todas las ofertas cuyo precio sea inferior a 20 euros y ordena los resultados alfabéticamente:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('SELECT o FROM AppBundle:Oferta o WHERE o.precio <
20 ORDER BY o.nombre ASC');
$ofertas = $consulta->getResult();
```

Estas son las cuatro secciones en las que se divide la consulta DQL:

- `'FROM AppBundle:Oferta o'`, la sección `FROM` indica las entidades sobre las que se realiza la consulta. La entidad se especifica mediante la notación `NombreBundle:NombreEntidad`. Después de la entidad se incluye un identificador (`o` en este caso). Este identificador se utiliza en otras partes de la consulta, por lo que debe ser conciso e identificar bien a la entidad (normalmente se elige la inicial del nombre de la entidad). La consulta se puede hacer sobre varias entidades a la vez, separando todas ellas entre sí mediante una coma.
- `'SELECT o'`, la sección `SELECT` indica qué información devuelve la consulta. En este caso se quiere obtener el objeto `Oferta` con toda su información, por lo que simplemente se indica el identificador de la entidad utilizado en `FROM`. Si sólo nos hiciera falta el nombre de las ofertas, se indicaría `SELECT o.nombre`. Si se quiere obtener el nombre y el precio: `SELECT o.nombre, o.precio` y así sucesivamente.
- `'WHERE o.precio < 20'`, la sección opcional `WHERE` indica las condiciones que deben cumplir los objetos para ser incluidos en los resultados de la consulta. Las propiedades de la entidad se indican igual que en la sección `SELECT`, utilizando el identificador de la entidad y el nombre de la propiedad (`o.precio` en este caso).
- `'ORDER BY o.nombre ASC'`, la última sección también es opcional y añade a la consulta nuevas condiciones o modifica el orden el que se devuelven los resultados. Además de `ORDER BY` también están disponibles `HAVING` y `GROUP BY`.

Para limitar el número de resultados devueltos se utiliza el método `setMaxResults()`. Para paginar los resultados, también puedes hacer uso del método `setFirstResult()` que indica la posición del primer elemento devuelto:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('SELECT o FROM AppBundle:Oferta o WHERE o.precio <
20 ORDER BY o.nombre ASC');
$consulta->setMaxResults(20);
$consulta->setFirstResult(10);
$ofertas = $consulta->getResult();
```

6.8.2.1 Parámetros

El siguiente ejemplo muestra la primera versión de la consulta que busca la *oferta del día* en la ciudad indicada:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('
    SELECT o FROM AppBundle:Oferta o
     WHERE o.ciudad = 1
       AND o.fechaPublicacion = "201X-XX-XX 00:00:00"');
$oferta = $consulta->getResult();
```

El código anterior incluye la ciudad y la fecha en la propia consulta. Sin embargo, por motivos de seguridad se recomienda pasar los parámetros a la consulta mediante el método `setParameter()`, que filtra la información para asegurarse de que no incluye ningún contenido malicioso:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('
    SELECT o FROM AppBundle:Oferta o
     WHERE o.ciudad = :ciudad
       AND o.fechaPublicacion = :fecha');
$consulta->setParameter('ciudad', 1);
$consulta->setParameter('fecha', '201X-XX-XX 00:00:00');
$oferta = $consulta->getResult();
```

Cuando la consulta incluye muchos parámetros, puede ser más interesante utilizar el método `setParameters()`:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('
    SELECT o FROM AppBundle:Oferta o
     WHERE o.ciudad = :ciudad
       AND o.fechaPublicacion = :fecha');
$consulta->setParameters(array(
    'ciudad' => 1,
    'fecha'  => '201X-XX-XX 00:00:00'
));
$oferta = $consulta->getResult();
```

6.8.2.2 Uniendo entidades con JOIN

La consulta que obtiene la *oferta del día* de una ciudad utiliza como parámetro el valor del atributo `id` de la ciudad. Esto es lo más sencillo desde el punto de vista de las consultas de Doctrine, pero también es una mala solución. El valor del `id` de una ciudad puede variar por circunstancias como cambiar la base de datos o cargar más ciudades de prueba en la aplicación. Así que lo lógico es utilizar el nombre o el *slug* de la ciudad.

La siguiente consulta muestra cómo buscar la oferta del día en la ciudad de Barcelona:

```
$em = $this->getDoctrine()->getManager();
$consulta = $em->createQuery('
    SELECT o FROM AppBundle:Oferta o
      JOIN o.ciudad c
     WHERE c.slug = :ciudad
       AND o.fechaPublicacion = :fecha');
$consulta->setParameter('ciudad', 'barcelona');
$consulta->setParameter('fecha',  '201X-XX-XX 00:00:00');
$oferta = $consulta->getResult();
```

Como la entidad `Oferta` no guarda el nombre de la ciudad sino solamente su atributo `id`, es necesario ampliar la consulta para buscar no sólo entre las entidades de tipo `Oferta` sino también entre las entidades `Ciudad`. Esto se consigue mediante un `JOIN` entre dos entidades. La sintaxis y funcionamiento de los `JOIN` de DQL no se parece casi en nada a los `JOIN` de SQL.

Los `JOIN` de DQL se definen a través de las propiedades de una entidad. Así, `JOIN o.ciudad` significa que la consulta también se realiza sobre las entidades asociadas a la propiedad `ciudad` de la entidad `Oferta`. Si observas el código de la entidad `Oferta`, verás que la propiedad `ciudad` está asociada con las entidades de tipo `Ciudad`. Así que la consulta anterior se realiza sobre `AppBundle:Oferta` y `AppBundle:Ciudad`.

A la entidad asociada también se le asigna un identificador (`c` en este caso) para poder utilizarla en el resto de secciones de la consulta. Como identificador también se suele utilizar la inicial del nombre de la entidad asociada. Una vez creado el `JOIN`, ya puedes utilizar la nueva entidad en cualquier condición de la consulta, como por ejemplo `WHERE c.slug = 'barcelona'`.

El `JOIN` de la consulta anterior se denomina *JOIN normal*, ya que sólo se emplea para las condiciones de la búsqueda, pero no modifica los objetos devueltos como resultado. Doctrine define otro tipo de `JOIN` mucho más interesante para las aplicaciones web: los *fetch JOIN*. Observa las dos siguientes consultas DQL:

```
SELECT o FROM AppBundle:Oferta o JOIN o.ciudad c
 WHERE c.slug = '...' AND o.fechaPublicacion = '...'

SELECT o, c FROM AppBundle:Oferta o JOIN o.ciudad c
 WHERE c.slug = '...' AND o.fechaPublicacion = '...'
```

La primera consulta es un *JOIN normal* tal y como se ha explicado anteriormente. La segunda consulta es un *fetch JOIN*, cuya sintaxis es idéntica a los *JOIN normales*, pero su comportamiento es

muy diferente. En el segundo `JOIN`, los resultados de la búsqueda contienen los objetos de las ofertas con todos sus datos y también los objetos de las ciudades con todos sus datos.

La primera consulta activa el *lazy loading* cuando en la plantilla se utiliza la expresión `{{ oferta.ciudad.nombre }}`. Esto significa una consulta a la base de datos cada vez que se quiere mostrar el nombre de una ciudad. Sin embargo, en el segundo caso los resultados de la consulta ya contienen todos los datos de las ofertas y de las ciudades (debido al `SELECT o, c`) por lo que `{{ oferta.ciudad.nombre }}` no genera ninguna consulta adicional.

Realizar un *fetch JOIN* es tan sencillo como añadir más de una entidad en la sección `SELECT` de la consulta. Así que **para reducir drásticamente el número de consultas a la base de datos, utiliza *fetch JOIN* siempre que sea posible**. Gracias al lenguaje DQL, definir `JOIN` entre entidades es mucho más sencillo y rápido que crear `JOIN` entre tablas con SQL.

6.8.2.3 Consultas UPDATE y DELETE

La mayoría de consultas de las aplicaciones web son de tipo `SELECT`, pero con DQL también puedes actualizar (`UPDATE`) y borrar (`DELETE`) información.

La siguiente consulta muestra cómo subir un 10% el precio de todas las ofertas:

```
UPDATE AppBundle:Oferta o SET o.precio = o.precio * 1.10
```

Y la siguiente consulta muestra cómo borrar todos los usuarios cuyo email empiece por `anonimo`:

```
DELETE AppBundle:Usuario u WHERE u.email LIKE 'anonimo%'
```

NOTA El manual oficial de Doctrine incluye muchos más ejemplos de consultas DQL: http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html Dominar el lenguaje DQL te permitirá crear consultas más eficientes.

6.8.2.4 Realizando la consulta en el repositorio

En los repositorios de las entidades suele utilizarse siempre el lenguaje DQL para realizar las consultas. A continuación se muestra el código completo de la consulta `findOfertaDelDia()` creada en el repositorio de las ofertas:

```
// src/AppBundle/Repository/OfertaRepository.php
namespace AppBundle\Entity;

use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
{
    public function findOfertaDelDia($ciudad)
    {
        $fechaPublicacion = new \DateTime('today');
        $fechaPublicacion->setTime(23, 59, 59);
```

```
        $em = $this->getEntityManager();

        $dql = 'SELECT o, c, t
                  FROM AppBundle:Oferta o
                  JOIN o.ciudad c JOIN o.tienda t
                 WHERE o.revisada = true
                   AND o.fechaPublicacion < :fecha
                   AND c.slug = :ciudad
              ORDER BY o.fechaPublicacion DESC';

        $consulta = $em->createQuery($dql);
        $consulta->setParameter('fecha', $fechaPublicacion);
        $consulta->setParameter('ciudad', $ciudad);
        $consulta->setMaxResults(1);

        return $consulta->getSingleResult();
    }
}
```

Normalmente el resultado de una búsqueda incluye muchos objetos, por lo que se utiliza el método `getResults()`. En el caso de la búsqueda de *la oferta del día* sólo nos interesa obtener un resultado, por lo que se emplea el método `getSingleResult()` que no devuelve un array de objetos sino simplemente un objeto.

Un último cambio necesario para que la consulta anterior funcione es que la ciudad por defecto ya no se indica con su propiedad `id`, sino que se utiliza su *slug*. Como este valor se obtiene desde el archivo de configuración, sólo debes cambiar el valor de una opción de configuración:

```
# app/config/config.yml

# antes
parameters:
    app.ciudad_por_defecto: '1'

# ahora
parameters:
    app.ciudad_por_defecto: 'barcelona'
```

6.9 Refactorizando la Vista

Si observas la plantilla utilizada para crear la portada verás que contiene todo el código HTML necesario. Sin embargo, como la mayoría de páginas del *frontend* tienen la misma estructura interna, no es lógico repetir en todas ellas gran parte del código HTML.

En el ámbito de la programación, cuando varias clases comparten mucho código, se crea una clase con el código común y el resto de clases heredan de ella. El mismo razonamiento se aplica a las plantillas: todo el código HTML y Twig común se guarda en una plantilla base de la que heredan

el resto de plantillas. Esta plantilla base suele denominarse *layout* y su uso es imprescindible en cualquier aplicación web real.

6.9.1 Creando el *layout* del *frontend*

El primer paso para definir el *layout* consiste en separar todo el código HTML y Twig que sea común a todas las páginas del *frontend*:

```
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>## TITULO DE LA PÁGINA ## | Cupon</title>
</head>

<body id="## ID DE LA PÁGINA ##"><div id="contenedor">
    <header>
        <h1><a href="{{ path('portada') }}">CUPON</a></h1>
        <nav>
            <ul>
                <li><a href="{{ path('portada') }}">Oferta del día</a></li>
                <li><a href="#">Ofertas recientes</a></li>
                <li><a href="#">Mis ofertas</a></li>
            </ul>
        </nav>
    </header>

    <article>
        ## CONTENIDO PRINCIPAL DE LA PÁGINA ##
    </article>

    <aside>
        ## CONTENIDO SECUNDARIO DE LA PÁGINA ##
    </aside>

    <footer>
        &copy; {{ 'now'|date('Y') }} - Cupon
        <a href="{{ path('pagina', { nombrePagina: 'ayuda' }) }}">Ayuda</a>
        <a href="{{ path('contacto') }}">Contacto</a>
        <a href="{{ path('pagina', { nombrePagina: 'privacidad' }) }}">Privacida
d</a>
        <a href="{{ path('pagina', { nombrePagina: 'sobre-nosotros' }) }}">Sobr
e nosotros</a>
    </footer>
</div></body>

</html>
```

Como se trata de una plantilla utilizada como base de muchas otras plantillas, se recomienda guardarla directamente en el directorio `app/Resources/views/`. El nombre del *layout* puedes elegirlo libremente, pero como esta aplicación va a requerir más de un *layout*, se utiliza el nombre `frontend.html.twig` para distinguirlo de los demás:

```
{# app/Resources/views/frontend.html.twig #}
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>## TITULO DE LA PÁGINA ## | Cupon</title>
</head>

<body id="## ID DE LA PÁGINA ##"><div id="contenedor">
{# ... #}
```

Para que la herencia de plantillas funcione correctamente, la plantilla base define las partes que pueden modificar las plantillas que heredan de ella. Para ello, se añaden elementos `{% block %}` siguiendo la sintaxis de la herencia de plantillas de Twig (página 472):

```
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}{% endblock %} | Cupon</title>
</head>

<body id="{% block id '' %}"><div id="contenedor">
    <header>
        {# ... #}
    </header>

    <article>
        {% block article %}{% endblock %}
    </article>

    <aside>
        {% block aside %}{% endblock %}
    </aside>

    <footer>
        {# ... #}
    </footer>
</div></body>

</html>
```

Por último, modifica la plantilla de la portada para que herede de este nuevo *layout*. Para ello, primero se añade la instrucción `{% extends %}` indicando la localización de la plantilla base:

```
{# app/Resources/views/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{# ... #}
```

Ahora ya sólo falta *rellenar los huecos* definidos por el *layout*, es decir, añadir contenidos a los bloques de la plantilla base:

```
{# app/Resources/views/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Cupon, cada día ofertas increíbles en tu
ciudad con descuentos de hasta el 90%{% endblock %}
{% block id 'portada' %}

{% block article %}
    <section class="descripcion">
        <h1><a href="#">{{ oferta.nombre }}</a></h1>
        {{ oferta.descripcion }}
        <a class="boton" href="#">Comprar</a>
    </section>

    <section class="galeria">
        <img alt="Fotografía de la oferta" src="{{ asset('uploads/images/' ~ ofe
rta.rutaFoto) }}">

        <p class="precio">{{ oferta.precio }} &euro; <span>{{ oferta.descuento
}}</span></p>

        <p><strong>Condiciones:</strong> {{ oferta.condiciones }}</p>
    </section>

    <section class="estado">
        <div class="tiempo">
            <strong>Faltan</strong>: {{ oferta.fechaExpiracion|date }}
        </div>

        <div class="compras">
            <strong>Compras</strong>: {{ oferta.compras }}
        </div>

        <div class="faltan">
            {% set faltan = oferta.umbral - oferta.compras %}
            {% if faltan > 0 %}
                Faltan <strong>{{ faltan }} compras</strong> <br/> para activar
la oferta
```

```
            {% else %}
            <strong>Oferta activada</strong> por superar las <strong>{{ oferta.u
mbral }}</strong> compras necesarias
            {% endif %}
        </div>
    </section>

    <section class="direccion">
        <h2>Disfruta de la oferta en</h2>
        <p>
            <a href="#">{{ oferta.tienda.nombre }}</a>
            {{ oferta.tienda.direccion }}
        </p>
    </section>

    <section class="tienda">
        <h2>Sobre la tienda</h2>
        {{ oferta.tienda.descripcion }}
    </section>
{% endblock %}

{% block aside %}
    ## FORMULARIO DE LOGIN ##

    <section id="nosotros">
        <h2>Sobre nosotros</h2>
        <p>Lorem ipsum dolor sit amet...</p>
    </section>
{% endblock %}
```

La refactorización realizada en esta sección es lo que se denomina *herencia de dos niveles*. Las aplicaciones web reales suelen construirse en base a una *herencia de tres niveles*, que es una generalización de este ejemplo y que se explica en el siguiente capítulo.

6.9.1.1 Estableciendo la ciudad seleccionada

Las dos rutas de la portada creadas en las secciones anteriores (una con la ciudad en su URL y otra sin nada) hacen necesario un último cambio en el *layout* del *frontend*. La funcionalidad deseada es que la portada apunte siempre a la ciudad del usuario *logueado*. Y si se trata de un usuario anónimo, que apunte a la portada de la ciudad por defecto.

Cuando más adelante se explique cómo registrar y *loguear* usuarios, aprenderás a redirigir al usuario a la portada de su ciudad asociada. Por el momento, simplemente vamos a suponer que *algo* hace que la petición del usuario incluya el *slug* de su ciudad. Como también se debe considerar a los usuarios anónimos, lo mejor es dejar que sea la plantilla la que decida a qué portada apuntar en cada momento. Para ello, añade lo siguiente en la plantilla `frontend.html.twig`:

```
{% set ciudadSeleccionada = app.request.attributes.get('ciudad')|default(ciuda
d_por_defecto) %}
```

```
{# ... #}

<header>
    <h1><a href="{{ path('portada', { ciudad: ciudadSeleccionada }) }}">CUPO
N</a></h1>
    <nav>
    {# ... #}
```

Si la petición del usuario contiene un parámetro llamado `ciudad`, se utiliza su valor como ciudad activa del sitio web. Si no existe ese parámetro, se utiliza directamente la ciudad por defecto.

La expresión `app.request` permite acceder al objeto `Request` desde la plantilla. Como se explicará con detalle más adelante en este mismo capítulo, este objeto contiene toda la información relacionada con la petición del usuario y muchos métodos útiles para extraer esa información.

6.10 Funcionamiento interno de Symfony

Una vez desarrollada la primera página dinámica completa con acceso a la base de datos, ya es posible explicar con todo detalle el funcionamiento interno de Symfony. El siguiente esquema muestra el proceso completo desde que el usuario solicita una URL hasta que el servidor web le entrega como respuesta una página:

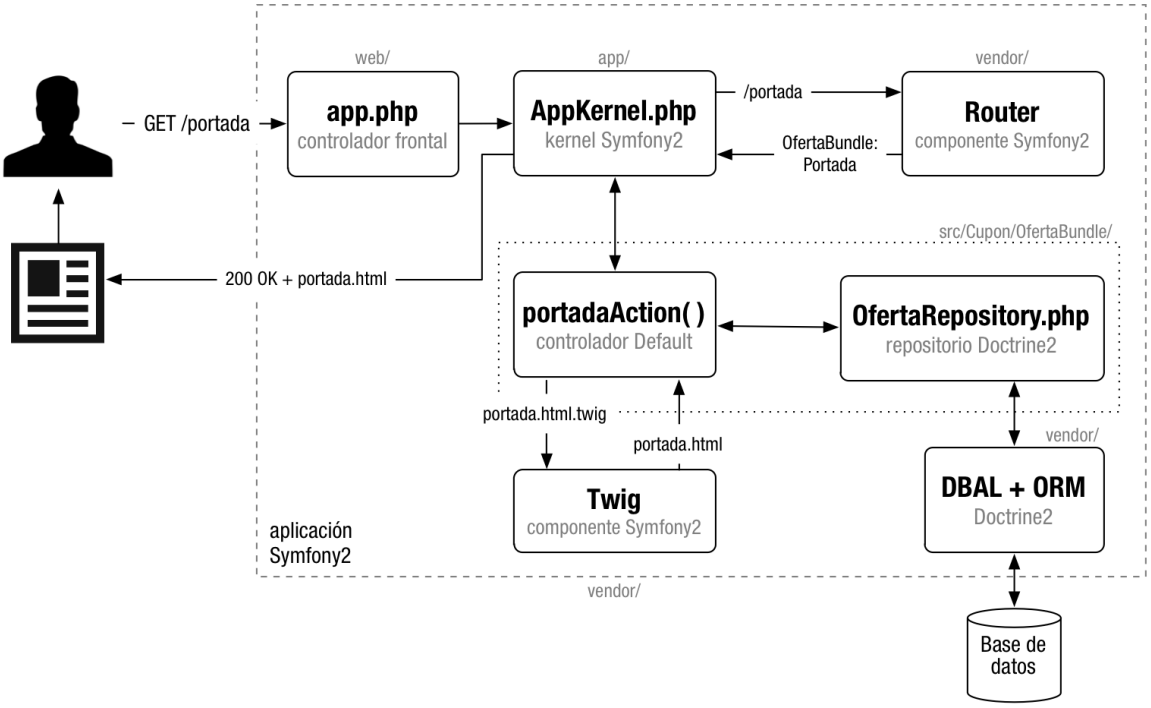

Figura 6.4 Esquema del funcionamiento interno de Symfony

El usuario solicita la portada del sitio a través de la URL `http://127.0.0.1:8000/`. Las reglas del archivo `.htaccess` del directorio `web/` de la aplicación redirigen la petición al script `app.php`. Este archivo se denomina **controlador frontal** y es el único punto de entrada a la aplicación.

El controlador frontal crea el *kernel* de la aplicación mediante una instancia de la clase `app/AppKernel.php` a la que se le pasa el valor `prod` como nombre del **entorno de ejecución**. El *kernel* registra todos los *bundles* de la aplicación y carga el archivo de configuración `app/config/config_prod.yml`. A su vez, este archivo importa el archivo `app/config/config.yml` que carga las rutas del archivo `app/config/routing.yml`.

A continuación, el controlador frontal crea un objeto de tipo `Request` rellenándolo con la información de la petición del usuario y se lo pasa al *kernel*. El procesado de la petición comienza notificando algunos eventos y continúa preguntando al componente de **enrutamiento** qué controlador se debe ejecutar para atender la petición del usuario.

Una vez localizado el **controlador** y la acción que corresponden a la URL solicitada, se ejecuta su código. Si es necesario, el controlador hace uso del **modelo** para buscar información en la base de datos. Normalmente esta búsqueda se realiza mediante un **repositorio** propio creado para las entidades de Doctrine.

Si el controlador devuelve un objeto de tipo `Response`, el *kernel* lo transforma en la página que se devuelve al usuario. Si el controlador devuelve el nombre de una plantilla, el *kernel* hace uso de la **vista** para renderizar la plantilla según el motor de plantillas configurado en la aplicación (Twig o PHP). La ejecución finaliza devolviendo al usuario la página creada a partir de la plantilla.

6.11 El objeto Request

Symfony encapsula toda la información de la petición del usuario en un objeto de tipo `Request`. Así se facilita y centraliza el acceso a toda la información de las variables globales de PHP como `$_GET`, `$_REQUEST`, `$_SERVER`, `$_FILES` y `$_COOKIE`.

Dentro de un controlador, el objeto de la petición se obtiene añadiendo un argumento de tipo `Request` en la acción del controlador:

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function portadaAction(Request $request)
    {
        // aquí ya está disponible la variable $request
        // con toda la información de la petición del usuario
    }
}
```

Cuando una acción tiene un argumento de tipo `Symfony\Component\HttpFoundation\Request`, Symfony lo interpreta como una solicitud para inyectar automáticamente el objeto que representa a la petición del usuario. Como solamente se tiene en cuenta el tipo de la variable, no importa ni su nombre ni la posición en la que aparezca ese argumento:

```
public function portadaAction($idOferta, $numArticulos, Request $request)
{
    // la variable que representa a la petición del usuario
    // sigue estando disponible en la variable $request
}
```

Una vez obtenido el objeto de tipo `Request`, ya puedes hacer uso de sus métodos y propiedades para acceder a la misma información que obtendrías con las variables globales de PHP, pero con un acceso más cómodo:

Propiedad de `Request`	Tipo de variable	Equivalente de PHP
`$query`	`ParameterBag`	variable global `$_GET`
`$request`	`ParameterBag`	variable global `$_POST`
`$attributes`	`ParameterBag`	parámetros incluidos en `$_SERVER['PATH_INFO']`
`$cookies`	`ParameterBag`	variable global `$_COOKIE`
`$files`	`FileBag`	variable global `$_FILES`
`$server`	`ServerBag`	variable global `$_SERVER`
`$headers`	`HeaderBag`	cabeceras incluidas en la variable global `$_SERVER`

Las variables de tipo `*Bag` son contenedores que almacenan pares clave/valor y que incluyen varios métodos útiles:

Método de `ParameterBag`	Explicación
`get($clave, $defecto = null)`	Devuelve el valor del parámetro cuya clave se indica como primer parámetro. Se puede indicar un segundo parámetro que será el valor devuelto cuando el parámetro no exista
`has($clave)`	Devuelve `true` si existe el parámetro cuya clave se indica como argumento. Devuelve `false` en cualquier otro caso
`set($clave, $valor)`	Modifica o añade el parámetro indicado mediante su clave y valor
`add($array)`	Añade los parámetros pasados como array, reemplazando los valores de los parámetros existentes
`all()`	Devuelve un array con todos los parámetros
`keys()`	Devuelve un array que sólo contiene las claves de los parámetros

Aunque la tabla anterior solamente muestra los métodos de `ParameterBag`, las diferencias con los otros tipos de variables `*Bag` son mínimas. A continuación se muestran algunos ejemplos útiles del uso de las propiedades públicas de `Request`:

```
// Obtener el valor del parámetro GET llamado 'ciudad'
// $ciudad valdrá 'null' si no existe el parámetro 'ciudad'
```

```
$ciudad = $request->query->get('ciudad');

// Mismo ejemplo, pero asignando un valor por defecto por si
// no existe el parámetro de tipo GET llamado 'ciudad'
$ciudad = $request->query->get('ciudad', 'paris');

// Obtener el valor del parámetro POST llamado 'ciudad'
// $ciudad valdrá 'null' si no existe el parámetro 'ciudad'
$ciudad = $request->request->get('ciudad');

// Mismo ejemplo, pero asignando un valor por defecto por si
// no existe el parámetro de tipo POST llamado 'ciudad'
$ciudad = $request->request->get('ciudad', 'paris');

// Saber qué navegador utiliza el usuario mediante la cabecera HTTP_USER_AGENT
$navegador = $request->server->get('HTTP_USER_AGENT');

// Mismo ejemplo, pero más fácil directamente a través de las cabeceras
$navegador = $request->headers->get('user-agent');

// Obtener el nombre de todas las cabeceras enviadas
$cabeceras = $request->headers->keys();

// Saber si se ha enviado una cookie de sesión
$hayCookieSesion = $request->cookies->has('PHPSESSID');
```

La clase `Request` también incluye un método llamado `get()` que obtiene el valor del parámetro cuyo nombre se indica como argumento. Este parámetro se busca, por este orden, en las variables `$_GET`, `$_SERVER['PATH_INFO']`, y `$_POST`:

```
$ciudad = $request->query->get('ciudad');
$ciudad = $request->get('ciudad');
```

Aunque el método `get()` es un atajo muy útil, se desaconseja su uso a menos que sea completamente imprescindible, ya que es mucho más lento que acceder a las propiedades `$query`, `$request`, etc.

Por otra parte, la clase `Request` también incluye varios métodos útiles relacionados con la URI. En los siguientes ejemplos se supone que el usuario ha solicitado la página `http://127.0.0.1:8000/app_dev.php/sitio/ayuda?parametro=valor`:

Método	Explicación
`getMethod()`	Devuelve el nombre en mayúsculas del método utilizado para la petición (`GET`, `POST`, `PUT`, `DELETE`). Ejemplo: `GET`
`getRequestUri()`	Devuelve la URI de la petición. Ejemplo: `/app_dev.php/ayuda?parametro=valor`

Método	Explicación
`getUri()`	Devuelve la URI *normalizada* (con todas sus partes). Ejemplo: `http://127.0.0.1:8000/app_dev.php/ayuda?parametro=valor`
`getScheme()`	Devuelve el esquema utilizado (`http` o `https`)
`getHost()`	Devuelve el *host* de la URI. Ejemplo: `cupon.local`
`getHttpHost()`	Igual que `getHost()`, pero también incluye el puerto si no es `80` o `443`. Ejemplo: `cupon.local`
`getPort()`	Devuelve el puerto al que se realiza la petición. Ejemplo: `80`
`getQueryString()`	Devuelve la *query string* normalizada de la URI (los parámetros se ordenan alfabéticamente y se *escapan* los caracteres problemáticos). Ejemplo: `parametro=valor`
`getScriptName()`	Devuelve el nombre del *script* PHP que se ejecuta. Ejemplo: `/app_dev.php` (incluye la barra `/` por delante)
`getPathInfo()`	Devuelve la parte de la ruta en relación al script que se ejecuta. Ejemplo: `/ayuda` (siempre incluye la barra `/` por delante)
`getBasePath()`	Devuelve la ruta base desde la que se está ejecutando la petición. La URL del ejemplo utilizado devuelve una cadena vacía. Si la URL es `http://127.0.0.1:8000/frontend/app_dev.php/ayuda`, se devuelve `/frontend` (siempre incluye la barra `/` por delante)
`getBaseUrl()`	Igual que `getBasePath()` pero también incluye el nombre del script que se ejecuta. Ejemplo: `/app_dev.php` (nunca se incluye la barra `/` al final)

Además de los métodos anteriores, la clase `Request` define otros atajos útiles para obtener información de la petición:

Método	Explicación
`getClientIp()`	Devuelve la dirección IP del usuario que ha realizado la petición. Si el usuario se encuentra detrás de un *proxy*, busca su dirección en los parámetros `HTTP_CLIENT_IP` o `HTTP_X_FORWARDED_FOR` en vez de `REMOTE_ADDR`
`getLanguages()`	Devuelve un array con el código de los idiomas preferidos por el usuario y ordenados por importancia. Ejemplo: `array('en_US', 'en', 'es')`
`getPreferredLanguage()`	Devuelve el idioma preferido por el usuario, que coincide con el primer idioma de `getLanguages()`. Puedes pasar como argumento un array con el código de los idiomas que soporta la aplicación. En este caso se devuelve el primer idioma que coincida en el array pasado y en el array devuelto por `getLanguages()`. Si no se producen coincidencias, se devuelve el primer idioma del array pasado.

Método	Explicación
`isXmlHttpRequest()`	Devuelve un valor *booleano* indicando si la petición es AJAX o no. Sólo se asegura un correcto funcionamiento con las peticiones realizadas por las librerías jQuery, Prototype y Mootools
`isSecure()`	Devuelve un valor *booleano* que indica si la petición es segura (HTTPS)

Utilizar la clase `Request` en vez de las variables globales de PHP puede parecer innecesario. En realidad, aunque los métodos tienen una apariencia y funcionamiento sencillo, internamente realizan comprobaciones largas y complejas. Observa por ejemplo el código fuente del método `isSecure()`, que realiza tres comprobaciones diferentes para decidir si una petición es segura o no:

```
public function isSecure()
{
    return (
        (strtolower($this->server->get('HTTPS')) == 'on' || $this->server->ge
t('HTTPS') == 1)
        ||
        (self::$trustProxy && strtolower($this->headers->get('SSL_HTTPS')) == 'o
n' || $this->headers->get('SSL_HTTPS') == 1)
        ||
        (self::$trustProxy && strtolower($this->headers->get('X_FORWARDED_PROT
O')) == 'https')
    );
}
```

La mayoría de peticiones a las aplicaciones web son de tipo HTML, pero en ocasiones se emplean formatos diferentes (XML, JSON, etc.) Por eso la clase `Request` dispone de dos métodos útiles relacionados con los formatos:

Método	Explicación
`getRequestFormat()`	Devuelve el formato de la petición en minúsculas. Ejemplo: `html`
`getMimeType()`	Devuelve el tipo MIME asociado al formato de la petición. Ejemplo: `text/html`

La clase de la petición también permite comprobar si existe una sesión iniciada (`hasSession()`) e incluso incluye un acceso directo al objeto de la sesión del usuario (`getSession()`).

Para conocer el resto de métodos y propiedades de la clase `Request`, puedes consultar la API de Symfony (http://api.symfony.com/2.8/) .

6.12 El objeto Response

Symfony encapsula en un objeto de tipo `Response` toda la información necesaria para generar la página que se entrega al usuario como resultado de su petición. La mayoría de controladores finalizan su ejecución invocando el método `render()` para generar una página a partir de la plantilla indicada:

```
class DefaultController extends Controller
{
    public function portadaAction()
    {
        // ...

        return $this->render('portada.html.twig');
    }
}
```

Este método `render()` en realidad es un atajo definido en la clase `Controller` de la que heredan por defecto los controladores de Symfony. Utilizando el sistema de plantillas definido en la aplicación, el método `render()` crea un objeto de tipo `Response` que Symfony convierte después en la página HTML que realmente se devuelve al usuario.

Estos atajos y métodos son imprescindibles para el desarrollo ágil de aplicaciones, pero si lo requieres también puedes utilizar el objeto `Response` al más bajo nivel:

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

class DefaultController extends Controller
{
    public function portadaAction()
    {
        $html = <<<HTML
<html>
  <head> <title>Portada del sitio</title> </head>
  <body> <h1>Portada</h1> </body>
</html>
HTML;

        $respuesta = new Response();

        $respuesta->headers->set('Content-Type', 'text/html');
        $respuesta->setStatusCode(200);
        $respuesta->setContent($html);

        return $respuesta;
    }
}
```

El código anterior también se puede definir de la siguiente manera:

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

class DefaultController extends Controller
{
    public function portadaAction()
    {
```

```
        $html = <<<HTML
<html>
  <head> <title>Portada del sitio</title> </head>
  <body> <h1>Portada</h1> </body>
</html>
HTML;

        return new Response($html, 200, array(
            'Content-Type' => 'text/html'
        ));
    }
}
```

La única propiedad pública de `Response` es `headers`, que permite consultar, añadir, modificar o borrar cualquier cabecera de la respuesta, además de las *cookies*. Los métodos más útiles son `(g|s)etContent()` para obtener/establecer el contenido de la respuesta y `(g|s)etStatusCode()` para obtener/establecer el código de estado de la respuesta.

El resto de métodos incluidos en la clase `Response` están relacionados con la caché de HTTP y se explican más adelante en el capítulo 20 (página 423): `(g|s)etAge()`, `(g|s)etEtag()`, `(g|s)etExpires()`, `(g|s)etLastModified()`, `(g|s)etMaxAge()`, `(g|s)etTtl()`, `(g|s)etVary()`, `isCacheable()`, `isFresh()` y `mustRevalidate()`.

Por último, si utilizas el componente `HttpFoundation` fuera de una aplicación Symfony, la clase `Response` incluye un método llamado `send()` que devuelve todas las cabeceras y el contenido que forman la respuesta. Si lo necesitas en alguna ocasión, también dispones del método `sendHeaders()` para devolver sólo las cabeceras y `sendContent()` para devolver el contenido.

Para conocer el resto de métodos y propiedades de la clase `Response`, puedes consultar la API de Symfony (http://api.symfony.com/2.8/) .

Esta página se ha dejado vacía a propósito

CAPÍTULO 7

Completando el frontend

7.1 Herencia de plantillas a tres niveles

En el capítulo anterior se utilizó la *herencia a dos niveles* para crear la plantilla de la portada del sitio. En las aplicaciones web reales suele ser habitual utilizar una *herencia a tres niveles* para reutilizar el máximo código posible.

En el caso de la aplicación *Cupon*, se definen los tres siguientes niveles de plantillas:

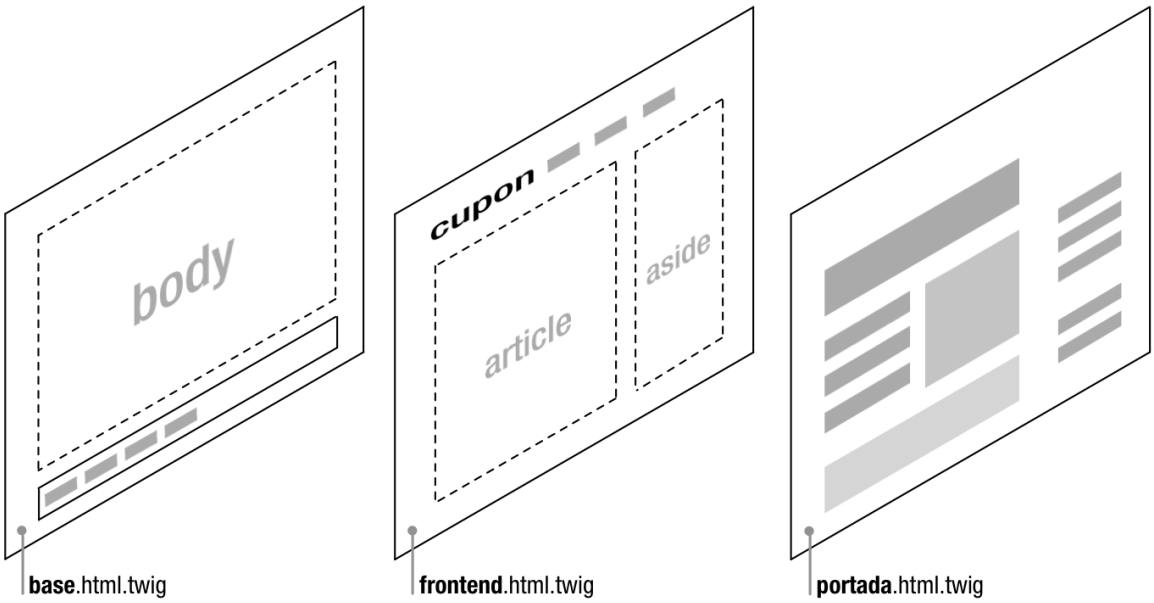

Figura 7.1 Herencia de plantillas a tres niveles utilizada para generar la portada

- **Primer nivel**: una sola plantilla base de toda la aplicación. Se llama `base.html.twig` e incluye solamente los elementos que se repiten en todas las páginas: `<html>`, `<head>`, `<body>`, `<title>`, `<footer>` y los enlaces a archivos CSS y JavaScript.
- **Segundo nivel**: una plantilla para cada sección del sitio web (`frontend.html.twig`, `extranet.html.twig` y `backend.html.twig`). Estas plantillas definen la estructura de los contenidos (columnas de información, barras laterales, etc.) y enlazan los archivos CSS y JavaScript adecuados.

- **Tercer nivel**: plantilla específica para cada página de la aplicación: `portada.html.twig` para crear la portada del *frontend*, `oferta.html.twig` para crear la página de detalle de una oferta, etc. Incluye todos los contenidos específicos de cada página.

La estructura de las plantillas con herencia a dos niveles es la siguiente:

```
{# app/Resources/views/frontend.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}{% endblock %} | Cupon</title>
</head>
<body id="{% block id '' %}"><div id="contenedor">
    <header> ... </header>
    <article>
        {% block article %}{% endblock %}
    </article>
    <aside>
        {% block aside %}{% endblock %}
    </aside>
    <footer> ... </footer>
</div></body>
</html>

{# app/Resources/views/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %} ... {% endblock %}
{% block id 'portada' %}

{% block article %} ... {% endblock %}
{% block aside %} ... {% endblock %}
```

Para definir la herencia a tres niveles, crea en primer lugar la plantilla base de la aplicación:

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}{% endblock %} | Cupon</title>
    {% block stylesheets %}{% endblock %}
</head>

<body id="{% block id '' %}"><div id="contenedor">
    {% block body %}{% endblock %}

    <footer>
```

```
        &copy; {{ 'now' | date('Y') }} - Cupon
        <a href="{{ path('pagina', { pagina: 'ayuda' }) }}">Ayuda</a>
        <a href="{{ path('pagina', { pagina: 'contacto' }) }}">Contacto</a>
        <a href="{{ path('pagina', { pagina: 'privacidad' }) }}">Privacidad</a>
        <a href="{{ path('pagina', { pagina: 'sobre-nosotros' }) }}">Sobre nosot
ros</a>
    </footer>

    {% block javascripts %}{% endblock %}
</div></body>
</html>
```

Los dos únicos bloques de la antigua plantilla `frontend.html.twig` que se incluyen en la nueva `base.html.twig` son `title` e `id`. Al valor del bloque `title` se le concatena al final el texto `| Cupon` y al bloque `id` se le asigna un valor por defecto vacío para aquellas plantillas que no lo definan.

Los dos primeros bloques nuevos de la plantilla son `stylesheets` y `javascripts`. En su interior se incluirán, respectivamente, los enlaces a los archivos CSS y JavaScript definidos por las plantillas de segundo y tercer nivel. El nombre utilizado para estos bloques es el habitual de las aplicaciones Symfony, aunque puedes elegirlo libremente. El bloque `javascripts` se incluye al final de la página para mejorar el tiempo de carga de las páginas.

Por último, esta plantilla define un bloque llamado `body` que incluye todos los contenidos de la página salvo el `<footer>` o pie de página. Esta solución es la más habitual porque así las páginas podrán utilizar cualquier estructura de contenidos (sin columnas, a dos columnas, etc.) El nombre `body` también es habitual en la mayoría de aplicaciones Symfony, pero puedes elegir cualquier otro nombre que prefieras.

Después de definir esta plantilla base, actualiza la antigua plantilla `frontend.html.twig`:

```
{# app/Resources/views/frontend.html.twig #}
{% extends 'base.html.twig' %}

{% block stylesheets %}{% endblock %}
{% block javascripts %}{% endblock %}

{% block body %}
<header>
    <h1><a href="{{ path('portada') }}">CUPON</a></h1>
    <nav>
        <ul>
            <li><a href="{{ path('portada') }}">Oferta del día</a></li>
            <li><a href="#">Ofertas recientes</a></li>
            <li><a href="#">Mis ofertas</a></li>
        </ul>
    </nav>
</header>
```

```
<article>
    {% block article %}{% endblock %}
</article>

<aside>
    {% block aside %}{% endblock %}
</aside>
{% endblock %}
```

Este segundo nivel hereda de la plantilla base (`{% extends 'base.html.twig'%}`) y añade en primer lugar los archivos CSS y JavaScript que cargan las páginas del *frontend* (su contenido por el momento permanece vacío hasta que no se explique en la próxima sección).

Después, se añade el contenido del bloque `body`. Como se trata del segundo nivel de herencia, sólo se incluyen los elementos comunes a todas las páginas del *frontend*: el menú principal de navegación y los elementos `<article>` y `<aside>` en los que cada página incluye sus contenidos específicos.

Observa cómo en esta plantilla no se añade ni el bloque `title` ni el bloque `id`. Cuando una plantilla hereda de otra, no es obligatorio que defina el contenido de todos los bloques de la plantilla superior. Así que estos dos bloques se definirán en las plantillas de tercer nivel.

En el tercer nivel cada plantilla es diferente, pero a continuación se muestra como ejemplo la nueva estructura de la plantilla de la portada desarrollada en el capítulo anterior:

```
{# app/Resources/views/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %} ... {% endblock %}
{% block id 'portada' %}

{% block article %} ... {% endblock %}

{% block aside %} ... {% endblock %}
```

A pesar de su sencillez, este ejemplo muestra claramente el poder de la herencia de plantillas de Twig: la plantilla añade el contenido de cuatro bloques, pero dos de ellos (`title` e `id`) se definen en la plantilla de primer nivel, mientras que los otros dos (`article` y `aside`) lo hacen en la plantilla de segundo nivel.

Después de todos estos cambios, la aplicación ya está preparada para crear correctamente cualquier plantilla, ya sea del *frontend*, de la *extranet* o del *backend*.

7.2 Assets web (hojas de estilo y archivos JavaScript)

El conjunto de hojas de estilo, archivos JavaScript e imágenes del sitio web, se denominan *web assets* o *archivos web*. Las aplicaciones web modernas utilizan decenas de estos archivos para crear sus interfaces.

En las primeras versiones de Symfony, se recomendaba el uso del proyecto Assetic (https://github.com/kriswallsmith/assetic) mediante el *bundle* AsseticBundle (https://github.com/symfony/assetic-bundle) para gestionar estos *assets*. Sin embargo, en los últimos años se han desarrollado numerosas herramientas JavaScript para gestionar mejor los *assets*.

Por ese motivo, a partir de Symfony 2.8 ya no se incluye por defecto el *bundle* AsseticBundle. A pesar de que todavía puedes usar Assetic en tus aplicaciones web, tal y como se explica en este artículo (http://symfony.com/doc/current/assetic/php.html) la buena práctica recomendada consiste en gestionar los *assets* mediante herramientas como Bower (https://bower.io/) , Gulp (http://gulpjs.com) y webpack (https://webpack.github.io) .

La aplicación que se está desarrollando define cuatro hojas de estilos:

- `normalizar.css`, neutraliza los estilos por defecto de los diferentes navegadores y se incluye en todas las páginas.
- `comun.css`, incluye los estilos comunes en todas las páginas del sitio web, tanto del *frontend* como de la *extranet* y del *backend*.
- `frontend.css`, solamente incluye los estilos específicos de las páginas del *frontend* que no hayan sido incluidos en las dos hojas de estilos anteriores.
- `extranet.css`, incluye los estilos específicos de las páginas que forman la *extranet*.

NOTA Si estás desarrollando la aplicación a medida que lees el libro, puedes descargar el contenido de estos archivos en: https://github.com/javiereguiluz/Cupon/blob/2.8/web/css

En las siguientes secciones se muestran las alternativas que ofrece Symfony para gestionar estos *assets*.

7.2.1 Organizando los archivos web de manera simple

Si los *assets* de la aplicación no son muy complejos, la solución más simple consiste en guardarlos dentro de los directorios `web/css/` y `web/js/` y enlazarlos en las plantillas con la función `asset()` de Twig:

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <!-- ... -->
    <link rel="stylesheet" href="{{ asset('css/normalizar.css') }}" />
    <link rel="stylesheet" href="{{ asset('css/comun.css') }}" />
</head>
```

7.2.2 Organizando los archivos web con *bundles*

Si tu aplicación es compleja y la has dividido en varios *bundles*, puedes guardar los *assets* de cada *bundle* en sus directorios `Resources/public/css/` y `Resources/public/js/` en vez de guardarlos todos en `web/css/` y `web/js/`.

Inicialmente esos directorios no existen, por lo que debes crearlos a mano. Suponiendo que has definido los archivos `normalizar.css` y `comun.css` en el directorio `src/AppBundle/Resources/public/css/`, puedes enlazar a ellos desde tus plantillas de la siguiente manera:

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <!-- ... -->
    <link rel="stylesheet" href="{{ asset('bundles/app/css/normalizar.css') }}"
/>
    <link rel="stylesheet" href="{{ asset('bundles/app/css/comun.css') }}" />
</head>
```

La ruta web de los *assets* guardados en bundles se construye de la siguiente manera: `bundles/` + nombre del bundle en minúsculas y sin sufijo `Bundle` (ej. `AppBundle` -> `app`) y después la ruta dentro de `Resources/public/` (en este caso, `css/normalizar.css` y `css/comun.css`).

No obstante, si después de enlazar estos *assets* cargas cualquier página de la aplicación (ej. la portada en `http://127.0.0.1:8000/`), verás que no se están aplicando. El motivo es que el enlace `asset('bundles/app/css/normalizar.css')` corresponde al archivo `<proyecto>/web/bundles/app/css/normalizar.css` y si abres el directorio `web/`, verás que ese archivo no existe.

Para que los archivos web de un *bundle* puedan ser utilizados en el sitio web, primero es necesario *instalarlos* mediante el comando `assets:install`:

```
$ php app/console assets:install

Installing assets for AppBundle into web/bundles/app
Installing assets for ...
```

Si guardas los archivos web en otro directorio diferente a `web/`, indica su ruta como argumento del comando anterior:

```
$ php app/console assets:install /ruta/de/algun/directorio
```

Instalar los *assets* consiste en copiar el contenido del directorio `Resources/public/` de cada *bundle* dentro de `web/bundles/{nombre-del-bundle}/`. Si en vez de copiar los contenidos prefieres crear un enlace simbólico, añade la opción `--symlink` al comando:

```
$ php app/console assets:install --symlink
```

Actualiza ahora la página y verás la portada con los nuevos estilos aplicados.

7.2.3 Organizando los archivos web con herramientas JavaScript

Debido al gran número de herramientas disponibles y a las diferentes formas de gestionar los *assets* dependiendo de cada aplicación, la gestión de los *assets* con herramientas JavaScript está fuera del objetivo de este libro.

Consulta la sección front-end (http://symfony.com/doc/current/frontend.html) de la documentación de Symfony para acceder a los artículos más recientes relacionados con este tema.

7.3 Mejorando la gestión de las imágenes

En el capítulo anterior, las imágenes de las ofertas se enlazaron de la siguiente manera con la función `asset()`:

```
<img alt="..." src="{{ asset('uploads/images/' ~ oferta.rutaFoto) }}">
```

El prefijo `uploads/images/` es necesario porque las imágenes de las ofertas se guardan en un directorio determinado. El problema es que debes incluir ese prefijo en cualquier plantilla de la aplicación que muestre fotos de ofertas, lo cual es bastante aburrido y propenso a cometer errores.

Symfony propone solucionar este problema mediante los **paquetes de assets**, que permiten definir una configuración común para una serie de *assets*. En este caso, el objetivo es prefijar automáticamente `uploads/images/` a las fotos de las ofertas. Para ello, añade primero la siguiente configuración que crea un paquete de *assets* llamado `ofertas`:

```
# app/config/config.yml
# ...
framework:
    # ...
    assets:
        packages:
            ofertas:
                base_path: 'uploads/images/'
```

Ahora, actualiza las plantillas para incluir un segundo argumento en la función `asset()` que indica la configuración que hay que usar para generar las URL:

```
{# antes #}
<img alt="..." src="{{ asset('uploads/images/' ~ oferta.rutaFoto) }}">

{# después #}
<img alt="..." src="{{ asset(oferta.rutaFoto, 'ofertas') }}">
```

La gran ventaja de utilizar la función `asset()` para enlazar los *assets* es que proporciona mucha flexibilidad a la aplicación para mover, reordenar y versionar sus *assets* con facilidad.

7.4 Seleccionando la ciudad activa

Una de las claves del buen funcionamiento del *frontend* es que la ciudad activa en cada momento se incluye como parte de todas las URL (salvo las que están relacionadas con los usuarios).

Además, los usuarios deben poder cambiar fácilmente la ciudad activa para ver las ofertas y tiendas de cualquier otra ciudad que les interese. Para cambiar la ciudad, se incluye dentro del menú principal de navegación una lista desplegable con todas las ciudades disponibles:

Figura 7.2 Lista desplegable para seleccionar la ciudad activa

Cuando el usuario seleccione cualquier ciudad de la lista, la aplicación le redirige automáticamente a la portada de la ciudad seleccionada. Para conseguirlo, añade en primer lugar una nueva ruta que se encargue de responder a la acción de cambiar de ciudad. Como la funcionalidad está relacionada con las ciudades, lo más correcto es añadir la en el controlador `CiudadController` dentro del *bundle* `AppBundle`.

El código del controlador es muy sencillo, ya que sólo debe obtener el valor de la nueva ciudad a partir de la URL `/ciudad/cambiar-a-{ciudad}` y redirigir al usuario a la portada de esa ciudad:

```
// src/AppBundle/Controller/CiudadController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class CiudadController extends Controller
{
    /**
     * @Route("/ciudad/cambiar-a-{ciudad}", name="ciudad_cambiar")
     */
    public function cambiarAction($ciudad)
    {
        return $this->redirectToRoute('portada', array('ciudad' => $ciudad));
    }

    // ...
}
```

Accede ahora por ejemplo a la URL `http://127.0.0.1:8000/app_dev.php/ciudad/cambiar-a-madrid` y verás cómo la aplicación te redirige a la portada de la ciudad cuyo *slug* es `madrid`. Sin embargo, si accedes por ejemplo a la portada de Vitoria-Gasteiz, verás una excepción con el siguiente mensaje de error:

```
"Parameter "ciudad" for route "ciudad_cambiar" must match "[^\-]+"
```

El motivo de este error es que la URL de la ruta `ciudad_cambiar` es `/ciudad/cambiar-a-{ciudad}` en vez de utilizar un patrón más habitual tipo `/ciudad/cambiar/{ciudad}`. Así que la URL mezcla partes fijas (`cambiar-a-`) con partes variables (`{ciudad}`) que pueden contener cualquier contenido.

Para evitar problemas, cuando Symfony procesa el patrón de una ruta, impide que la parte variable (en este caso, `{ciudad}`) contenga cualquier separador de la parte fija (en este caso, el guión medio `-`). De la misma forma, si el patrón fuese `/ciudad/cambiar_a_{ciudad}`, el nombre la ciudad no podría tener guiones bajos `_`.

Aunque este es el comportamiento por defecto de Symfony, resulta muy sencillo modificarlo mediante la opción `requirements` de la ruta. Define un requisito para la variable `ciudad` y haz que su expresión regular permita cualquier carácter:

```
// Antes
/**
 * @Route(
 *     "/ciudad/cambiar-a-{ciudad}",
 *     name="ciudad_cambiar"
 * )
 */

// Después
/**
 * @Route(
 *     "/ciudad/cambiar-a-{ciudad}",
 *     requirements={ "ciudad" = ".+" },
 *     name="ciudad_cambiar"
 * )
 */
```

El siguiente paso consiste en crear la lista desplegable que muestra a los usuarios del sitio web todas las ciudades disponibles en la aplicación. Suponiendo que la lista de ciudades es muy larga y evoluciona con el tiempo, no es recomendable definir el listado como una opción de configuración sino hacer directamente una consulta a la base de datos.

Un problema añadido es que la lista desplegable se incluye en todas las páginas del *frontend*, por lo que todas las acciones de todos los controladores deberían hacer la misma consulta y pasar la misma variable a las plantillas.

La forma más eficiente de no repetir una y otra vez la misma lógica en diferentes plantillas consiste en renderizar directamente la respuesta de una acción mediante la función `render()` de Twig. Abre el *layout* del *frontend* y añade lo siguiente dentro del menú principal de navegación:

```
{# app/Resources/views/frontend.html.twig #}

{# ... #}
```

```
<header>
    <h1><a href="{{ path('portada') }}">CUPON</a></h1>
    <nav>
        <ul>
            <li><a href="{{ path('portada') }}">Oferta del día</a></li>
            ...

            <li>{{ render(controller('AppBundle:Ciudad:listaCiudades')) }}</li
        </ul>
    </nav>
</header>
```

La instrucción `render()` hace que en ese punto de la plantilla se inserte el resultado de ejecutar la acción `listaCiudadesAction()` del controlador `CiudadController.php` del *bundle* `AppBundle`.

La acción `listaCiudadesAction()` es realmente sencilla, ya que sólo debe obtener el listado de todas las ciudades de la aplicación. Además, no es necesario que le asocies ninguna ruta mediante `@Route`, ya que este método solo se va a ejecutar mediante una llamada a la función `render()` y nunca directamente a través de una URL pública:

```
// src/AppBundle/Controller/CiudadController.php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class CiudadController extends Controller
{
    // ...

    public function listaCiudadesAction()
    {
        $em = $this->getDoctrine()->getManager();
        $ciudades = $em->getRepository('AppBundle:Ciudad')->findAll();

        return $this->render('ciudad/_lista_ciudades.html.twig', array(
            'ciudades' => $ciudades
        ));
    }
}
```

Su plantilla asociada recorre el array de ciudades que le pasa la acción y crea una lista desplegable con todos sus valores:

```
{# app/Resources/views/ciudad/_lista_ciudades.html.twig #}
<select id="ciudadseleccionada">
{% for ciudad in ciudades %}
    <option value="{{ ciudad.slug }}">{{ ciudad.nombre }}</option>
{% endfor %}
</select>

<script type="text/javascript">
```

```
    var lista  = document.getElementById('ciudadseleccionada');
    var ciudad = lista.options[lista.selectedIndex].value;

    lista.onchange = function() {
        var url = "{{ path('portada', { ciudad: ciudad }) }}";
    };
</script>
```

El código JavaScript anterior hace que cada vez que el usuario selecciona una nueva ciudad en la lista desplegable, la aplicación le redirige a la portada de la nueva ciudad. Sin embargo, si pruebas esta plantilla pronto descubrirás que tiene un error muy importante.

Cuando se selecciona una nueva ciudad, se obtiene su *slug* a partir de la lista desplegable y se genera la URL de la portada con la función `path()` de Twig. El problema es que Twig sólo funciona en el servidor, no en el navegador del usuario. Así que el código anterior no es capaz de generar la URL y no podrá redirigir al usuario a ninguna página.

La solución más sencilla sería no utilizar la función `path()` y en su lugar, crear *a mano* la URL de la portada. Esta técnica no es aplicable a escenarios más complejos, así que es mejor hacer uso de una solución alternativa. Como el número de elementos de la lista no es muy elevado, se pueden generar todas las rutas en el servidor y añadirlas después a cada elemento de la lista mediante un atributo llamado `data-url`:

```
{# app/Resources/views/ciudad/_lista_ciudades.html.twig #}
<select id="ciudadseleccionada">
{% for ciudad in ciudades %}
    <option value="{{ ciudad.slug }}"
            data-url="{{ url('ciudad_cambiar', { ciudad: ciudad.slug }) }}">
        {{ ciudad.nombre }}
    </option>
{% endfor %}
</select>

<script type="text/javascript">
    var lista = document.getElementById('ciudadseleccionada');

    lista.onchange = function() {
        var url = lista.options[lista.selectedIndex].getAttribute('data-url');
        window.location = url;
    };
</script>
```

Con esta técnica, el código HTML de la lista desplegable tendrá el siguiente aspecto:

```
<select id="ciudadseleccionada">
  <option value="alicante"
   data-url="http://127.0.0.1:8000/app_dev.php/ciudad/cambiar-a-alicante">
     Alicante
  </option>
```

```
  <option value="badalona"
   data-url="http://127.0.0.1:8000/app_dev.php/ciudad/cambiar-a-badalona">
     Badalona
  </option>

  <option value="barcelona"
   data-url="http://127.0.0.1:8000/app_dev.php/ciudad/cambiar-a-barcelona">
     Barcelona
  </option>

  <!-- ... -->
</select>
```

Finalmente, para redirigir a la nueva página, sólo es necesario obtener mediante JavaScript el valor del atributo `data-url` de la opción seleccionada en la lista desplegable.

NOTA El estándar HTML5 permite añadir atributos propios a cualquier elemento de la página. El nombre de estos atributos siempre comienza por `data-` y pueden almacenar cualquier información. Estos atributos no son visibles por pantalla y no afectan ni al contenido ni al aspecto de la página. Más información: http://www.w3.org/TR/html5/elements.html

Para dar por terminada la lista desplegable, lo único que falta es que aparezca seleccionada la ciudad activa de la aplicación. Para ello, desde el *layout* del *frontend* se pasa una variable a la acción `listaCiudades`, que a su vez pasa la variable a la plantilla que crea la lista desplegable.

Define una variable llamada `ciudadSeleccionada` en el *layout* del *frontend* y pásala a la acción (por el momento no te preocupes en cómo obtener el valor de la variable `ciudadSeleccionada`):

```
{# app/Resources/views/frontend.html.twig #}

{% set ciudadSeleccionada = ... %}

{# ... #}

<li>{{ render(controller('AppBundle:Ciudad:listaCiudades', {
    ciudad: ciudadSeleccionada
})) }}</li>
```

Después, recoge la variable en la acción y pásala a la plantilla que muestra la lista desplegable:

```
// src/AppBundle/Controller/CiudadController.php
class CiudadController extends Controller
{
    // ...

    public function listaCiudadesAction($ciudad)
```

```
    {
        $em = $this->getDoctrine()->getManager();
        $ciudades = $em->getRepository('AppBundle:Ciudad')->findAll();

        return $this->render('ciudad/_lista_ciudades.html.twig', array(
            'ciudadActual' => $ciudad,
            'ciudades'     => $ciudades
        ));
    }
}
```

Por último, haz que la lista muestre seleccionado el elemento que coincida con la ciudad actual:

```
{# app/Resources/views/ciudad/_lista_ciudades.html.twig #}
<select id="ciudadseleccionada">
{% for ciudad in ciudades %}
    <option value="{{ ciudad.slug }}"
            data-url="{{ url('ciudad_cambiar', { ciudad: ciudad.slug }) }}"
            {{ ciudadActual == ciudad.slug ? 'selected' }}>
        {{ ciudad.nombre }}
    </option>
{% endfor %}
</select>

{# ... #}
```

Para que los cambios anteriores funcionen correctamente, sólo falta establecer el valor de la variable `ciudadSeleccionada` en el *layout* del *frontend*:

```
{# app/Resources/views/frontend.html.twig #}
{% set ciudadSeleccionada = app.request.attributes.has('ciudad')
    ? app.request.attributes.get('ciudad')
    : ciudad_por_defecto
%}
```

Si la URL de la página incluye una variable llamada `ciudad`, el método `app.request.attributes.has('ciudad')` devuelve `true` y por tanto, la ciudad activa de la aplicación es directamente la ciudad incluida en la URL. Si no se cumple lo anterior, se utiliza el valor de la ciudad por defecto de la aplicación.

El método `get()` anterior admite dos parámetros. El primero es el nombre del atributo cuyo valor quieres obtener. El segundo es el valor que devuelve el método cuando ese atributo no existe. Gracias a este funcionamiento, es posible simplificar el código de la plantilla anterior:

```
{# app/Resources/views/frontend.html.twig #}
{% set ciudadSeleccionada = app.request.attributes.get('ciudad', ciudad_por_defe
cto) %}
```

La variable llamada `ciudad_por_defecto` es una **variable global de Twig**, que se define mediante la opción de configuración `globals`:

```
# app/config/config.yml
twig:
    debug:            %kernel.debug%
    strict_variables: %kernel.debug%
    globals:
        ciudad_por_defecto: 'barcelona'
```

Las variables globales de Twig están disponibles en cualquier plantilla de la aplicación. Aunque en Symfony son muy fáciles de definir, no deberías abusar de las variables globales por razones de rendimiento y para no empeorar la calidad de tu código.

Después de añadir la variable global, el valor de la ciudad por defecto se define en dos opciones de configuración diferentes:

```
# app/config/config.yml
twig:
    # ...
    globals:
        ciudad_por_defecto: 'barcelona'

# ...

parameters:
    app.ciudad_por_defecto:  'barcelona'
```

Solucionar esta duplicidad es muy sencillo, ya que Symfony permite hacer referencia al valor de cualquier parámetro encerrando su nombre entre `%` y `%`:

```
# app/config/config.yml
twig:
    # ...
    globals:
        ciudad_por_defecto: %app.ciudad_por_defecto%

# ...

parameters:
    app.ciudad_por_defecto:  'barcelona'
```

7.5 Creando la página de detalle de una oferta

Cada oferta se muestra en una página con toda su información detallada y con los datos de la tienda en la que se publica. Además, en la zona lateral de la página se incluye un listado con cinco ofertas relacionadas.

En primer lugar, define la nueva acción y ruta dentro del controlador `OfertaController`:

```
// src/AppBundle/Controller/OfertaController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class OfertaController extends Controller
{
    /**
     * @Route("/{ciudad}/ofertas/{slug}", name="oferta")
     */
    public function ofertaAction($ciudad, $slug)
    {
        // ...
    }

    // ...
}
```

El nombre de la ruta (`oferta`) es muy corto para que las plantillas sean más concisas, ya que incluyen muchos enlaces a la página de detalle de las ofertas. El patrón de la ruta incluye tanto el *slug* de la oferta como el *slug* de la ciudad en la que se publica. Al crear un enlace se deben pasar dos parámetros, pero tiene la ventaja de que el mismo *slug* se puede utilizar en dos o más ofertas de diferentes ciudades:

```
{{ path('oferta', { slug: oferta.slug, ciudad: oferta.ciudad.slug }) }}
```

A continuación, añade el código necesario para buscar la oferta en la base de datos y renderizar el resultado mediante una nueva plantilla llamada `oferta/detalle.html.twig`:

```
// src/AppBundle/Controller/OfertaController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class OfertaController extends Controller
{
    /**
     * @Route("/{ciudad}/ofertas/{slug}", name="oferta")
     */
    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')
                     ->findOferta($ciudad, $slug);

        return $this->render('oferta/detalle.html.twig', array(
            'oferta' => $oferta
```

```
        ));
    }

    // ...
}
```

Para buscar la oferta en la base de datos se utiliza un método propio llamado `findOferta()` al que se le pasan los *slugs* de la ciudad y de la oferta. Como este método no existe en Doctrine, añádelo en el repositorio de la entidad `Oferta`:

```
// src/AppBundle/Repository/OfertaRepository.php
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
{
    public function findOferta($ciudad, $slug)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, c, t
              FROM AppBundle:Oferta o
              JOIN o.ciudad c JOIN o.tienda t
             WHERE o.revisada = true
               AND o.slug = :slug
               AND c.slug = :ciudad');
        $consulta->setParameter('slug', $slug);
        $consulta->setParameter('ciudad', $ciudad);
        $consulta->setMaxResults(1);

        return $consulta->getSingleResult();
    }
}
```

La plantilla que muestra los detalles de la oferta se llama `oferta/detalle.html.twig`. Como esta página es parecida a la portada, puedes copiar su código y actualizar los bloques `title` e `id`:

```
{# app/Resources/views/oferta/detalle.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Detalles de {{ oferta.nombre }}{% endblock %}
{% block id 'oferta' %}

{% block article %}
    <section class="descripcion">
        <h1><a href="#">{{ oferta.nombre }}</a></h1>
        {{ oferta.descripcion }}
        <a class="boton" href="#">Comprar</a>
```

```
    </section>

    {# ... #}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

Obviamente, copiar y pegar código entre varias plantillas no es una buena práctica recomendable. Cuando dos o más plantillas comparten código, es mejor extraer el código común en una nueva plantilla e incluirla después con la función `{{ include() }}` de Twig.

NOTA No es obligatorio, pero resulta habitual añadir un guión bajo (_) por delante del nombre de los "trozos" de plantilla para distinguirlos mejor de las plantillas reales.

Así que crea la plantilla común `oferta/_oferta_completa.html.twig` y copia el siguiente código:

```
{# app/Resources/views/oferta/_oferta_completa.html.twig #}
<section class="descripcion">
    <h1><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slug: oferta.sl
ug }) }}">{{ oferta.nombre }}</a></h1>

    {{ oferta.descripcion }}

    <a class="boton" href="#">Comprar</a>
</section>

<section class="galeria">
    <img alt="..." src="{{ asset(oferta.rutaFoto, 'ofertas') }}">

    <p class="precio">{{ oferta.precio }} &euro; <span>{{ oferta.descuento }}</s
pan></p>

    <p><strong>Condiciones:</strong> {{ oferta.condiciones }}</p>
</section>

{# ... #}

<section class="direccion">
    <h2>Disfruta de la oferta en</h2>
    <p>
        <a href="#">{{ oferta.tienda.nombre }}</a>
        {{ oferta.tienda.direccion }}
    </p>
</section>

<section class="tienda">
```

```
    <h2>Sobre la tienda</h2>
    {{ oferta.tienda.descripcion }}
</section>
```

Ahora que el código que muestra los detalles de una oferta se encuentra en una plantilla aparte, ya puedes simplificar el código de la plantilla `oferta/detalle.html.twig`:

```
{# app/Resources/views/oferta/detalle.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Detalles de {{ oferta.nombre }}{% endblock %}
{% block id 'oferta' %}

{% block article %}
    {{ include('oferta/_oferta_completa.html.twig') }}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

La plantilla de la portada también se puede simplificar haciendo uso de la plantilla común que muestra la información de una oferta:

```
{# app/Resources/views/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Cupon, cada día ofertas increíbles en tu ciudad con descuentos
de hasta el 90%{% endblock %}
{% block id 'portada' %}

{% block article %}
    {{ include('oferta/_oferta_completa.html.twig') }}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

7.5.1 Mostrando las ofertas de otras ciudades

La zona lateral de la página de detalle de una oferta muestra un listado con cinco ofertas de ciudades diferentes a la ciudad de la oferta que se está viendo. Para ello, la acción debe buscar las cinco ofertas y pasarlas a la plantilla:

```
// src/AppBundle/Controller/OfertaController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
class OfertaController extends Controller
{
    /**
     * @Route("/{ciudad}/ofertas/{slug}", name="oferta")
     */
    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')
                     ->findOferta($ciudad, $slug);
        $relacionadas = $em->getRepository('AppBundle:Oferta')
                           ->findRelacionadas($ciudad);

        return $this->render('oferta/detalle.html.twig', array(
            'oferta' => $oferta,
            'relacionadas' => $relacionadas
        ));
    }

    // ...
}
```

Como es evidente, el método `findRelacionadas()` para encontrar ofertas relacionadas no está definido por Doctrine, por lo que se debe incluir en el repositorio propio de la entidad `Oferta`:

```
// src/AppBundle/Repository/OfertaRepository.php
class OfertaRepository extends EntityRepository
{
    // ...

    public function findRelacionadas($ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, c
              FROM AppBundle:Oferta o
              JOIN o.ciudad c
             WHERE o.revisada = true
               AND o.fechaPublicacion <= :fecha
               AND c.slug != :ciudad
          ORDER BY o.fechaPublicacion DESC');
        $consulta->setMaxResults(5);
        $consulta->setParameter('ciudad', $ciudad);
        $consulta->setParameter('fecha', new \DateTime('today'));

        return $consulta->getResult();
```

```
    }
}
```

La consulta necesaria para obtener las ofertas relacionadas depende de la definición que la aplicación haga de *"oferta relacionada"*. En este ejemplo sencillo, se considera que una oferta está relacionada con una ciudad si es una oferta aprobada, publicada y no pertenece a la ciudad indicada.

Con esta información, la plantilla de detalle de una oferta ya puede mostrar el listado de ofertas relacionadas:

```
{# app/Resources/views/oferta/detalle.html.twig #}
{% extends 'frontend.html.twig' %}

{# ... #}

{% block aside %}
    {{ parent() }}

    <section id="relacionadas">
        <h2>Ofertas en otras ciudades</h2>
        <ul>
            {% for oferta in relacionadas %}
            <li>{{ oferta.ciudad.nombre }}: <a href="{{ path('oferta', { ciuda
d: oferta.ciudad.slug, slug: oferta.slug }) }}">{{ oferta.nombre }}</a></li>
            {% endfor %}
        </ul>
    </section>
{% endblock %}
```

Recuerda que la función `parent()` hace referencia al contenido de ese mismo bloque en la *plantilla padre* de esta plantilla. Por tanto, la lista de ofertas relacionadas se muestra debajo de cualquier otro contenido que defina la plantilla padre (en este caso, `app/Resources/views/frontend.html.twig`).

7.6 Completando las plantillas con extensiones de Twig

La portada y la página de detalle de una oferta ya incluyen toda la información necesaria, pero no muestran el aspecto deseado. El tiempo que resta para que expire la oferta debería mostrarse como una cuenta atrás de JavaScript actualizada cada segundo. Y la descripción de la oferta se debe mostrar como una lista de elementos (`<ul>`).

Como se trata de transformar el aspecto de la información, esta tarea es responsabilidad de la parte de la vista, es decir, de Twig. Para ello se crea una extensión propia de Twig que defina los nuevos filtros y funciones, tal como se explica en la sección *[Creando extensiones propias de Twig](#creando-extensiones-propias-de-twig)* del apéndice A.

La extensión propia de Twig que se va a desarrollar incluye los siguientes filtros y funciones:

- `descuento($precio, $descuentoEnEuros, $decimales = 0)`, función que muestra el descuento porcentual que supone el `$descuentoEnEuros` que se le pasa respecto al `$precio`. El parámetro `$decimales` indica el número de decimales con el que se muestra el descuento.
- `mostrar_como_lista($tipo = 'ul')`, filtro que formatea un contenido de texto para convertirlo en una lista de elementos. Cada línea del contenido original se transforma en un elemento `<li>`. El tipo de lista es por defecto `<ul>`, pero se puede modificar con el único argumento del filtro.
- `cuenta_atras`, filtro que transforma la fecha sobre la que se aplica en una cuenta atrás del tiempo que falta hasta llegar a esa fecha, actualizada cada segundo mediante JavaScript.

Aunque los filtros tienen funcionalidades muy dispares, se incluyen todos en una misma extensión llamada `Cupon`. Para seguir las convenciones recomendadas por Symfony, crea el directorio `Twig/Extension/` dentro de `src/AppBundle/`. En su interior, añade un archivo llamado `CuponExtension.php` con el siguiente contenido:

```
# src/AppBundle/Twig/Extension/CuponExtension.php
namespace AppBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    public function getName()
    {
        return 'cupon';
    }
}
```

A continuación, activa la extensión configurando un nuevo servicio:

```
# app/config/services.yml
services:
    # ...
    app.twig.cupon_extension:
        class: AppBundle\Twig\Extension\CuponExtension
        tags:
            - { name: twig.extension }
```

Ahora ya puedes añadir la primera función `descuento()` en la extensión:

```
# src/AppBundle/Twig/Extension/CuponExtension.php
namespace AppBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    public function getFunctions()
    {
        return array(
            new \Twig_SimpleFunction('descuento', array($this, 'descuento')),
        );
```

```
    }

    public function descuento($precio, $descuento, $decimales = 0)
    {
        if (!is_numeric($precio) || !is_numeric($descuento)) {
            return '-';
        }

        if ($descuento == 0 || $descuento == null) {
            return '0%';
        }

        $precio_original = $precio + $descuento;
        $porcentaje = ($descuento / $precio_original) * 100;

        return '-'.number_format($porcentaje, $decimales).'%';
    }

    public function getName()
    {
        return 'cupon';
    }
}
```

Ahora ya puedes visualizar correctamente el precio y su descuento en la plantilla que muestra los detalles de una oferta:

```
{# app/Resources/views/oferta/_oferta_completa.html.twig #}
<section class="galeria">
    {# ... #}

    <p class="precio">{{ oferta.precio }} &euro;
    <span>{{ descuento(oferta.precio, oferta.descuento) }}</span></p>

    {# ... #}
</section>
```

El siguiente filtro que se añade en la extensión es `mostrar_como_lista`:

```
# src/AppBundle/Twig/Extension/CuponExtension.php
namespace AppBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    // ...

    public function getFilters()
    {
        return array(
```

```
            new \Twig_SimpleFilter(
                'mostrar_como_lista',
                array($this, 'mostrarComoLista'),
                array('is_safe' => array('html'))
            ),
        );
    }

    public function mostrarComoLista($value, $tipo='ul')
    {
        $html = "<".$tipo.">\n";
        $html .= "  <li>".str_replace("\n", "</li>\n  <li>", $value)."</li>\n";
        $html .= "</".$tipo.">\n";

        return $html;
    }
}
```

Utilizando este filtro ya es posible mostrar la descripción de una oferta en forma de lista HTML:

```
{# app/Resources/views/oferta/_oferta_completa.html.twig #}
<section class="descripcion">
    <h1><a href="{{ path( ... ) }}">{{ oferta.nombre }}</a></h1>

    {{ oferta.descripcion|mostrar_como_lista }}

    {# ... #}
```

El último filtro puede parecer el más complejo debido al uso de código JavaScript, pero técnicamente se define igual que cualquier otro filtro:

```
# src/AppBundle/Twig/Extension/CuponExtension.php
namespace AppBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    // ...

    public function getFilters()
    {
        return array(
            // ...
            new \Twig_SimpleFilter(
                'cuenta_atras',
                array($this, 'cuentaAtras'),
                array('is_safe' => array('html'))
            ),
        );
    }
```

```
    public function cuentaAtras($fecha)
    {
        $fecha = $fecha->format('Y,')
                 .($fecha->format('m')-1)
                 .$fecha->format(',d,H,i,s');

        $html = <<<EOJ
<script type="text/javascript">
function muestraCuentaAtras(){
    var horas, minutos, segundos;
    var ahora = new Date();
    var fechaExpiracion = new Date($fecha);
    var falta = Math.floor( (fechaExpiracion.getTime() - ahora.getTime()) / 100
0 );

    if (falta < 0) {
        cuentaAtras = '-';
    }
    else {
        horas = Math.floor(falta/3600);
        falta = falta % 3600;

        minutos = Math.floor(falta/60);
        falta = falta % 60;

        segundos = Math.floor(falta);

        cuentaAtras = (horas < 10    ? '0' + horas    : horas)    + 'h '
                    + (minutos < 10  ? '0' + minutos  : minutos)  + 'm '
                    + (segundos < 10 ? '0' + segundos : segundos) + 's ';

        setTimeout('muestraCuentaAtras()', 1000);
    }

document.getElementById('tiempo').innerHTML = '<strong>Faltan:</strong> ' + cuen
taAtras;
}

muestraCuentaAtras();
</script>
EOJ;

        return $html;
    }
}
```

Ahora ya puedes mostrar cualquier fecha futura como una cuenta atrás del tiempo restante:

```
{# app/Resources/views/oferta/_oferta_completa.html.twig #}
<section class="estado">
    <div id="tiempo">
        <strong>Faltan</strong>: {{ oferta.fechaExpiracion|cuenta_atras }}
    </div>

    {# ... #}
```

NOTA El código fuente público de la aplicación *Cupon* (https://github.com/javiereguiluz/Cupon) incluye una versión ligeramente más avanzada del filtro `cuenta_atras` que permite incluir varias cuentas atrás en una misma página.

7.7 Creando la página de ofertas recientes de una ciudad

La página de ofertas recientes de una ciudad se muestra al pinchar la opción *Ofertas recientes* del menú principal de navegación. La página muestra las cinco ofertas más recientes de la ciudad activa. Además, en la zona lateral se incluye un listado con enlaces a las cinco ciudades más cercanas a la ciudad activa.

El desarrollo de la página se realiza siguiendo los mismos pasos que en las páginas anteriores: ruta, acción, repositorio y plantilla. En primer lugar, añade la siguiente acción en `CiudadController`:

```
// src/AppBundle/Controller/CiudadController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class CiudadController extends Controller
{
    /**
     * @Route("/{ciudad}/recientes", name="ciudad_recientes")
     */
    public function recientesAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();

        $ciudad = $em->getRepository('AppBundle:Ciudad')
                     ->findOneBySlug($ciudad);
        $cercanas = $em->getRepository('AppBundle:Ciudad')
                       ->findCercanas($ciudad->getId());
        $ofertas  = $em->getRepository('AppBundle:Oferta')
                       ->findRecientes($ciudad->getId());

        return $this->render('ciudad/recientes.html.twig', array(
            'ciudad'   => $ciudad,
            'cercanas' => $cercanas,
            'ofertas'  => $ofertas
```

```
        ));
    }

    // ...
}
```

El parámetro `$ciudad` que pasa la ruta a la acción es el *slug* de la ciudad. Para obtener el objeto completo de la ciudad, utiliza el método `findOneBySlug()` de la entidad `Ciudad`.

Después, para obtener las ofertas recientes y las ciudades cercanas se utilizan respectivamente los métodos `findRecientes()` y `findCercanas()`. Como estos métodos no están definidos en Doctrine, debes añadirlos al repositorio propio de cada entidad.

El repositorio propio de la entidad `Ciudad` todavía no existe, así que primero añade el parámetro `repositoryClass` en la clase de la entidad:

```
// src/AppBundle/Entity/Ciudad.php
/**
* @ORM\Entity(repositoryClass="AppBundle\Entity\CiudadRepository")
*/
class Ciudad
{
    // ...
}
```

Después, crea el archivo `CiudadRepository` y añade el código de la consulta `findCercanas()`:

```
// src/AppBundle/Repository/CiudadRepository.php
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class CiudadRepository extends EntityRepository
{
    public function findCercanas($ciudad_id)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT c
              FROM CiudadBundle:Ciudad c
             WHERE c.id != :id
          ORDER BY c.nombre ASC');
        $consulta->setMaxResults(5);
        $consulta->setParameter('id', $ciudad_id);

        return $consulta->getResult();
    }
}
```

La consulta del método anterior simplemente busca cinco ciudades cualquiera que sean distintas a la ciudad indicada. En una aplicación web real, las entidades de tipo `Ciudad` deberían tener más información para poder realizar geobúsquedas.

A continuación, añade el método `findRecientes()` en el repositorio de la entidad `Oferta`:

```
// src/AppBundle/Repository/OfertaRepository.php
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
{
    // ...

    public function findRecientes($ciudad_id)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t
              FROM AppBundle:Oferta o
              JOIN o.tienda t
             WHERE o.revisada = true
               AND o.fechaPublicacion < :fecha
               AND o.ciudad = :id
          ORDER BY o.fechaPublicacion DESC');
        $consulta->setMaxResults(5);
        $consulta->setParameter('id', $ciudad_id);
        $consulta->setParameter('fecha', new \DateTime('today'));

        return $consulta->getResult();
    }
}
```

Después de obtener la información a través de los repositorios de las entidades, la acción `recientesAction()` utiliza la plantilla `ciudad/recientes.html.twig` para mostrar sus resultados:

```
{# app/Resources/views/ciudad/recientes.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Ofertas recientes en {{ ciudad.nombre }}{% endblock %}
{% block id 'recientes' %}

{% block article %}
    <h1>Ofertas recientes en <strong>{{ ciudad.nombre }}</strong></h1>

    ## LISTADO DE OFERTAS ##
{% endblock %}
```

```
{% block aside %}
    {{ parent() }}

    <section id="cercanas">
        <h2>Ofertas en otras ciudades</h2>
        <ul>
            {% for ciudad in cercanas %}
            <li><a href="{{ path('ciudad_recientes', { ciudad: ciudad.slug }) }}">{{ ciudad.nombre }}</a></li>
            {% endfor %}
        </ul>
    </section>
{% endblock %}
```

Tras haber desarrollado varias páginas en los capítulos y secciones anteriores, el código de la plantilla anterior es trivial. El único código que falta es el de la zona marcada como `## LISTADO DE OFERTAS ##`. Este listado es idéntico al que se utilizará en la página que muestra las compras más recientes del usuario logueado (y que se desarrolla en el próximo capítulo).

Como dos o más plantillas van a utilizar exactamente el mismo código, se incluye todo el código común en una nueva plantilla. Siguiendo el mismo razonamiento que en la plantilla `_oferta_completa.html.twig`, se define una plantilla llamada `oferta/_oferta_simple.html.twig` en ese mismo directorio. Las dos plantillas son similares, pero `_oferta_simple.html.twig` muestra mucha menos información para la oferta, lo que la hace ideal para utilizarla en listados de ofertas.

A continuación se muestra el código de la nueva plantilla:

```
{# app/Resources/views/oferta/_oferta_simple.html.twig #}
<section class="oferta mini">
    <div class="descripcion">
        <h2><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slug: oferta.slug }) }}">{{ oferta.nombre }}</a></h2>

        {{ oferta.descripcion|mostrar_como_lista }}

        <a class="boton" href="#">Comprar</a>

        <div class="estado">
          <strong>Faltan</strong>: {{ oferta.fechaExpiracion|cuenta_atras }}
        </div>
    </div>

    <div class="galeria">
        <img alt="Fotografía de la oferta" src="{{ asset(oferta.rutaFoto, 'ofertas') }}">

        <p class="precio">{{ oferta.precio }} &euro; <span>{{ descuento(oferta.p
```

```
recio, oferta.descuento) }}</span></p>

        <p>Disfruta de esta oferta en <a href="#">{{ oferta.tienda.nombre
}}</a></p>
    </div>
</section>
```

Ahora ya puedes utilizar el código anterior en cualquier plantilla:

```
{# app/Resources/views/ciudad/recientes.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Ofertas recientes en {{ ciudad.nombre }}{% endblock %}
{% block id 'recientes' %}

{% block article %}
    <h1>Ofertas recientes en <strong>{{ ciudad.nombre }}</strong></h1>

    {% for oferta in ofertas %}
        {{ include('oferta/_oferta_simple.html.twig') }}
    {% else %}
        <p>Esta ciudad todavía no ha publicado ninguna oferta</p>
    {% endfor %}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

7.8 Creando la portada de cada tienda

La portada de una tienda es la página que se muestra al pinchar sobre el nombre de cualquier tienda en cualquier página del sitio web. La página muestra información básica sobre la tienda, una tabla con sus últimas ofertas publicadas y un listado de otras tiendas de la misma ciudad.

La página está relacionada con las tiendas, así que añade la siguiente acción en el controlador `TiendaController`:

```
// src/AppBundle/Controller/TiendaController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TiendaController extends Controller
{
    /**
     * @Route("/{ciudad}/tiendas/{tienda}", requirements={ "ciudad" = ".+" }, na
me="tienda_portada")
     */
```

```
    public function portadaAction(Request $request, $ciudad, $tienda)
    {
        $em = $this->getDoctrine()->getManager();

        $ciudad = $em->getRepository('AppBundle:Ciudad')
                     ->findOneBySlug($ciudad);

        $tienda = $em->getRepository('AppBundle:Tienda')->findOneBy(array(
            'slug'   => $tienda,
            'ciudad' => $ciudad->getId()
        ));

        if (!$tienda) {
            throw $this->createNotFoundException('No existe esta tienda');
        }

        $ofertas = $em->getRepository('AppBundle:Tienda')
                      ->findUltimasOfertasPublicadas($tienda->getId());

        $cercanas = $em->getRepository('AppBundle:Tienda')->findCercanas(
            $tienda->getSlug(),
            $tienda->getCiudad()->getSlug()
        );

        return $this->render('tienda/portada.html.twig', array(
            'tienda'   => $tienda,
            'ofertas'  => $ofertas,
            'cercanas' => $cercanas
        ));
    }

    // ...
}
```

Como es habitual, el código del controlador consiste en pedir información al modelo (los repositorios de Doctrine) y después pasarla a la vista (la plantilla Twig). Para obtener la información se utilizan dos métodos de búsqueda propios: `findUltimasOfertasPublicadas($tienda_id)` que devuelve las últimas ofertas publicadas por la tienda cuyo atributo `id` se indica y `findCercanas($tienda, $ciudad)`, que devuelve las cinco tiendas más cercanas a la tienda indicada.

La entidad `Tienda` todavía no dispone de un repositorio propio, así que primero hay que crearlo. Modifica el código de la entidad para indicar que ahora se utiliza un repositorio propio:

```
// src/AppBundle/Entity/Tienda.php
namespace AppBundle\Entity;
use Doctrine\ORM\Mapping as ORM;
```

```
/**
 * @ORM\Entity(repositoryClass="AppBundle\Entity\TiendaRepository")
 */
class Tienda
{
    // ...
}
```

Después, crea la clase del repositorio y copia el siguiente código para definir los dos métodos necesarios:

```
// src/AppBundle/Repository/TiendaRepository.php
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class TiendaRepository extends EntityRepository
{
    public function findUltimasOfertasPublicadas($tienda_id, $limite = 10)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t
              FROM AppBundle:Oferta o
              JOIN o.tienda t
             WHERE o.revisada = true
               AND o.fechaPublicacion < :fecha
               AND o.tienda = :id
          ORDER BY o.fechaExpiracion DESC');
        $consulta->setMaxResults($limite);
        $consulta->setParameter('id', $tienda_id);
        $consulta->setParameter('fecha', new \DateTime('now'));

        return $consulta->getResult();
    }

    public function findCercanas($tienda, $ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT t, c
              FROM AppBundle:Tienda t
              JOIN t.ciudad c
             WHERE c.slug = :ciudad
               AND t.slug != :tienda');
        $consulta->setMaxResults(5);
        $consulta->setParameter('ciudad', $ciudad);
        $consulta->setParameter('tienda', $tienda);
```

```
        return $consulta->getResult();
    }
}
```

Para terminar la página, crea la plantilla `tienda/portada.html.twig` y añade el siguiente código:

```
{# app/Resources/views/tienda/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Tienda {{ tienda.nombre }}{% endblock %}
{% block id 'tienda' %}

{% block article %}
    <section id="descripcion">
        <h1>{{ tienda.nombre }}</h1>
        <p>{{ tienda.descripcion }}</p>
    </section>

    <section id="ultimas">
    <h2>Últimas ofertas publicadas</h2>

    <table>
        <thead>
            <tr>
                <th>Fecha</th>
                <th>Oferta</th>
                <th>Precio</th>
                <th>Descuento</th>
                <th>Compras</th>
            </tr>
        </thead>
        <tbody>
            {% for oferta in ofertas %}
            <tr>
                <td>{{ oferta.fechaPublicacion|date }}</td>
                <td><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slu
g: oferta.slug }) }}">{{ oferta.nombre }}</a></td>
                <td>{{ oferta.precio }} &euro;</td>
                <td>{{ oferta.descuento }} &euro;</td>
                <td>{{ oferta.compras }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
    </section>
{% endblock %}

{% block aside %}
```

```
    {{ parent() }}

    <section id="cercanas">
        <h2>Otras tiendas en {{ tienda.ciudad.nombre }}</h2>
        <ul>
            {% for tienda in cercanas %}
            <li><a href="{{ path('tienda_portada', { ciudad: tienda.ciudad.slu
g, tienda: tienda.slug}) }}">{{ tienda.nombre }}</a></li>
            {% endfor %}
        </ul>
    </section>
{% endblock %}
```

7.9 Refactorización final

El código de las acciones, repositorios y plantillas de las secciones anteriores se ha simplificado al máximo para facilitar su explicación. Sin embargo, en una aplicación web real serían necesarios varios retoques y mejoras.

7.9.1 Ofertas expiradas

La plantilla que muestra los detalles de una oferta no tiene en cuenta si esta ya ha finalizado. Como se conoce la fecha de expiración, resulta muy sencillo refactorizar las plantillas `oferta.html.twig` y `_oferta_simple.html.twig`:

```
{# app/Resources/views/oferta/_oferta_completa.html.twig #}
{% set expirada=oferta.fechaExpiracion|date('YmdHis') < 'now'|date('YmdHis') %}
<section class="descripcion">
    <h1><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slug: oferta.sl
ug }) }}">{{ oferta.nombre }}</a></h1>

    {{ oferta.descripcion|mostrar_como_lista }}

    {% if not expirada %}
        <a class="boton" href="#">Comprar</a>
    {% endif %}
</section>

<section class="galeria">
    {# ... #}
</section>

<section class="estado {{ expirada ? 'expirada' }}">
{% if not expirada %}
    <div class="tiempo">
        <strong>Faltan</strong>: {{ oferta.fechaExpiracion|cuenta_atras }}
    </div>

    <div class="compras">
        <strong>Compras</strong>: {{ oferta.compras }}
```

```
    </div>

    <div class="faltan">
        {% set faltan = oferta.umbral - oferta.compras %}
        {% if faltan > 0 %}
            Faltan <strong>{{ faltan }} compras</strong> <br/>
            para activar la oferta
        {% else %}
            <strong>Oferta activada</strong> por superar las
            <strong>{{ oferta.umbral }}</strong> compras necesarias
        {% endif %}
    </div>
{% else %}
    <div class="tiempo">
        <strong>Finalizada</strong> el {{ oferta.fechaExpiracion|date }}
    </div>

    <div class="compras">
        <strong>Compras</strong>: {{ oferta.compras }}
    </div>
{% endif %}
</section>

<section class="direccion">
    {# ... #}
</section>

<section class="tienda">
    {# ... #}
</section>
```

La clave del código anterior es la variable `expirada` que define la propia plantilla. Su valor es `true` o `false` en función de si la fecha de expiración de la oferta es anterior a la fecha actual. Como Twig no permite comparar dos fechas directamente, se formatea cada fecha con el primer argumento del filtro `date()` y se comparan los valores resultantes.

Una vez definida la variable, se emplea la instrucción `{% if not expirada %}` para ejecutar el código de las ofertas que todavía no han expirado. Así por ejemplo el botón *Comprar* sólo se muestra para las ofertas que todavía se pueden comprar.

En la plantilla `_oferta_simple.html.twig` se puede utilizar la misma estrategia para mostrar información diferente según la oferta haya expirado o no:

```
{# app/Resources/views/oferta/_oferta_simple.html.twig #}
{% set expirada = oferta.fechaExpiracion|date('YmdHis') < 'now'|date('YmdHis')
%}

<section class="oferta mini">
    <div class="descripcion">
```

```
        <h2><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slug: ofert
a.slug }) }}">{{ oferta.nombre }}</a></h2>

        {{ oferta.descripcion|mostrar_como_lista }}

        {% if not expirada %}
            <a class="boton" href="#">Comprar</a>
        {% endif %}

        <div class="estado {{ expirada ? 'expirada' }}">
            {% if not expirada %}
            <strong>Faltan</strong>: {{ oferta.fechaExpiracion|cuenta_atras }}
            {% else %}
            Finalizada el {{ oferta.fechaExpiracion|date }}
            {% endif %}
        </div>
    </div>

    <div class="galeria">
        {# ... #}
    </div>
</section>
```

7.9.2 Controlando los errores y las situaciones anómalas

La calidad de una aplicación depende en gran medida de la forma en la que gestiona los errores y las situaciones anómalas que inevitablemente se van a producir durante su ejecución.

Las acciones por ejemplo nunca deben suponer que las consultas a la base de datos devuelven los resultados esperados. Así, la acción que muestra los detalles de una oferta debe estar preparada para cuando no exista la oferta solicitada:

```
// src/AppBundle/Controller/OfertaController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class OfertaController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}/ofertas/{slug}", name="oferta")
     */
    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')
                     ->findOferta($ciudad, $slug);

        if (!$oferta) {
```

```
            throw $this->createNotFoundException('No existe la oferta');
        }

        // ...
    }
}
```

Igualmente, la acción que muestra las ofertas más recientes de una ciudad debe considerar la posibilidad de que no exista la ciudad solicitada:

```
// src/AppBundle/Controller/CiudadController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class CiudadController extends Controller
{
    /**
     * @Route("/{ciudad}/recientes", name="ciudad_recientes")
     */
    public function recientesAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();
        $ciudad = $em->getRepository('AppBundle:Ciudad')
                  ->findOneBySlug($ciudad);

        if (!$ciudad) {
            throw $this->createNotFoundException('No existe la ciudad');
        }

        // ...
    }
}
```

Finalmente, la acción que muestra la portada de cada tienda también debe redirigir a la página de error 404 cuando no exista la tienda indicada:

```
// src/AppBundle/Controller/TiendaController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class TiendaController extends Controller
{
    /**
     * @Route("/{ciudad}/tiendas/{tienda}", requirements={ "ciudad" = ".+" }, na
me="tienda_portada")
     */
    public function portadaAction($ciudad, $tienda)
    {
        // ...
```

```
        $tienda = $em->getRepository('AppBundle:Tienda')->findOneBy(array(
            'slug'   => $tienda,
            'ciudad' => $ciudad->getId()
        ));

        if (!$tienda) {
            throw $this->createNotFoundException('No existe la tienda');
        }

        // ...
    }
}
```

Por otra parte, las plantillas también deben estar preparadas para cuando reciban un array vacío en vez del array con objetos que esperaban. Para ello, se utiliza la estructura de control `for..else` de Twig. La página que muestra las ofertas recientes de una ciudad, incluye un mensaje para cuando no haya ofertas:

```
{# app/Resources/views/ciudad/recientes.html.twig #}

{# ... #}

{% for oferta in ofertas %}
    {{ include('oferta/_oferta_simple.html.twig') }}
{% else %}
    <p>Esta ciudad todavía no ha publicado ninguna oferta</p>
{% endfor %}

{# ... #}
```

Y la portada de una tienda también tiene en cuenta el caso en el que la tienda no haya publicado todavía ninguna oferta:

```
{# app/Resources/views/tienda/portada.html.twig #}

{# ... #}

{% for oferta in ofertas %}
<tr>
    <td>{{ oferta.fechaPublicacion|date }}</td>
    <td><a href="{{ path('oferta', { ciudad: oferta.ciudad.slug, slug: oferta.sl
ug }) }}">{{ oferta.nombre }}</a></td>
    <td>{{ oferta.precio }} &euro;</td>
    <td>{{ oferta.descuento }} &euro;</td>
    <td>{{ oferta.compras }}</td>
</tr>
{% else %}
<tr>
```

```
    <td colspan="5">Esta tienda no ha publicado ninguna oferta</td>
</tr>
{% endfor %}
```

CAPÍTULO 8

Registrando usuarios

El componente de seguridad de Symfony es el más potente y a la vez complejo de los que se incluyen en el *framework*. Internamente se divide en dos partes: autenticación y autorización. La **autenticación** consiste en demostrar que eres quien dices ser (por ejemplo con un usuario + contraseña, certificado X.509, token de API, etc.) La **autorización** consiste en decidir si tienes permiso para hacer lo que quieres hacer (por ejemplo acceder a una zona restringida del sitio web, borrar un contenido, etc.)

En el capítulo anterior se desarrollaron todas las páginas públicas del sitio. En las próximas secciones se crean el resto de páginas que faltan para completar el *frontend*. Primero se crea la página que muestra las compras recientes de un usuario y después se restringe su acceso para que sólo puedan verla los usuarios *logueados*. Por último, se crea el formulario de registro y la página con los datos del perfil del usuario.

8.1 Creando la página de compras recientes

La página con las compras recientes de un usuario se muestra al pinchar sobre el enlace *Mis ofertas* del menú principal de navegación. La página muestra un listado de las últimas ofertas compradas por el usuario. La información que se muestra para cada oferta es la misma que la de la página *Ofertas recientes*.

Como es habitual, para desarrollar esta funcionalidad se va a crear una nueva acción en un controlador, una nueva consulta en un repositorio de Doctrine y por último, una nueva plantilla Twig.

La página de compras recientes está relacionada con los usuarios, así que se empieza creando una acción llamada `comprasAction()` en el controlador `UsuarioController`:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * @Route("/usuario")
 */
class UsuarioController extends Controller
{
    /**
     * @Route("/compras", name="usuario_compras")
     */
```

```
    public function comprasAction()
    {
        $usuarioId = 1;

        $em = $this->getDoctrine()->getManager();
        $compras = $em->getRepository('AppBundle:Usuario')
                      ->findTodasLasCompras($usuarioId);

        return $this->render('usuario/compras.html.twig', array(
            'compras'  => $compras
        ));
    }

    // ...
}
```

La acción `comprasAction()` busca las últimas compras del usuario *logueado* en la aplicación y pasa la información a la plantilla correspondiente. Como todavía no es posible *loguear* usuarios en la aplicación, se considera que el usuario *logueado* siempre es aquél cuyo `id` es `1`. Más adelante se explica cómo obtener el usuario *logueado*.

Aunque la consulta a la base de datos es muy sencilla, se va a seguir la buena práctica de definir para ella un método dentro del repositorio de la entidad. Por tanto, modifica primero el código de la entidad `Usuario` para indicar que utiliza un repositorio propio:

```
// src/AppBundle/Entity/Usuario.php
// ...

/**
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UsuarioRepository")
 */
class Usuario
{
    // ...
}
```

A continuación, crea el repositorio `UsuarioRepository` y añade el código de la consulta `findTodasLasCompras()`:

```
// src/AppBundle/Repository/UsuarioRepository.php
namespace AppBundle\Repository;
use Doctrine\ORM\EntityRepository;

class UsuarioRepository extends EntityRepository
{
    public function findTodasLasCompras($usuario)
    {
        $em = $this->getEntityManager();
```

```
        $consulta = $em->createQuery('
            SELECT v, o, t
              FROM AppBundle:Venta v
              JOIN v.oferta o
              JOIN o.tienda t
             WHERE v.usuario = :id
          ORDER BY v.fecha DESC');
        $consulta->setParameter('id', $usuario);

        return $consulta->getResult();
    }
}
```

La búsqueda obtiene como resultado un array de objetos de tipo `Venta` (para evitar el *lazy loading* de Doctrine, se hace un `JOIN` para incluir la información de las ofertas y de las tiendas).

Para finalizar la página, crea la plantilla `usuario/compras.html.twig` y añade el siguiente código:

```
{# app/Resources/views/usuario/compras.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Últimas ofertas que has comprado{% endblock %}
{% block id 'compras' %}

{% block article %}
    <h1>{{ block('title') }}</h1>

    {% for compra in compras %}
        {{ include('oferta/_oferta_simple.html.twig') }}
    {% endfor %}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

Como el listado de ofertas incluye la misma información que el de la página *Ofertas recientes*, se puede reutilizar la plantilla `_oferta_simple.html.twig`. No obstante, si pruebas la página en el navegador (`http://127.0.0.1:8000/app_dev.php/usuario/compras/`) verás un mensaje de error.

Recuerda que al utilizar la función `{{ include() }}` la plantilla incluida tiene acceso a todas las variables de la plantilla principal. El problema es que `_oferta_simple.html.twig` necesita que la información de la oferta se encuentre en una variable llamada `oferta` y la plantilla utiliza una variable llamada `compra` con los datos de una entidad de tipo `Venta`.

La solución consiste en pasar explícitamente a la función `{{ include() }}` una variable llamada `oferta`:

```
{# app/Resources/views/usuario/compras.html.twig #}

{# ...#}

{% for compra in compras %}
    {{ include('oferta/_oferta_simple.html.twig', { oferta: compra.oferta }) }}
{% endfor %}

{# ... #}
```

Vuelve a cargar la página y ahora sí que verás correctamente el listado de las compras más recientes del usuario. Si no ves ninguna compra, es posible que el usuario utilizado no tenga ninguna, ya que las compras se generan aleatoriamente al cargar los archivos de datos o *fixtures*. Solucionalo volviendo a cargar los *fixtures* o cambiando el valor de la variable `$usuarioId` en la acción `comprasAction()`.

Antes de dar por completada esta página, es posible mejorar el código de las plantillas. Aunque la misma plantilla `_oferta_simple.html.twig` se reutiliza en varias plantillas, su contenido podría variar ligeramente para adaptarse mejor a cada caso:

- En la plantilla *Ofertas recientes*, se muestran botones *Comprar* en todas las ofertas que todavía se puedan comprar. En el resto de ofertas se muestra la fecha de expiración con el formato *"Finalizada el ..."*
- En la plantilla *Mis ofertas*, nunca se muestra el botón *Comprar*, puesto que todas las ofertas ya se han comprado. Además, se debe mostrar la fecha de compra, no la fecha de publicación o expiración de las ofertas.

Abre el código de la plantilla `_oferta_simple.html.twig` y define en primer lugar una variable llamada `comprada` cuyo valor se obtenga de la plantilla principal y cuyo valor por defecto sea `false`:

```
{# app/Resources/views/oferta/_oferta_simple.html.twig #}
{% set expirada = oferta.fechaExpiracion|date('YmdHis') < 'now'|date('YmdHis')
%}
{% set comprada = comprada|default(false) %}
{# ... #}
```

Después de definir esta nueva variable, el primer cambio consiste en mostrar el botón solamente para las ofertas que no se hayan comprado (y que tampoco hayan expirado):

```
{# app/Resources/views/oferta/_oferta_simple.html.twig #}

{# ... #}
{% if not expirada and not comprada %}
    <a class="boton" href="#">Comprar</a>
{% endif %}
{# ... #}
```

El siguiente cambio consiste en mostrar la fecha adecuada en cada caso: fecha de compra para las ofertas compradas, fecha de expiración para las ofertas no compradas que hayan expirado y una cuenta atrás para las ofertas no compradas que todavía se puedan comprar:

```
{# app/Resources/views/oferta/_oferta_simple.html.twig #}

{# ... #}
<div class="estado {{ expirada ? 'expirada' }} {{ comprada ? 'comprada' }}">
    {% if comprada %}
        Comprada el {{ fechaCompra|date('d/m/Y') }}
    {% elseif not expirada %}
        <strong>Faltan</strong>: {{ oferta.fechaExpiracion|cuenta_atras }}
    {% else %}
        Finalizada el {{ oferta.fechaExpiracion|date('d/m/Y') }}
    {% endif %}
</div>
{# ... #}
```

Después de estos cambios, la página *Ofertas recientes* sigue funcionando correctamente sin tener que modificarla, pero no ocurre lo mismo con la página *Mis ofertas*. Para que esta última siga funcionando, es necesario que cree las variables `oferta`, `comprada` y `fechaCompra` para cada una de las compras del usuario. Esto es muy sencillo gracias al paso de variables de la función `{{ include() }}`:

```
{# app/Resources/views/usuario/compras.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Últimas ofertas que has comprado{% endblock %}
{% block id 'compras' %}

{% block article %}
    <h1>{{ block('title') }}</h1>

    {% for compra in compras %}
        {{ include('oferta/_oferta_simple.html.twig', {
            oferta: compra.oferta,
            comprada: true,
            fechaCompra: compra.fecha
        }) }}
    {% endfor %}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

8.2 Restringiendo el acceso

La página desarrollada en la sección anterior sólo puede ser accedida por usuarios registrados que se hayan *logueado* en la aplicación. Así, cuando un usuario anónimo trata de acceder a esa página, se le redirige al formulario de *login*.

La configuración básica de la seguridad de las aplicaciones Symfony se define en el archivo de configuración `app/config/security.yml`. Abre ese archivo, borra todos sus contenidos y copia la siguiente configuración:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            pattern:     ^/*
            provider:    usuarios
            anonymous:   ~
            form_login:
                login_path: usuario_login
                check_path: usuario_login_check
            logout:
                path:       usuario_logout

    access_control:
        - { path: ^/usuario/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/registro, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/*, roles: ROLE_USUARIO }

    providers:
        usuarios:
            entity: { class: AppBundle\Entity\Usuario, property: email }

    encoders:
        AppBundle\Entity\Usuario: bcrypt
```

Si tratas de acceder ahora a la página `/usuario/compras`, ya no verás el listado de las últimas compras sino una página de error. Para llegar a ver el formulario de *login*, tendrás que hacer algún cambio más en el código de la aplicación, tal y como se explica en las siguientes secciones.

Volviendo a la configuración del archivo `security.yml`, la primera sección importante se define bajo la clave `firewalls`:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            pattern:    ^/*
            provider:   usuarios
            anonymous:  ~
```

```
            form_login: ~

    # ...
```

Según la terminología de Symfony, un *firewall* es el mecanismo con el que se protegen las diferentes partes de un sitio web. Cada aplicación puede definir tantos *firewalls* como necesite, siempre que asigne un nombre único a cada uno. El *firewall* anterior se denomina `frontend` porque está configurado para proteger las zonas restringidas del `frontend`.

La función de cada *firewall* consiste en comprobar si la URL solicitada por el usuario se encuentra bajo su protección. Más técnicamente, si la URL solicitada coincide con la expresión regular definida en su opción `pattern`, se activa el mecanismo de autenticación asociado al *firewall*.

El patrón del *firewall* `frontend` es simplemente `^/*`, por lo que todas las URL de la aplicación se encuentran protegidas por este *firewall*. Esto significa que cualquier usuario que acceda a cualquier página del sitio debería *loguearse*. Para evitarlo, añade la opción `anonymous: ~` que permite el acceso a los usuarios anónimos.

La última opción de configuración del *firewall* es el tipo de autenticación que utilizan los usuarios. Los tipos de autenticación más utilizados son:

- `http_basic` y `http_digest`, que hacen que el navegador muestre la típica caja en la que se solicita usuario y contraseña.
- `x509`, que solicita al usuario un certificado de tipo X.509 (como por ejemplo el que contiene el DNI electrónico de España).
- `form_login`, que muestra una página con un formulario para introducir el usuario y la contraseña.

A partir de la versión 2.8, Symfony ha añadido otros mecanismos de autenticación como LDAP y Guard. Lee la referencia de configuración de seguridad (http://symfony.com/doc/2.8/reference/configuration/security.html) de Symfony para conocer todos ellos.

La segunda sección del archivo `security.yml` configura las opciones de autorización:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/usuario/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/registro, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/*, roles: ROLE_USUARIO }
```

La opción `access_control` indica qué tipo de usuarios pueden acceder a cada tipo de URL. Añade lo siguiente para que las URL que empiezan por `/usuario` sólo puedan ser accedidas por usuarios *logueados*:

```
security:
    access_control:
        - { path: ^/usuario/*, roles: ROLE_USUARIO }
```

La opción `path` es la expresión regular que indica las URL a las que se aplica esta configuración. La opción `roles` indica de qué tipo debe ser el usuario para poder acceder.

Si ahora se unen las opciones de configuración `firewalls` y `access_control`:

```
security:
    firewalls:
        frontend:
            pattern:     ^/*
            provider:    usuario
            anonymous:   ~
            form_login:  ~
    access_control:
        - { path: ^/usuario/*, roles: ROLE_USUARIO }
```

Esta configuración indica que cuando un usuario trata de acceder por ejemplo a la URL `/usuario/compras`, se le muestra un formulario de *login*. La opción `access_control` es la parte de **autorización** que impide el acceso a los usuarios normales (ya que es obligatorio que sean de tipo `ROLE_USUARIO`). La opción `form_login` del *firewall* `frontend` es la parte de **autenticación**, que se encarga de mostrar el formulario de *login* para que el usuario pueda demostrar quién es.

El tipo de usuario, denominado *rol*, se define mediante la opción `roles`. Cuando esta opción incluye más de un rol, el usuario debe tener al menos uno de ellos:

```
security:
    access_control:
        - { path: ^/usuario/*, roles: [ROLE_USUARIO, ROLE_ADMIN] }
```

El nombre del *rol* se puede elegir libremente, siempre que empiece por `ROLE_`. En las próximas secciones se explica cómo definir diferentes tipos de usuarios en la aplicación.

Un error común al configurar la autorización es impedir el acceso al propio formulario de *login*. En la configuración anterior, todas las rutas `/usuario/*` obligan a que el usuario esté *logueado*, pero como el formulario de *login* se encuentra en `/usuario/login`, nadie podrá *loguearse*.

Para evitar este problema, añade excepciones para todas aquellas URL que tengan que ser accedidas por todos los usuarios. Para ello puedes hacer uso de un *rol* especial llamado `IS_AUTHENTICATED_ANONYMOUSLY` que define Symfony para referirse a los usuarios anónimos. En nuestro caso, además del formulario de *login*, es necesario permitir el acceso al formulario de registro que se creará próximamente:

```
security:
    access_control:
        - { path: ^/usuario/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

```
        - { path: ^/usuario/registro, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/*, roles: ROLE_USUARIO }
```

Ten en cuenta que Symfony respeta el orden en el que añades las rutas de `access_control`, por lo que las excepciones siempre se definen por delante de la regla general.

La tercera sección del archivo `security.yml` configura los proveedores, que son los encargados de *crear* y buscar los usuarios de la aplicación:

```
# app/config/security.yml
security:
    # ...
    providers:
        usuarios:
            entity: { class: AppBundle\Entity\Usuario, property: email }
```

Los dos tipos de proveedores más populares son: `memory` (los usuarios se crean en memoria con los datos definidos en el propio archivo `security.yml`) y `entity` (los usuarios se crean mediante entidades de Doctrine). Por tanto, la configuración anterior indica que los usuarios se crean a partir de la entidad `Usuario` y el nombre de usuario será la propiedad `email` de la entidad.

Se pueden definir varios proveedores e incluso encadenarlos, como muestra el siguiente ejemplo:

```
# app/config/security.yml
security:
    # ...
    providers:
        todos:
            chain:
                providers: [usuarios, tiendas]
        usuarios:
            entity: { class: AppBundle\Entity\Usuario, property: email }
        tiendas:
            entity: { class: AppBundle\Entity\Tienda, property: login }
```

Si empleas el proveedor llamado `todos`, Symfony buscará al usuario *logueado* primero entre los usuarios de tipo `Usuario` y si no lo encuentra, lo buscará después entre los usuarios de tipo `Tienda`.

La última sección del archivo `security.yml` configura cómo se codifican las contraseñas de los usuarios. Esta configuración es obligatoria para cada proveedor definido en la opción `providers`.

El tipo de codificación más simple es `plaintext`, que guarda las contraseñas en claro. Obviamente es absurdo utilizarlo en producción, pero puede ser muy útil cuando se está empezando a desarrollar la aplicación:

```
security:
    # ...
    encoders:
        AppBundle\Entity\Usuario: plaintext
```

El tipo de codificación recomendado para las aplicaciones web reales se llama `bcrypt`:

```
security:
    # ...
    encoders:
        AppBundle\Entity\Usuario: bcrypt
```

La codificación `bcrypt` ha sido diseñada a propósito para que cueste mucho tiempo codificar las contraseñas. De esta forma se evitan los ataques de fuerza bruta porque es imposible codificar miles de contraseñas por segundo.

Sin embargo, como los ordenadores son cada vez más potentes, lo que hoy cuesta mucho tiempo, dentro de unos años no costará nada. Por eso la codificación `bcrypt` define una opción de configuración llamada `cost` que modifica el algoritmo para que codificar una contraseña cueste más o menos. Su valor por defecto es `10` y puede tomar cualquier valor entre `4` y `31`:

```
security:
    # ...
    encoders:
        # '6' hará que la codificación sea muy rápida ... pero insegura
        AppBundle\Entity\Usuario: { algorithm: 'bcrypt', cost: 6 }
        # '25' hará que la codificación sea muy segura ... pero lentísima
        AppBundle\Entity\Usuario: { algorithm: 'bcrypt', cost: 25 }
```

8.3 Creando proveedores de usuarios

La configuración de seguridad anterior establece que los usuarios de la aplicación se crean a partir de la entidad `Usuario`. No obstante, para que una entidad se convierta en proveedor de usuarios, es necesario realizar algunos cambios en su código.

En concreto, las entidades que son proveedores de usuarios deben implementar la interfaz `UserInterface`. Así que edita la entidad `AppBundle\Entity\Usuario` y añade la instrucción `implements` correspondiente (no olvides tampoco importar la interfaz `UserInterface` mediante la instrucción `use`):

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Usuario implements UserInterface
{
    ...
}
```

Implementar la interfaz `UserInterface` obliga a definir cinco métodos en la entidad:

- `eraseCredentials()`, se invoca cuando la aplicación necesita borrar la información más sensible del usuario (como por ejemplo su contraseña) antes de serializar la información del usuario para guardarla.
- `getPassword()`, se invoca cada vez que la aplicación necesita obtener la contraseña del usuario.
- `getRoles()`, cuando se autentica a un usuario, se invoca este método para obtener un array con todos los *roles* que posee.
- `getSalt()`, devuelve el valor que se utilizó para aleatorizar la contraseña cuando se creó el usuario. Se invoca siempre que la aplicación necesita comprobar la contraseña del usuario. Si utilizas la codificación `bcrypt`, puedes dejar este método vacío porque la contraseña ya incluye su propio valor aleatorio, por lo que el valor `salt` siempre es `null`.
- `getUsername()`, se invoca para obtener el *login* o nombre de usuario que se utiliza para autenticar a los usuarios. De esta forma se puede utilizar cualquier propiedad de la entidad como *login*, como por ejemplo su email.

TRUCO Symfony también incluye otra interfaz más avanzada llamada `AdvancedUserInterface` que extiende de `UserInterface` y añade varios métodos muy útiles para las aplicaciones que necesitan gestionar el estado de las cuentas de los usuarios. Los métodos que añade esta otra interfaz son `isAccountNonExpired()` (permite por ejemplo expirar las cuentas de los usuarios que ya no pagan por el servicio), `isAccountNonLocked()` (permite bloquear a aquellos usuarios que no han pinchado en el email de confirmación del registro), `isCredentialsNonExpired()` (permite obligar a los usuarios a modificar su contraseña), `isEnabled()` (permite bloquear a los usuarios por cualquier otro motivo).

Siguiendo la explicación anterior, ya puedes modificar la entidad `Usuario` para añadir los métodos que requiere la interfaz:

```
// src/AppBundle/Entity/Usuario.php
class Usuario implements UserInterface
{
    function getRoles()
    {
        return array('ROLE_USUARIO');
    }

    function getUsername()
    {
        return $this->getEmail();
    }

    function eraseCredentials()
    {
```

```
        $this->password = null;
    }

    function getSalt()
    {
        // las contraseñas se codifican con 'bcrypt', por lo que no
        // es necesario definir el valor del 'salt'
        return null;
    }

  // ...
}
```

El método `getRoles()` devuelve un array con un único valor (`ROLE_USUARIO`) porque todos los usuarios del *frontend* son del mismo tipo. Aunque en las aplicaciones Symfony es habitual utilizar los *roles* `ROLE_USER`, `ROLE_ADMIN` y `ROLE_SUPER_ADMIN`, el nombre del *rol* se puede elegir libremente, por eso se utiliza el nombre `ROLE_USUARIO`.

Si tu aplicación web dispone de varios tipos de usuarios en el *frontend*, lo correcto es añadir una nueva propiedad a la entidad `Usuario` en la que se almacene el *rol* de cada usuario. Así, si tu aplicación tiene diferentes niveles de uso en función de lo que paga el usuario, podrías utilizar los *roles* `ROLE_USER_FREE`, `ROLE_USER_PLUS`, `ROLE_USER_PREMIUM`, etc.

El último método añadido es `getUsername()` que devuelve el nombre de usuario que utilizan los usuarios para hacer el *login* (en este caso, su email).

De los cinco métodos requeridos por la interfaz solamente se han definido cuatro. La razón es que el método `getPassword()` ya estaba definido por ser el *getter* de la propiedad `password` de la entidad.

Después de estos cambios, la aplicación ya está configurada para impedir el acceso a las páginas privadas del *frontend* a cualquier usuario que no esté *logueado*. Además, Symfony también sabe cómo crear usuarios y cómo comprobar que las credenciales son las correctas. Así que el último paso antes de poder probar la seguridad consiste en crear el formulario de *login*.

8.4 Añadiendo el formulario de *login*

El proceso de *login* mediante un formulario en Symfony está asociado con tres rutas:

- `/login`, se utiliza para mostrar el formulario de *login*.
- `/login_check`, es la acción que comprueba que el usuario y contraseña introducidos son correctos.
- `/logout`, se emplea para desconectar al usuario *logueado*.

Symfony se encarga de gestionar el `login_check` y del `logout` automáticamente, por lo que no es necesario que añadas ningún código en sus acciones. La acción que sí debes implementar es la asociada al `login`.

Abre el controlador de los usuarios y añade los siguientes tres métodos con sus rutas correspondientes:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * @Route("/usuario")
 */
class UsuarioController extends Controller
{
    /**
     * @Route("/login", name="usuario_login")
     */
    public function loginAction()
    {
        // crear aquí el formulario de login ...
    }

    /**
     * @Route("/login_check", name="usuario_login_check")
     */
    public function loginCheckAction()
    {
        // el "login check" lo hace Symfony automáticamente, por lo que
        // no hay que añadir ningún código en este método
    }

    /**
     * @Route("/logout", name="usuario_logout")
     */
    public function logoutAction()
    {
        // el logout lo hace Symfony automáticamente, por lo que
        // no hay que añadir ningún código en este método
    }

    // ...
}
```

Recuerda que la clase `UsuarioController` define la anotación `@Route("/usuario")`, por lo que las rutas de todas sus acciones incluirán el prefijo `/usuario`. Así que en realidad, los patrones completos de las rutas anteriores son: `/usuario/login`, `/usuario/login_check` y `/usuario/logout`.

Symfony espera que la ruta de *login* sea `/login`, pero en este caso es `/usuario/login`, por lo que debes actualizar la configuración de seguridad. Para ello, indica explícitamente las rutas en el *firewall* `frontend` mediante las opciones `login_path` y `check_path`:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            form_login:
                login_path: /usuario/login
                check_path: /usuario/login_check
```

En lugar de las URL, también puedes indicar el nombre de las rutas:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            form_login:
                login_path: usuario_login
                check_path: usuario_login_check
```

A continuación, añade el siguiente código en el método `loginAction()` para renderizar el formulario de login que se muestra al usuario:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * @Route("/usuario")
 */
class UsuarioController extends Controller
{
    /**
     * @Route("/login", name="usuario_login")
     */
    public function loginAction()
    {
        $authUtils = $this->get('security.authentication_utils');

        return $this->render('usuario/login.html.twig', array(
            'last_username' => $authUtils->getLastUsername(),
            'error' => $authUtils->getLastAuthenticationError(),
        ));
```

```
    }

    // ...
}
```

La clave para entender el código anterior es el servicio `security.authentication_utils`, que proporciona métodos para obtener información relacionada con el proceso de login. De esta manera se obtiene el último error de login producido (`getLastAuthenticationError()`) y el último nombre de usuario que utilizó el usuario para intentar el login (`getLastUsername()`).

Aunque no es obligatorio obtener esta información y pasarla al formulario de login, si lo haces y se produce algún error (porque el usuario ha escrito mal su contraseña, porque hay un error en el sistema, etc.) al usuario podrás mostrarle un mensaje de error adecuado.

La plantilla necesaria para mostrar el formulario de *login* es la siguiente:

```
{# app/Resources/views/usuario/login.html.twig  #}
{% extends 'frontend.html.twig' %}

{% block title %}Formulario de acceso{% endblock %}
{% block id 'usuario' %}

{% block article %}
    {% if error %}
        <div>{{ error.message }}</div>
    {% endif %}

    <form action="{{ path('usuario_login_check') }}" method="post">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="_username"
               value="{{ last_username }}" />

        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="_password" />

        <input type="submit" name="login" value="Entrar" />
    </form>
{% endblock %}

{% block aside %}{% endblock %}
```

Puedes utilizar cualquier plantilla para mostrar el formulario de *login* siempre que cumplas las siguientes condiciones:

- El atributo `action` del formulario es la ruta que apunta a la acción `loginCheckAction()`. No es necesario añadir código en esta acción porque Symfony intercepta el envío del formulario y se encarga de comprobar el usuario y contraseña.

- El campo del nombre de usuario debe tener un atributo `name` igual a `_username` (incluyendo el guión bajo por delante).
- El campo de la contraseña debe tener un atributo `name` igual a `_password` (incluyendo el guión bajo por delante).

NOTA Si por cualquier motivo los formularios de *login* de tu aplicación no pueden utilizar los valores `_username` o `_password` como atributo `name`, puedes cambiar sus valores con las opciones `username_parameter` y `password_parameter` de la opción `form_login` del *firewall*.

Además de la página del formulario de *login*, en el lateral de todas las páginas del *frontend* se muestra una pequeña caja para que el usuario pueda *loguearse*:

Figura 8.1 Caja de login que muestra el lateral de todas las páginas

Para crear este nuevo formulario, crea una plantilla llamada `_caja_login.html.twig`:

```
{# app/Resources/views/usuario/_caja_login.html.twig #}
<h2>Accede a tu cuenta</h2>

<form action="{{ path('usuario_login_check') }}" method="post">
    {% if error %}
        <div>{{ error.message }}</div>
    {% endif %}

    <label for="login_user">Email</label>
    <input id="login_user" type="text" name="_username"
           value="{{ last_username|default('') }}" />

    <label for="login_pass">Contraseña</label>
    <input id="login_pass" type="password" name="_password" />

    <input type="submit" value="Entrar" />
</form>
```

Como la caja de *login* se muestra en todas las páginas del *frontend*, lo más adecuado es añadirlo en la plantilla `frontend.html.twig` de la que heredan todas las demás plantillas:

```
{# app/Resources/views/frontend.html.twig #}
{% extends 'base.html.twig' %}

{# ... #}

{% block body %}

{# ... #}

<aside>
    {% block aside %}
    <section id="login">
        {{ render(controller('AppBundle:Usuario:cajaLogin')) }}
    </section>
    {% endblock %}
</aside>
{% endblock %}
```

Para que esta plantilla pueda funcionar, no olvides añadir la acción `cajaLoginAction()` en el controlador `UsuarioController`. Puedes utilizar para ello el mismo código que el de la acción `loginAction()` anterior.

Con estos últimos cambios, ya puedes probar a *loguearte* en la aplicación utilizando cualquiera de los formularios de *login* y con los datos de cualquier usuario de prueba creado con los archivos de datos o *fixtures*.

Si lo haces, verás que Symfony muestra un error. El motivo es que los usuarios de prueba guardan su contraseña en claro en la base de datos y Symfony codifica la contraseña del formulario de *login* mediante `bcrypt`.

Hasta que no se actualicen los archivos de datos o *fixtures* más adelante, la única solución para poder probar el *login* consiste en modificar la configuración de seguridad para que las contraseñas de los usuarios no se codifiquen (esto es una solución temporal; no lo hagas nunca en tus aplicaciones reales):

```
# app/config/security.yml
security:
    # ...
    encoders:
        AppBundle\Entity\Usuario: plaintext
```

Si tratas de acceder ahora a la página `/usuario/compras`, se mostrará el formulario de *login* y si introduces el email y contraseña de cualquier usuario de prueba, verás sus compras más recientes.

8.4.1 Manteniendo a los usuarios conectados

Resulta habitual que los formularios de *login* de las aplicaciones incluyan la opción *"No cerrar la sesión"*. Esta opción, conocida en inglés como *Remember Me*, hace que el usuario no tenga que volver a *loguearse* la próxima vez que visite el sitio web.

El componente de seguridad de Symfony ya incluye soporte para esta opción, por lo que solo tendrás que activarla mediante la opción `remember_me` del *firewall*:

```
# app/config/security.yml
firewalls:
    frontend:
        # ...
        form_login:
            # ...
            remember_me: true
```

Por defecto el usuario se recuerda durante una hora. Si quieres cambiar este valor o cualquier otro relacionado con esta opción, lee la referencia de la configuración de seguridad (http://symfony.com/doc/2.8/reference/configuration/security.html) de Symfony.

Después de añadir la opción `remember_me`, modifica los formularios de *login* para incluir la opción que permite a los usuarios decidir si quieren permanecer conectados:

```
{# app/Resources/views/usuario/login.html.twig #}

{# ... #}
<form action="{{ path('usuario_login_check') }}" method="post">
    {# ... #}

    <input type="checkbox" id="no_cerrar" name="_remember_me" checked />
    <label for="no_cerrar">No cerrar sesión</label>

    <input type="submit" name="login" value="Acceder" />
</form>
{# ... #}
```

```
{# app/Resources/views/usuario/_caja_login.html.twig #}
<h2>Accede a tu cuenta</h2>

<form action="{{ path('usuario_login_check') }}" method="post">
    {# ... #}

    <input type="submit" value="Entrar" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">No cerrar sesión</label>
</form>
```

La opción *remember me* se debe incluir como un campo de tipo `checkbox` y su atributo `name` debe ser `_remember_me` (con el guión bajo por delante).

8.4.2 Ejecutando código después del *login*

Cuando el usuario se *loguea* correctamente, la aplicación debería redirigirle a la portada de la ciudad con la que está asociado. Sin embargo, ahora mismo Symfony redirige a la misma página que

solicitó el usuario antes de que saltara el formulario de *login*. Este comportamiento se debe a la opción `use_referer`, que por defecto vale `true` y hace que se redirija al usuario a la página anterior al formulario de login:

```
security:
    firewalls:
        frontend:
            # ...
            form_login:
                use_referer: true
```

Cuando la aplicación no puede obtener la página anterior o cuando el usuario accede directamente al formulario de login, se le redirige a la portada. Este comportamiento se puede modificar con la opción `default_target_path`, que indica la ruta a la que se redirige al usuario en las situaciones que se acaban de describir:

```
security:
    firewalls:
        frontend:
            # ...
            form_login:
                default_target_path: /usuario/compras
```

Si quieres redirigir a todos los usuarios a la misma página, puedes combinar esta opción con `always_use_default_target_path`:

```
security:
    firewalls:
        frontend:
            # ...
            form_login:
                default_target_path: /usuario/compras
                always_use_default_target_path:  true
```

Otra opción para redirigir al usuario a una página determinada después del *login* es definir un campo llamado `_target_path` en el formulario de *login*:

```
{# src/AppBundle/Resources/views/Default/_caja_login.html.twig #}
<form action="{{ path('usuario_login_check') }}" method="post">
    {# ... #}

    <input type="submit" value="Entrar" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">No cerrar sesión</label>

    <input type="hidden" name="_target_path" value="/usuario/compras" />
</form>
```

En cualquier caso, ninguna de las opciones anteriores sirve para redirigir a cada usuario a la portada de su ciudad. La única solución posible consiste en utilizar el sistema de eventos de Symfony para ejecutar código justo después de que el usuario haga *login*.

> **NOTA** Si todavía no lo has hecho, ha llegado el momento de leer el apéndice B (página 505) dedicado a la inyección de dependencias. Si no lo haces, te resultará muy difícil entender el código de los siguientes ejemplos.

Symfony utiliza el patrón de diseño *Observer* (http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)) para la gestión de sus eventos. Durante la ejecución de la aplicación, Symfony notifica diversos **eventos** (ejemplo: "ha llegado una nueva petición", "voy a ejecutar este controlador", etc.) Si nadie está "escuchando" estos eventos, no sucederá nada en la aplicación. Si alguien los está escuchando, se puede ejecutar código como respuesta a los eventos que se han producido.

Los "escuchadores" se llaman *listeners* y *subscribers* en inglés. Se trata de clases PHP que se pueden "suscribir" a uno o más eventos de Symfony para que sean notificados cada vez que se produzcan.

Así por ejemplo, el evento relacionado con el *login* del usuario se llama `security.interactive_login` y se notifica justo después de que el usuario introduzca su usuario y contraseña y pulse el botón *Acceder*.

El primer paso consiste en indicar a Symfony nuestro interés por este evento. Para ello se define un nuevo servicio asociado con la etiqueta `kernel.event_listener`, para que Symfony sepa que se trata de un *listener*. Además, debes indicar el nombre del evento al que te quieres suscribir mediante la opción `event` de la etiqueta:

```
# app/config/services.yml
services:
    app.login_listener:
        class: AppBundle\Listener\LoginListener
        tags:
            - { name: kernel.event_listener, event: security.interactive_login }
```

Por convención, las clases que actúan de *listeners* se definen dentro del directorio `Listener/` del *bundle*. Así que crea ese directorio a mano, y añade en su interior una clase llamada `LoginListener.php` con el siguiente código:

```
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
```

```
    }
}
```

El mecanismo encargado de notificar los eventos (llamado *dispatcher*) entrega a cada *listener* un objeto llamado `$event` con información útil sobre el propio evento y acceso a varios objetos de la aplicación. Según el tipo de evento, el objeto que se pasa es de un tipo diferente. Para el evento `security.interactive_login` el objeto es de tipo `InteractiveLoginEvent`, que incluye dos métodos: `getRequest()` para obtener el objeto de la petición y `getAuthenticationToken()` para obtener el *token* asociado con el usuario que acaba de hacer *login*.

Así que si por ejemplo quisieras guardar la fecha y hora a la que se conecta cada usuario, podrías hacerlo mediante el siguiente código (la entidad `Usuario` debería contener una propiedad llamada `ultimaConexion` para que el siguiente código funcione):

```
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
        $usuario = $event->getAuthenticationToken()->getUser();
        $usuario->setUltimaConexion(new \DateTime());
    }
}
```

El evento `InteractiveLoginEvent` no proporciona acceso al objeto `Response` de la petición, por lo que no es posible redirigir al usuario a ninguna página. El evento que sí proporciona este acceso es `kernel.response`, por lo que el *listener* también tiene que suscribirse a ese evento:

```
# app/config/services.yml
services:
    app.login_listener:
        class: AppBundle\Listener\LoginListener
        tags:
            - { name: kernel.event_listener, event: security.interactive_login }
            - { name: kernel.event_listener, event: kernel.response }
```

Ahora el *listener* puede utilizar el evento `security.interactive_login` para establecer el valor de una variable llamada `$ciudad`, mientras que el evento `kernel.response` puede encargarse de redirigir al usuario si la variable `$ciudad` está definida:

```
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
```

```
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    private $ciudad = null;

    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
        $token = $event->getAuthenticationToken();
        $this->ciudad = $token->getUser()->getCiudad()->getSlug();
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        if (null === $this->ciudad) {
            return;
        }

        $urlPortada = ...
        $event->setResponse(new RedirectResponse($urlPortada));
    }
}
```

Para generar la ruta de la portada es necesario acceder al sistema de enrutamiento de la aplicación. Así que al definir el servicio `app.login_listener`, debes inyectar el servicio `router` como argumento:

```
# app/config/services.yml
services:
    app.login_listener:
        class: AppBundle\Listener\LoginListener
        arguments: ['@router']
        tags:
            - { name: kernel.event_listener, event: security.interactive_login }
            - { name: kernel.event_listener, event: kernel.response }
```

Ahora añade un constructor en el *listener* para recoger el argumento que le pasa Symfony al crear el servicio y genera la ruta de la portada de cada usuario:

```
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\Routing\Router;
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
```

```
    private $router, $ciudad = null;

    public function __construct(Router $router)
    {
        $this->router = $router;
    }

    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
        $token = $event->getAuthenticationToken();
        $this->ciudad = $token->getUser()->getCiudad()->getSlug();
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        if (null === $this->ciudad) {
            return;
        }

        $urlPortada = $this->router->generate('portada', array(
            'ciudad' => $this->ciudad
        ));
        $event->setResponse(new RedirectResponse($urlPortada));
    }
}
```

El *listener* que se acaba de desarrollar afecta a todas las partes de la aplicación y a todos los tipos de usuario, ya que no se controla de ninguna manera su alcance. Si estás aplicando este código en tu propia aplicación o si estás desarrollando libremente la aplicación Cupon, debes tener en cuenta que este código se refactoriza más adelante en la sección *Refactorizando el evento asociado al login* (página 332) para que funcione bien con otros tipos de usuarios.

8.5 Modificando las plantillas

Cuando un usuario accede a la aplicación estando *logueado*, resulta habitual que el contenido de algunas plantillas varíe. Así por ejemplo, la zona lateral ya no muestra un formulario de *login* sino el nombre del usuario y un enlace para desconectarse. Pero antes de modificar las plantillas, es necesario obtener el usuario y determinar su tipo o *rol*.

8.5.1 Obteniendo el usuario *logueado*

Dentro de un controlador, el usuario se obtiene a través del atajo `getUser()`:

```
class DefaultController extends Controller
{
    public function defaultAction()
    {
        $usuario = $this->getUser();
        $nombre = $usuario->getNombre();
```

```
        // ...
    }
}
```

El método `getUser()` es en realidad un atajo del siguiente código que obtiene en primer lugar el *token* del usuario *logueado* y después obtiene el objeto de tipo `Usuario` que representa al usuario:

```
$usuario = $this->container->get('security.token_storage')->getToken()->getUse
r();
```

Dentro de una plantilla, el usuario *logueado* se obtiene a través de la propiedad `user` de la variable global `app` creada por Symfony:

```
{% set usuario = app.user %}
Nombre: {{ usuario.nombre }}

{# Código equivalente y más conciso #}
Nombre: {{ app.user.nombre }}
```

8.5.2 Determinando el tipo de usuario *logueado*

Si el contenido a mostrar depende del tipo de usuario, también se debe comprobar si el usuario dispone de un determinado *rol*. Dentro de un controlador utiliza el método `isGranted()`:

```
class DefaultController extends Controller
{
    public function defaultAction()
    {
        if ($this->isGranted('ROLE_USUARIO')) {
            // el usuario tiene el role 'ROLE_USUARIO'
        } elseif ($this->isGranted('ROLE_ADMIN')) {
            // el usuario tiene el role 'ROLE_ADMIN'
        }

        // ...
    }
}
```

El método `isGranted()` es un atajo del siguiente código PHP:

```
$checker = $this->container->get('security.authorization_checker');
if ($checker->isGranted('ROLE_USUARIO')) {
    // ...
}
```

Si el usuario no dispone de la autorización necesaria, puedes mostrar una página de error lanzando la excepción `AccessDeniedException`:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

class DefaultController extends Controller
```

```
{
    public function defaultAction()
    {
        if (false === $this->isGranted('ROLE_ADMIN')) {
            throw new AccessDeniedException();
        }

        // ...
    }
}
```

Además de los *roles* que definas en tu aplicación, Symfony dispone de varios *roles* internos que asigna a los usuarios en función de su tipo y que también puedes utilizar en tu código:

```
class DefaultController extends Controller
{
    public function defaultAction()
    {
        if ($this->isGranted('IS_AUTHENTICATED_FULLY')) {
            // El usuario está autenticado y la razón es que acaba de
            // introducir su nombre de usuario y contraseña
        } elseif ($this->isGranted('IS_AUTHENTICATED_REMEMBERED')) {
            // El usuario está autenticado, pero no ha introducido su
            // contraseña. La autenticación se ha producido por la
            // cookie de la opción "remember me"
        } elseif ($this->isGranted('IS_AUTHENTICATED_ANONYMOUSLY')) {
            // Se trata de un usuario anónimo. Técnicamente, en Symfony los
            // usuarios anónimos también están autenticados.
        }

        // ...
    }
}
```

Las plantillas de Twig también disponen de una función llamada `is_granted()` cuyo comportamiento es idéntico al del método `isGranted()` de los controladores:

```
{% if is_granted('ROLE_USUARIO') %}
    {# ... #}
{% elseif is_granted('ROLE_ADMIN') %}
    {# ... #}
{% endif %}
```

8.5.3 Modificando las plantillas de los usuarios *logueados*

La única modificación requerida por la aplicación *Cupon* es la de la caja de *login* que se muestra en la zona lateral de todas las páginas del *frontend*. Este *trozo de plantilla* se define en la plantilla `_caja_login.html.twig`. Actualiza su contenido por lo siguiente:

```
{# el usuario está logueado #}
{% if is_granted('ROLE_USUARIO') %}
    <p>Conectado como {{ usuario.nombre ~ ' ' ~ usuario.apellidos }}</p>

    <a href="#">Ver mi perfil</a>
    <a href="{{ path('usuario_logout') }}">Cerrar sesión</a>
{# el usuario todavía no ha hecho login #}
{% else %}
    <h2>Accede a tu cuenta</h2>

    <form action="{{ path('usuario_login_check') }}" method="post">
        {# ... #}
    </form>
{% endif %}
```

El código anterior tiene en cuenta que desde el controlador se le pasa una variable llamada `usuario` que representa al usuario actualmente *logueado*. Si no está disponible esa variable, puedes reemplazar el código anterior por lo siguiente:

```
{# antes #}
<p>Conectado como {{ usuario.nombre ~ ' ' ~ usuario.apellidos }}</p>

{# ahora #}
<p>Conectado como {{ app.user.nombre ~ ' ' ~ app.user.apellidos }}</p>
```

Si además sigues la buena práctica de añadir métodos `__toString()` en las entidades de Doctrine, puedes simplificar más el código:

```
{# antes #}
<p>Conectado como {{ usuario.nombre ~ ' ' ~ usuario.apellidos }}</p>

{# ahora #}
<p>Conectado como {{ app.user }}</p>
```

Para que este último código funcione de forma equivalente al anterior, en la entidad `Usuario` debe existir el siguiente método `__toString()`:

```
// src/AppBundle/Entity/Usuario.php
// ...
class Usuario implements UserInterface
{
    // ...

    public function __toString()
    {
        return $this->getNombre().' '.$this->getApellidos();
    }
}
```

Por otra parte, para cerrar la sesión del usuario la plantilla utiliza la ruta `usuario_logout`, definida anteriormente en el controlador `UsuarioController`. Al igual que sucede con `usuario_login_check`, no es necesario añadir código en `logoutAction()`, ya que Symfony se encarga de gestionarlo. No obstante, la ruta que espera Symfony es `/logout` y la ruta de la aplicación es `/usuario/logout`. Como es habitual, para solucionarlo simplemente hay que definir una opción de configuración en el archivo `security.yml`:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            form_login:
                login_path: usuario_login
                check_path: usuario_login_check
            logout:
                path: usuario_logout
```

La opción `path` de la clave `logout` indica la URL o la ruta de la acción de desconectarse de la aplicación. Además, si quieres redirigir a los usuarios a una página determinada después del *logout*, puedes hacerlo definiendo su URL en la opción `target`:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            logout:
                path:   usuario_logout
                target: /sitio/vuelve-pronto
```

8.5.4 Impersonando usuarios

Al desarrollar la aplicación puede resultar útil ver las plantillas con diferentes tipos de usuarios. Para no tener que hacer el *logout* y el *login* cada vez que quieras cambiar de usuario, Symfony incluye una opción llamada `switch_user` que permite impersonar usuarios:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            switch_user: true
```

Ahora, cuando quieras ver una página como si fueras otro usuario, sólo tienes que añadir como *query string* un parámetro llamado `_switch_user`:

```
http://127.0.0.1:8000/app_dev.php/usuario/compras/?_switch_user=usuario3@localho
st
http://127.0.0.1:8000/app_dev.php/?_switch_user=usuario100@localhost
```

Para volver a utilizar el usuario original, indica el valor especial `_exit` (`http://127.0.0.1:8000/app_dev.php/usuario/compras/?_switch_user=_exit`).

Por motivos de seguridad Symfony no deja impersonarse a cualquier usuario. Así, por defecto sólo pueden utilizar esta característica los usuarios que dispongan del *rol* `ROLE_ALLOWED_TO_SWITCH`. Para restringir todavía más su uso, puedes indicar el *rol* necesario para utilizar la impersonación:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            # ...
            switch_user: { role: ROLE_ADMIN }
```

8.6 Creando los archivos de datos de usuarios

La configuración de seguridad mostrada al principio de este capítulo codifica las contraseñas de los usuarios utilizando el algoritmo `bcrypt`. Después, su valor se cambió a `plaintext` para poder probar la aplicación con los usuarios de prueba creados mediante los archivos de datos o *fixtures*.

A continuación se muestra cómo codificar la contraseña del usuario dentro de un controlador utilizando la misma configuración que la definida en el archivo `security.yml`:

```
use AppBundle\Entity\Usuario;

class DefaultController extends Controller
{
    public function defaultAction()
    {
        $usuario = new Usuario();
        $encoder = $this->get('security.encoder_factory')
                        ->getEncoder($usuario);

        $password = $encoder->encodePassword(
            'la-contraseña-en-claro',
            $usuario->getSalt()
        );

        $usuario->setPassword($password);
    }
}
```

La clave del código anterior consiste en obtener a través del servicio `security.encoder_factory` el objeto `$encoder` que codifica las contraseñas. Como argumento se le pasa el objeto del usuario para el que se quiere codificar la contraseña.

Determinando el tipo de objeto que se le pasa y utilizando la configuración del archivo `security.yml`, el *encoder* es capaz de obtener la codificación utilizada para las contraseñas de esa

entidad. Una vez obtenido el *encoder*, ya puedes codificar cualquier contraseña mediante el método `encodePassword()`.

Codificar las contraseñas dentro de un archivo de datos es igual de sencillo, pero antes debes inyectar el contenedor de servicios en la clase (tal como se explica en el apéndice B (página 505)):

```
// src/AppBundle/DataFixtures/ORM/Usuarios.php
namespace AppBundle\DataFixtures\ORM;

use AppBundle\Entity\Usuario;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class Usuarios implements FixtureInterface, ContainerAwareInterface
{
    private $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }

    public function load(ObjectManager $manager)
    {
        for ($i=1; $i<=500; $i++) {
            $usuario = new Usuario();
            $encoder = $this->container->get('security.encoder_factory')
                            ->getEncoder($usuario);

            $passwordEnClaro = 'usuario'.$i;
            $password = $encoder->encodePassword($passwordEnClaro, null);
            $usuario->setPassword($password);

            // ...
        }
    }
}
```

Si en las pruebas anteriores habías modificado la configuración del archivo `security.yml`, vuelve a cambiarla para utilizar `bcrypt` en las contraseñas de los usuarios:

```
# app/config/security.yml
security:
    # ...
    encoders:
        AppBundle\Entity\Usuario: 'bcrypt'
```

Ahora vuelve a cargar los archivos de datos con el siguiente comando:

```
$ php app/console doctrine:fixtures:load
```

La aplicación debería seguir funcionando igual que antes y si introduces los datos de un usuario en el formulario de *login* podrás entrar en las partes restringidas de la aplicación. No obstante, si observas las tablas de la base de datos, ahora no verás las contraseñas en claro sino codificadas.

8.7 Formulario de registro

Crear usuarios de prueba es algo útil mientras se desarrolla la aplicación, pero cuando esté en producción los propios usuarios deben ser capaces de registrarse mediante un formulario.

Antes de empezar a crear el formulario, prepara la nueva ruta, acción y plantilla. Añade en primer lugar un nuevo método dentro de `UsuarioController`:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class UsuarioController extends Controller
{
    // ...

    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        return $this->render('usuario/registro.html.twig');
    }
}
```

Y ahora crea una plantilla llamada `registro.html.twig`:

```
{# app/Resources/views/usuario/registro.html.twig #}
{% extends 'frontend.html.twig' %}

{% block id 'usuario' %}
{% block title %}Regístrate gratis como usuario{% endblock %}

{% block article %}
    <h1>{{ block('title') }}</h1>
{% endblock %}

{% block aside %}{% endblock %}
```

8.7.1 Creando el formulario en el controlador

Utilizando la acción `registroAction()` que se acaba de añadir, ya es posible crear el formulario de registro y pasarlo a la plantilla asociada:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class UsuarioController extends Controller
{
    // ...

    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();

        $formulario = $this->createFormBuilder($usuario)
            ->add('nombre')
            ->add('apellidos')
            ->add('direccion', 'text')
            ->add('fechaNacimiento', 'date')
        ->getForm();

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

Los formularios se crean mediante un *form builder* obtenido a través del método `createFormBuilder()`, cuyo primer parámetro es un objeto con los datos iniciales que mostrará el formulario. Opcionalmente, puedes pasar un segundo parámetro al método con las opciones utilizadas para crear el formulario, que se explicarán más adelante.

Después de obtener el *form builder*, se añaden los campos del formulario mediante el método `add()`, que dispone de los siguientes tres argumentos:

- El primer argumento es el **nombre** del campo, que debe ser único dentro del formulario. En realidad, debe ser único dentro del grupo de campos de formulario al que pertenece.
- El segundo argumento es opcional e indica el **tipo** de campo utilizado para representarlo. Si no se indica explícitamente, el campo es de tipo `text` y se muestra como `<input type="text" />`.

- El tercer argumento también es opcional y consiste en un array con las **opciones** utilizadas para crear el campo, tal y como se explicará más adelante.

Los métodos `add()` simplemente añaden los campos al formulario. Para crear el verdadero objeto que representa al formulario y sus datos iniciales, se invoca el método `getForm()`. Este objeto no se puede pasar directamente a la plantilla. Antes hay que invocar el método `createView()`, que prepara la representación visual de cada campo con la información disponible y las opciones indicadas. Olvidar la instrucción `$formulario->createView()` al pasar el formulario a la vista es uno de los principales errores que se cometen al empezar con los formularios.

8.7.2 Creando el formulario en su propia clase

Crear el formulario dentro del controlador es la forma más rápida y sencilla de crearlo. El inconveniente es que ese formulario no se puede reutilizar en ninguna otra parte de la aplicación.

La buena práctica recomendada por Symfony es definir una clase por cada formulario. Estas clases se definen por convención en el directorio `Form/` del *bundle* y su nombre acaba en `Type`. De esta forma, para definir este formulario, crea el archivo `src/AppBundle/Form/UsuarioType.php` con el siguiente contenido:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('apellidos')
            ->add('email')
            ->add('password')
            ->add('direccion')
            ->add('permiteEmail')
            ->add('fechaNacimiento')
            ->add('dni')
            ->add('numero_tarjeta')
            ->add('ciudad')
            ->add('registrarme', 'submit')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
```

```
            'data_class' => 'AppBundle\Entity\Usuario',
        ));
    }

    public function getBlockPrefix()
    {
        return 'usuario';
    }
}
```

La estructura de las clases de formularios siempre es la que muestra el código anterior. En primer lugar, la clase hereda de `AbstractType`, que actúa de formulario base. Después se configuran los campos del formulario en el método `buildForm()`. A continuación, en el método `configureOptions()` se indica el *namespace* de la entidad cuyos datos modifica este formulario. Por último, se define un nombre único para el formulario con el método `getBlockPrefix()` (este valor se usa para generar los `id` y `name` de las etiquetas HTML del formulario).

Aunque después se realizarán varios ajustes en cada campo, al crear el formulario lo más sencillo es simplemente añadir con `add()` las propiedades de la entidad que se podrán manipular con el formulario y dejar que sea Symfony el que decida cuál es el tipo de campo más adecuado para cada propiedad.

NOTA A partir de la versión 2.3 de Symfony, además de las propiedades de la entidad que se va a manipular mediante el formulario, también es posible añadir botones. En el código anterior se añade un evento de tipo `submit` con el título `Registrarme`, pero también se pueden añadir botones de tipo `button` y `reset`.

Cuando un formulario se define mediante su propia clase, el código del controlador se simplifica mucho. Abre la acción `registroAction()` y reemplaza sus contenidos por lo siguiente:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class UsuarioController extends Controller
{
    // ...

    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
```

```
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

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

Gracias al atajo `createForm()` disponible en todos los controladores que heredan de la clase `Controller`, crear un formulario sólo requiere una línea de código. Como primer argumento se pasa el FQCN de la clase del formulario (que es igual a su *namespace* completo incluyendo el nombre de la clase).

Opcionalmente, como segundo argumento de `createForm()` se pasa un objeto con los datos que muestra inicialmente el formulario. Esto es muy útil por ejemplo para establecer el valor por defecto de algunos campos, como por ejemplo marcar la casilla que permite el envío de emails publicitarios:

```
// src/AppBundle/Controller/UsuarioController.php
class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $usuario->setPermiteEmail(true);

        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

        // ...
    }
}
```

Independientemente de si creas el formulario en el controlador o mediante una clase, la forma más sencilla de mostrarlo en la plantilla siempre es la misma:

```
{# app/Resources/views/usuario/registro.html.twig #}
{% extends 'frontend.html.twig' %}

{% block id 'usuario' %}
{% block title %}Regístrate gratis como usuario{% endblock %}

{% block article %}
    <h1>{{ block('title') }}</h1>
    {{ form(formulario) }}
{% endblock %}
```

```
{% block aside %}{% endblock %}
```

La función `form()` de Twig, a la que se le pasa la variable que guarda el formulario, muestra todos los campos del formulario, junto con los posibles mensajes de error y el título de cada campo. Una simple función de Twig basta para mostrar cualquier formulario, sin importar lo complejo que sea.

Por defecto la función `form()` supone que el formulario se envía a la misma URL que lo mostró y que se utiliza el método `POST`. Si necesitas modificar cualquiera de estas dos opciones, pasa los nuevos valores como parámetros de la función `form()`:

```
{# cambiar sólo la ruta del atributo 'action' #}
{{ form(formulario, { action: path('nombre_otra_ruta') }) }}

{# cambiar la ruta del atributo 'action' y el método de envío #}
{{ form(formulario, { action: path('nombre_otra_ruta'), method: 'GET' }) }}
```

La funcionalidad básica que permite mostrar el formulario de registro ya está completa. Para probarlo, añade el botón *Regístrate* encima de la caja de *login* que se muestra en la parte lateral de todas las páginas. Abre la plantilla `_caja_login.html.twig` y añade lo siguiente:

```
{# app/Resources/views/usuario/_caja_login.html.twig #}
{% if is_granted('ROLE_USUARIO') %}
    {# ... #}
{% else %}
    <a class="boton" href="{{ path('usuario_registro') }}">Regístrate</a>

    <h2>Accede a tu cuenta</h2>
    {# ... #}
{% endif %}
```

Si pulsas ahora el botón *Regístrate*, accederás a la URL `/usuario/registro` y verás el formulario de registro completo. Antes de añadir en el controlador la lógica necesaria para procesar y guardar los datos enviados por el usuario, se van a realizar varios ajustes en los campos del formulario y en la forma en que se muestran.

8.7.2.1 Ajustando los campos del formulario

Si no se indica el tipo de cada campo, Symfony lo infiere a partir de la información de la entidad asociada al formulario. El problema es que de esta forma muchos campos acaban mostrándose como un simple campo `<input type="text" />`. Como los navegadores actuales soportan todos los tipos de campos definidos por HTML5, es mejor explicitar siempre el tipo de cada campo:

```
// src/AppBundle/Form/UsuarioType.php
// ...

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
```

```
        $builder
            // ...
            ->add('email', 'email')
            // ...
            ->add('fechaNacimiento', 'birthday')
            // ...
            ->add('registrarme', 'submit')
        ;
    }

    // ...
}
```

Ahora el campo `email` se muestra como un campo de correo electrónico: visualmente parece un cuadro de texto, pero el valor que introduce el usuario tiene que ser un email válido. Además, la fecha de nacimiento se muestra con un tipo de campo `birthday`, que restringe el rango de años que se pueden seleccionar con respecto a un campo de fecha normal (sólo se pueden seleccionar los últimos 120 años).

Resulta habitual que los formularios de registro obliguen al usuario a introducir dos veces la misma contraseña, para asegurar que la ha escrito bien. Symfony dispone de un campo especial llamado `repeated` que muestra un campo dos veces y obliga a escribir el mismo valor en los dos:

```
// src/AppBundle/Form/UsuarioType.php
// ...

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('password', 'repeated', array(
                'type' => 'password',
                'invalid_message' => 'Las dos contraseñas deben coincidir',
                'first_options'   => array('label' => 'Contraseña'),
                'second_options'  => array('label' => 'Repite Contraseña'),
            ))
            // ...
        ;
    }

    // ...
}
```

Aunque el campo `repeated` se puede añadir simplemente con la instrucción `->add('password', 'repeated')`, se utiliza el tercer argumento del método `add()` para definir las opciones con las que se crea el campo:

- `type`, indica de qué tipo son los dos campos que se deben crear (en este caso, de tipo `password`).
- `invalid_message`, establece el mensaje de error que se muestra cuando los dos valores introducidos por el usuario no coinciden.
- `first_options` y `second_options`, estos arrays permiten definir las opciones individuales de cada uno de los dos campos que forman el campo `repeated`, como por ejemplo su título.

Si actualizas ahora el formulario de registro, verás dos campos para la contraseña. Si además introduces un valor cualquiera en el campo de email y tratas de enviar el formulario, el navegador muestra un mensaje de error y no envía el formulario.

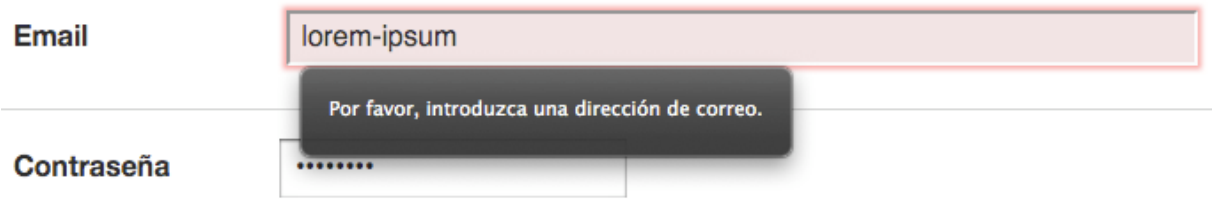

Figura 8.2 Mensaje de error mostrado por Google Chrome cuando no escribes un email válido

Por otra parte, Symfony marca por defecto todos los campos del formulario como obligatorios. Así que si el usuario no rellena todos los campos, el navegador no le permite enviar el formulario. Este comportamiento casi siempre es el deseado, salvo en los campos de tipo `checkbox`.

Si un campo `checkbox` se marca como `required`, es obligatorio que el usuario marque la casilla de verificación para poder enviar el formulario. Esto no es el comportamiento deseado para aquellas opciones que no sea obligatorio aceptar, como por ejemplo la opción de apuntarse a un boletín de noticias.

De forma que para permitir que un usuario pueda desactivar un *checkbox*, es necesario indicar que no es `required`. Esto se define mediante las opciones de configuración del tercer parámetro del método `->add()`:

```
// src/AppBundle/Form/UsuarioType.php
// ...

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('permiteEmail', 'checkbox', array('required' => false))
            // ...
        ;
    }

    // ...
}
```

Como las opciones están en el tercer parámetro del método `->add()`, también hay que indicar el segundo parámetro del método. Puedes escribir el tipo de campo correcto (`checkbox` en este caso) o indicar el valor `null` para que sea Symfony el que decida cuál es el tipo de campo más adecuado.

8.7.2.2 Ajustando el aspecto del formulario

Mostrar el formulario en la plantilla con la función `{{ form(formulario) }}` sólo es aconsejable cuando estás prototipando la aplicación. En una aplicación web real seguramente querrás controlar cómo se muestra cada campo del formulario, modificando sus títulos, añadiendo mensajes de ayuda, modificando el lugar donde se muestran los mensajes de error, etc.

Para ello puedes hacer uso de las siguientes funciones de Twig:

- `form_start(formulario)`, muestra la etiqueta `<form>` de apertura del formulario e incluye el atributo `enctype` si es necesario.
- `form_errors(formulario)`, muestra todos los errores globales del formulario, es decir, aquellos errores que no se muestran al lado de cada campo.
- `form_row(formulario.campo)`, muestra el título, errores y etiqueta HTML del campo de formulario indicado.
- `form_end(formulario)`, muestra la etiqueta `<form>` de cierre del formulario y añade antes cualquier campo del formulario que no se haya mostrado mediante las funciones anteriores. Esto es muy útil por ejemplo para mostrar los campos ocultos de tipo `hidden` sin tener que añadirlos a mano.

NOTA En las versiones anteriores a Symfony 2.3 también se utilizaban las funciones `form_enctype()` y `form_rest()`. Su uso está desaconsejado en las versiones recientes de Symfony porque es mejor utilizar las nuevas funciones equivalentes `form_start()` y `form_end()`.

Utilizando las funciones anteriores puedes reordenar y agrupar los campos del formulario como quieras:

```
{# app/Resources/views/usuario/registro.html.twig #}

{# ... #}

{{ form_start(formulario) }}
    {{ form_errors(formulario) }}

    <fieldset>
       <legend>Datos personales</legend>

       {{ form_row(formulario.nombre) }}
       {{ form_row(formulario.apellidos) }}
       {{ form_row(formulario.fechaNacimiento) }}
       {{ form_row(formulario.dni) }}
       {{ form_row(formulario.direccion) }}
```

```
    </fieldset>

    <fieldset>
       <legend>Datos de acceso</legend>

       {{ form_row(formulario.email) }}
       {{ form_row(formulario.password) }}
    </fieldset>

    <fieldset>
       <legend>Datos económicos</legend>

       {{ form_row(formulario.numero_tarjeta) }}
       {{ form_row(formulario.ciudad) }}
    </fieldset>

    {{ form_row(formulario.permiteEmail) }}
{{ form_end(formulario) }}

{# ... #}
```

La función `form_start()`, al igual que se explicó anteriormente para la función `form()`, supone que el formulario se envía a la misma URL que lo mostró y que se utiliza el método `POST`. Para modificar estas opciones, pasa los nuevos valores como parámetros de la función `form_start()`:

```
{# cambiar sólo la ruta del atributo 'action' #}
{{ form_start(formulario, { action: path('nombre_otra_ruta') }) }}

{# cambiar la ruta del atributo 'action' y el método de envío #}
{{ form_start(formulario, { action: path('nombre_otra_ruta'), method: 'GET' })
}}
```

El valor de estas opciones también se pueden establecer en el propio controlador, utilizando el tercer argumento opcional del método `createForm()`:

```
// src/AppBundle/Controller/UsuarioController.php
class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario,
array(
            'action' => $this->generateUrl('nombre_otra_ruta'),
            'method' => 'GET',
        ));
```

```
        // ...
    }
}
```

Por otra parte, la función `form_row()` muestra cada campo de formulario encerrado en una etiqueta `<div>`. Si necesitas un control más preciso sobre las diferentes partes de cada campo, puedes utilizar las siguientes funciones:

- `form_label(formulario.campo)`, muestra el título del campo (etiqueta `<label>`).
- `form_errors(formulario.campo)`, muestra los errores específicos de este campo.
- `form_widget(formulario.campo)`, muestra la etiqueta o etiquetas HTML necesarias para representar este campo. Puede ser un simple `<input type="text">`, varias `<select>` juntas para seleccionar una fecha, etc.

Estas funciones también se pueden combinar con las anteriores en un mismo formulario:

```
{# app/Resources/views/usuario/registro.html.twig #}

{# ... #}

{{ form_start(formulario) }}
  {{ form_errors(formulario) }}

  <fieldset>
     <legend>Datos personales</legend>

     <div>
         <strong>{{ form_label(formulario.nombre) }}</strong>
         <div class="error">{{ form_errors(formulario.nombre) }}</div>
         <span>{{ form_widget(formulario.nombre) }}</span>
     </div>

     {{ form_row(formulario.apellidos) }}
     {{ form_row(formulario.fechaNacimiento) }}
     {{ form_row(formulario.dni) }}
     {{ form_row(formulario.direccion) }}
  </fieldset>

  {# ... #}
```

Symfony infiere el título de cada campo a partir del nombre de su propiedad asociada. Si necesitas ajustar el título autogenerado, por ejemplo para añadir acentos, puedes indicar tu propio título como segundo argumento de la función `form_label()`:

```
{{ form_label(formulario.direccion, 'Dirección postal') }}
```

La función `form_widget()` también permite pasar como segundo parámetro un array asociativo de opciones. Cada campo de formulario define de sus propias opciones, pero todos ellos disponen de la opción `attr`, con la que puedes añadir atributos a la etiqueta HTML generada:

```
{{ form_widget(formulario.nombre, { attr: {
    accesskey : 'n',
    tabindex  : 1,
    class     : 'destacado',
    size      : 25
} }) }}
```

8.7.3 Procesando los datos enviados por el usuario

La recomendación de Symfony consiste en utilizar un único controlador para mostrar el formulario inicial vacío y para procesar los datos que envía el usuario después de rellenarlo. El controlador utiliza el tipo de petición para determinar qué debe hacer:

- Si la petición es de tipo `GET`, se crea un formulario vacío y se pasa a la plantilla.
- Si la petición es de tipo `POST`, se obtienen los datos enviados por el usuario, se validan y se guardan en la base de datos. Después se redirige al usuario a otra página.

El siguiente código muestra por tanto la estructura común recomendada para todas las acciones que muestran y procesan formularios:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            // Validar los datos enviados y guardarlos en la base de datos
        }

        return $this->render('usuario/registro.html.twig', array(
            'formulario' => $formulario->createView()
        ));
```

```
    }
}
```

Symfony simplifica el procesado de los formularios a solamente dos líneas de código:

```
$formulario->handleRequest($request);
if ($formulario->isValid()) {
    // ...
}
```

El método `handleRequest()` asocia el objeto `$formulario` creado anteriormente con los datos enviados por el usuario en la `$request`. Después de ejecutar esta instrucción, el objeto asociado con el formulario (en el ejemplo anterior, el objeto `$usuario`) está relleno con la nueva información enviada por el usuario, sin importar si esta es válida o no.

El método `isValid()` comprueba si los datos del formulario (es decir, los datos enviados por el usuario) cumplen con las reglas de validación. Si cumplen, el método devuelve `true` y como el formulario es válido, ya se puede guardar la información en la base de datos. Si al menos un campo no es válido, el método devuelve `false` y Symfony se encarga de volver a mostrar el formulario, con la información enviada por el usuario y con los mensajes de error apropiados en cada campo.

Con todo lo anterior, el siguiente código muestra cómo procesar el formulario de registro:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            $encoder = $this->get('security.encoder_factory')->getEncoder($usuar
io);
            $passwordCodificado = $encoder->encodePassword($usuario->getPasswor
d(), null);
            $usuario->setPassword($passwordCodificado);

            $em = $this->getDoctrine()->getManager();
```

```
            $em->persist($usuario);
            $em->flush();

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

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

Como en el formulario de registro el usuario introduce su contraseña en claro, antes de guardar la información en la base de datos es necesario codificar la contraseña siguiendo el procedimiento explicado en las secciones anteriores.

Recuerda que después de ejecutar el método `handleRequest()`, el objeto asociado con el formulario ya contiene toda la información enviada por el usuario. Así que para guardarla en la base de datos, simplemente se pasa el objeto al método `persist()` del *entity manager* de Doctrine.

Después de guardar la información, además de redirigir al usuario a una nueva página (normalmente la portada) es habitual mostrarle un mensaje indicando que se ha registrado correctamente. Este tipo de mensajes se denominan **mensajes flash** y una vez creados, solamente están disponibles en la siguiente página que se visita:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        // ...

        if ($formulario->isValid()) {
            // ...

            $this->addFlash('info', ' Enhorabuena! Te has registrado correctamen
te en Cupon');

            return $this->redirectToRoute('portada', array(
                'ciudad' => $usuario->getCiudad()->getSlug()
```

```
            ));
        }

        // ...
    }
}
```

Los mensajes *flash* se crean a través de la sesión del usuario. Dentro de un controlador puedes utilizar el método `addFlash()`, cuyo primer argumento identifica al tipo de mensaje (que puedes elegir libremente) y cuyo segundo argumento es el propio contenido del mensaje.

En las plantillas, los mensajes *flash* están disponibles a través de una variable especial llamada `flashbag` y que pertenece a la sesión. Para mostrar solamente los mensajes de un determinado tipo, utiliza el método `get()` indicando el tipo de mensaje como argumento. Este método devuelve un array de mensajes, por lo que tendrás que utilizar un código como el siguiente:

```
{# app/Resources/views/usuario/_caja_login.html.twig #}
{% if is_granted('ROLE_USUARIO') %}

    {% for mensaje in app.session.flashbag.get('info') %}
        <p class="info">{{ mensaje }}</p>
    {% endfor %}

    <p>Conectado como {{ usuario.nombre ~ ' ' ~ usuario.apellidos }}</p>

    <a href="#">Ver mi perfil</a>
    <a href="{{ path('usuario_logout') }}">Cerrar sesión</a>
{% else %}
    {# ... #}
{% endif %}
```

Para mostrar en una plantilla todos los mensajes *flash* que puedan existir, sin importar el tipo, utiliza el método `all()`:

```
{% for tipo, mensajes in app.session.flashbag.all() %}
    {% for mensaje in mensajes %}
        <p class="flash-{{ tipo }}">{{ mensaje }}</p>
    {% endfor %}
{% endfor %}
```

Una última mejora en el formulario de registro sería *loguear* al nuevo usuario antes de redirigirle a la portada y mostrarle el mensaje *flash*. Para ello hay que crear el *token* que representa al usuario conectado. La forma de crearlo depende del tipo de autenticación utilizada. Para el caso de la autenticación con usuario y contraseña, el código necesario es el siguiente:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;

class UsuarioController extends Controller
{
    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        // ...

        if ($formulario->isValid()) {
            // ...

            $token = new UsernamePasswordToken(
                $usuario,
                $usuario->getPassword(),
                'frontend',
                $usuario->getRoles()
            );
            $this->container->get('security.token_storage')->setToken($token);

            return $this->redirectToRoute('portada', array(
                'ciudad' => $usuario->getCiudad()->getSlug()
            ));
        }

        // ...
    }
}
```

Los usuarios del *frontend* se autentican con usuario y contraseña, por lo que la clase adecuada para crear el *token* es `UsernamePasswordToken` (no olvides importarla primero con la instrucción `use`). Su primer argumento es el nombre de usuario (o el objeto completo que representa al usuario) y el segundo es la contraseña. El tercer argumento es el nombre del firewall asociado al proveedor de este usuario y el cuarto parámetro es un array con los *roles* del usuario.

8.7.4 Validación de los campos del formulario

Como se explicó anteriormente, la instrucción `$formulario->isValid()` se encarga de validar que la información del formulario sea correcta. En realidad, lo que se valida es la información del objeto asociado con el formulario. Por tanto, la validación se configura en las entidades, no en los formularios.

Las reglas de validación se pueden configurar mediante YAML, PHP, XML o anotaciones. Como se sabe, las anotaciones es el formato más cómodo, ya que no es necesario crear un nuevo archivo de configuración. Los siguientes ejemplos utilizan las anotaciones, así que asegúrate de que estén

activadas comprobando el valor de la opción `enable_annotations` del archivo general de configuración:

```
# app/config/config.yml
framework:
    # ...
    validation: { enable_annotations: true }
```

A continuación, modifica la clase de la entidad `Usuario` para añadir la primera regla de validación sobre la propiedad `nombre`:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @ORM\Column(type="string", length=100)
     * @Assert\NotBlank()
     */
    private $nombre;

    // ...
}
```

Symfony denomina *constraints* a las reglas de validación. Para facilitar su configuración, añade esta instrucción al principio de la entidad:

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

Ahora ya puedes configurar cualquier *constraint* fácilmente mediante el prefijo `@Assert\`. Symfony incluye predefinidas numerosas *constraints*, como por ejemplo `NotBlank()`, que comprueba que el valor introducido no sea `null` o una cadena de texto vacía. Así que la anotación `@Assert\NotBlank()` sobre la propiedad `nombre` impide que el usuario deje vacío su nombre.

Guarda los cambios en la entidad y recarga la página que muestra el formulario de registro. No verás ninguna diferencia en el campo `nombre`. La razón es que Symfony marca por defecto todos los campos del formulario como requeridos, lo que ya obliga a introducir un valor no vacío. No obstante, aunque en el formulario del *frontend* no se vea ninguna diferencia, ahora la entidad sí que valida correctamente que el nombre no esté vacío y antes no lo hacía.

Añade a continuación la siguiente regla de validación sobre la propiedad `password` para que su longitud sea de al menos seis caracteres:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min = 6)
     */
    private $password;

    // ...
}
```

Prueba ahora a recargar la página del formulario de registro y escribe una contraseña de menos de seis caracteres. Si envías el formulario, se mostrará de nuevo el mismo formulario, con toda la información que acabas de introducir y con un mensaje de error al lado del campo de contraseña indicando que debe tener al menos seis caracteres.

Si el mensaje de error se muestra en inglés (*"This value is too short. It should have 6 characters or more"*), cambia el idioma de la aplicación mediante la opción `locale` del archivo `app/config/parameters.yml`:

```
parameters:
    # ...
    locale: es
```

Después de cambiar esta opción, borra la caché de la aplicación (`php app/console cache:clear --no-warmup --env=dev`) y borra todas las *cookies* del navegador. Si ahora vuelves a probar a registrarte con una contraseña de menos de seis caracteres, verás el mensaje de error *"Este valor es demasiado corto. Debería tener 6 caracteres o más"*.

8.7.4.1 Reglas de validación incluidas en Symfony

Symfony incluye decenas de reglas de validación o *constraints* predefinidas listas para usar en cualquier entidad. A continuación se muestran todas ellas junto con sus opciones más importantes:

Validaciones **básicas**:

- `@Assert\NotBlank()`, el valor no es `null` o una cadena de texto vacía.
- `@Assert\Blank()`, el valor es `null` o una cadena de texto vacía.
- `@Assert\NotNull()`, el valor no es estrictamente `null` (comparación realizada con `===`).
- `@Assert\Null()`, el valor es estrictamente `null` (comparación realizada con `===`).
- `@Assert\True()`, el valor es estrictamente `true`, el número `1` o la cadena de texto `"1"` (comparación realizada con `===`).
- `@Assert\False()`, el valor es estrictamente `false`, el número `0` o la cadena de texto `"0"` (comparación realizada con `===`).
- `@Assert\Type()`, el valor es del tipo indicado en la opción `type`. Ej: `@Assert\Type(type="array")`

Validaciones para **cadenas de texto**:

- `@Assert\Email()`, el valor tiene el aspecto de un email válido. Si quieres asegurarte de que la dirección de correo electrónico exista, puedes utilizar la opción `checkMX`, que utiliza la función `checkdnsrr()` de PHP para comprobar que existe el servidor al que referencia el email. Ej: `@Assert\Email(checkMX=true)`
- `@Assert\Length`, el valor tiene la longitud mínima indicada en la opción `min` y/o la longitud máxima indicada en la opción `max`. Ej: `@Assert\Length(max=255)`, `@Assert\Length(min=4, max=12)`
- `@Assert\Url()`, el valor tiene el aspecto de una URL válida.
- `@Assert\Regex()`, el valor cumple con el patrón de la expresión regular definido en la opción `path`. Ej: `@Assert\Regex(path="/\d{9}/")` para validar un número de teléfono de nueve cifras. Si quieres que el valor **no** cumpla con el patrón indicado, añade la opción `match=false`. Ej: `@Assert\Regex(path="/\d/", match=false)` para asegurar que el valor no contiene ningún número.
- `@Assert\Ip()`, el valor tiene el aspecto de una dirección IP. Permite validar direcciones IPv4, IPv6, todas las direcciones salvo los rangos privados, etc.

Validaciones para **números**:

- `@Assert\Range()`, el valor está comprendido entre el mínimo indicado en la opción `min` y/o el máximo indicado en la opción `max`. Ej: `@Assert\Range(min=18)`, `@Assert\Range(min=18, max=65)`

Validaciones para **comparaciones**:

- `@Assert\EqualTo`, el valor es igual al indicado en la opción `value`. Ejemplo: `@Assert\EqualTo(value = 3)`. La comparación es de tipo `==`. Si quieres comparar el valor con `===`, utiliza la validación `IdenticalTo`.

- `@Assert\NotEqualTo`, el valor es distinto al indicado en la opción `value`. Ejemplo: `@Assert\NotEqualTo(value = 3)`. La comparación es de tipo `!=`. Si quieres comparar el valor con `!==`, utiliza la validación `NotIdenticalTo`.
- `@Assert\IdenticalTo`, el valor es idéntico al indicado en la opción `value`. Ejemplo: `@Assert\IdenticalTo(value = 3)`. La comparación es de tipo `===`. Si quieres comparar el valor con `==`, utiliza la validación `EqualTo`.
- `@Assert\NotIdenticalTo`, el valor no es idéntico al indicado en la opción `value`. Ejemplo: `@Assert\NotIdenticalTo(value = 3)`. La comparación es de tipo `!==`. Si quieres comparar el valor con `!=`, utiliza la validación `NotEqualTo`.
- `@Assert\LessThan`, el valor es menor que el indicado en la opción `value`. Ejemplo: `@Assert\LessThan(value = 3)`. La comparación es de tipo `<`. Si quieres comparar el valor con `<=`, utiliza la validación `LessThanOrEqual`.
- `@Assert\LessThanOrEqual`, el valor es menor o igual que el indicado en la opción `value`. Ejemplo: `@Assert\LessThanOrEqual(value = 3)`. La comparación es de tipo `<=`. Si quieres comparar el valor con `<`, utiliza la validación `LessThan`.
- `@Assert\GreaterThan`, el valor es mayor que el indicado en la opción `value`. Ejemplo: `@Assert\GreaterThan(value = 3)`. La comparación es de tipo `>`. Si quieres comparar el valor con `>=`, utiliza la validación `GreaterThanOrEqual`.
- `@Assert\GreaterThanOrEqual`, el valor es mayor o igual que el indicado en la opción `value`. Ejemplo: `@Assert\GreaterThanOrEqual(value = 3)`. La comparación es de tipo `>=`. Si quieres comparar el valor con `>`, utiliza la validación `GreaterThan`.

Validaciones para **fechas**:

- `@Assert\Date()`, el valor es un objeto de tipo `DateTime` o una cadena de texto (u objeto convertible a cadena de texto) con el formato `YYYY-MM-DD`
- `@Assert\DateTime()`, el valor es un objeto de tipo `DateTime` o una cadena de texto (u objeto convertible a cadena de texto) con el formato `YYYY-MM-DD HH:MM:SS`
- `@Assert\Time()`, el valor es un objeto de tipo `DateTime` o una cadena de texto (u objeto convertible a cadena de texto) con el formato `HH:MM:SS`

Validaciones para **colecciones**:

- `@Assert\Choice()`, el valor se encuentra dentro de los valores indicados mediante la opción `choices`. Ej: `@Assert\Choice(choices = {"hombre", "mujer"})`.
- `@Assert\Collection()`, cuando el valor de una propiedad es un array de valores, esta regla de validación permite definir la validación de cada valor dentro del array.
- `@Assert\Count()`, comprueba que la colección de valores (por ejemplo un array) contiene al menos el número de elementos indicados en la opción `min` y/o como máximo el número de elementos indicado en la opción `max`. Ej: `@Assert\Count(min = 2)`, `@Assert\Count(min = 10, max = 100)`
- `@Assert\Language()`, el valor es un código válido de idioma (`es`, `de`, `ja`)

- `@Assert\Locale()`, el valor es un código válido de *locale* o cultura, indicada mediante dos letras según el estándar ISO639-1 (`es`, `de`, `ja`) seguidas opcionalmente por un guión bajo y el código de un país según el estándar ISO3166 (`es_ES`, `es_AR`, `en_US`, `fr_FR`).
- `@Assert\Country()`, el valor es un código válido de país compuesto de dos letras.
- `@DoctrineAssert\UniqueEntity()`, asegura que un valor es único dentro de una misma entidad (es decir, que no se repite el mismo valor en otras filas de la misma tabla de la base de datos). Esta validación se indica directamente sobre la entidad, no sobre sus propiedades:

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * No puede haber dos emails iguales en los registros de la base de
 * datos de la entidad Usuario:
 *
 * @UniqueEntity("email")
 */
class Usuario { ... }

/**
 * No puede haber dos emails iguales o dos DNIs iguales en los
 * registros de la base de datos de la entidad Usuario:
 *
 * @UniqueEntity("email")
 * @UniqueEntity("dni")
 */
class Usuario { ... }

/**
 * No puede haber una combinación email + DNI repetida en los
 * registros de la base de datos de la entidad Usuario. Por ejemplo
 * dos registros pueden tener el mismo email, pero solo si su DNI
 * es diferente:
 *
 * @UniqueEntity(fields = { "email", "dni" })
 */
class Usuario { ... }
```

Validaciones para **archivos**:

- `@Assert\File()`, el valor es un objeto de tipo `File`, `UploadedFile` o una cadena de texto (u objeto convertible a cadena de texto) que corresponde a la ruta de un archivo existente. Se puede limitar su tamaño máximo con `maxSize` y el tipo de archivo con `mimeTypes`. Ej: `@Assert\File(maxSize = "5M", mimeTypes = {"application/pdf", "application/x-pdf"})` para validar que se suba un archivo PDF con un peso máximo de 5 MB.
- `@Assert\Image()`, es exactamente igual que `@Assert\File()` pero la opción `mimeTypes` está preconfigurada con los tipos de archivos adecuados para las imágenes.

Validaciones para **códigos numéricos** especiales:

- `@Assert\CardScheme`, el valor es un número de tarjeta de crédito/débito que es correcto para el tipo o tipos de tarjeta indicados en la opción `schemes`. Ej: `@Assert\CardScheme(schemes = {"VISA"})`. Los tipos de tarjeta soportados son: `AMEX`, `CHINA_UNIONPAY`, `DINERS`, `DISCOVER`, `INSTAPAYMENT`, `JCB`, `LASER`, `MAESTRO`, `MASTERCARD` y `VISA`.
- `@Assert\Currency`,el valor corresponde al código de alguna de las divisas definidas en el estándar ISO-4217 (http://en.wikipedia.org/wiki/ISO_4217) . Cada divisa se identifica mediante una cadena de tres caracteres (`EUR` para el euro, `USD` para los dólares americanos, etc.)
- `@Assert\Luhn`, el valor es una cadena de números que cumple con el algoritmo de Luhn (http://es.wikipedia.org/wiki/Algoritmo_de_Luhn) , que es el que utilizan todas las tarjetas de crédito (el validador `@Assert\CardScheme` además de esto valida que el número corresponda a la empresa indicada).
- `@Assert\Iban`, el valor corresponde a un número de cuenta bancaria que cumple el formato IBAN (http://es.wikipedia.org/wiki/International_Bank_Account_Number) .
- `@Assert\Isbn`, el valor corresponde a un número válido según el estándar ISBN (*International Standard Book Numbers*). Se puede comprobar si el número corresponde al ISBN-10, al ISBN-13 o a ambos. Ejemplo: `@Assert\Isbn(isbn10 = true, isbn13 = true)`
- `@Assert\Issn`, el valor corresponde a un número válido según el estándar ISSN (*International Standard Serial Number*).

Otras validaciones **especiales**:

- `@Assert\Callback()`, el valor se valida mediante el *callback* de PHP indicado en la opción `methods`. Ej: `@Assert\Callback(methods={"compruebaISBN"})`
- `@Assert\All()`, permite aplicar varias validaciones a todos los valores de un array. Ej: `@Assert\All({ @Assert\NotNull, @Assert\MaxLength(100) })`
- `@Assert\UserPassword`, valida que el valor pasado es igual que la contraseña del usuario actualmente conectado. Resulta útil para los formularios que permiten al usuario cambiar su contraseña y que primero deben comprobar que el usuario ha introducido correctamente su contraseña actual.
- `@Assert\Valid()`, valida las propiedades que en realidad son objetos con sus propias reglas de validación.

Cuando se produce un error de validación, Symfony muestra un mensaje predefinido para cada error. Los mensajes están disponibles en más de 30 idiomas y puedes consultar todos los mensajes en español en el archivo `validators.es.xliff`. Si quieres modificar el contenido de algún mensaje, utiliza la opción `message` disponible en casi todas las *constraints* explicadas anteriormente:

```
/**
 * @Assert\NotBlank()
 */
```

```
protected $nombre;

// Mensaje mostrado: "Este valor no debería estar vacío"


/**
 * @Assert\NotBlank(message = "Por favor, escribe tu nombre")
 */
protected $nombre;

// Mensaje mostrado: "Por favor, escribe tu nombre"
```

Los mensajes también pueden utilizar el valor de cualquier otra opción definida en la regla de validación:

```
/**
 * @Assert\Length(min = 5)
 */
protected $direccion;

// Mensaje mostrado:
// "Este valor es demasiado corto. Debería tener 5 caracteres o más"


/**
 * @Assert\Length(min = 5, minMessage = "La dirección debería tener {{ limit }}
caracteres o más para considerarse válida")
 */
protected $direccion;

// Mensaje mostrado:
// "La dirección debería tener 5 caracteres o más para considerarse válida"
```

Algunas *constraints* complejas no disponen de una simple opción `message` sino que incluyen varios mensajes para cubrir todos los posibles errores. Así por ejemplo la *constraint* `File` dispone de los mensajes `maxSizeMessage`, `mimeTypesMessage`, `notFoundMessage`, `notReadableMessage`, `uploadIniSizeErrorMessage`, `uploadFormSizeErrorMessage` y `uploadErrorMessage`.

Haciendo uso de las *constraints* del listado anterior, resulta sencillo añadir las reglas de validación a la entidad `Usuario`. La única validación especial es la de la propiedad `email`, ya que primero se comprueba que el usuario haya escrito un email correcto y después se comprueba que ese email no exista en alguna otra entidad de tipo `Usuario`:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
```

```
use Symfony\Bridge\Doctrine\Validator\Constraints as DoctrineAssert;

/**
 * @ORM\Entity
 * @DoctrineAssert\UniqueEntity("email")
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @ORM\Column(type="string", length=255, unique=true)
     * @Assert\Email()
     */
    private $email;

    // ...
}
```

Para asegurar que el valor de la propiedad `email` sea un email válido, se añade la anotación `@Assert\Email()` en la propiedad. Para asegurar que dos usuarios no utilicen el mismo email se añade la anotación `@DoctrineAssert\UniqueEntity("email")` en la entidad. Para utilizar esta última validación, no olvides importar la nueva clase y definir el prefijo `@DoctrineAssert`.

8.7.4.2 Reglas de validación propias

A pesar de que las *constraints* incluidas en Symfony cubren las validaciones más comunes, es muy probable que tengas que crear tu propio validador para comprobar el valor de alguna propiedad. En la entidad `Usuario` por ejemplo, la propiedad DNI requiere claramente un validador a medida.

La regla de validación `@Assert\Callback()` definida sobre la entidad indica qué métodos adicionales se ejecutan para considerar válidos los datos de la entidad. Siguiendo la nomenclatura sugerida por Symfony, el método que valida el DNI se llama `esDniValido()`:

```
// src/AppBundle/Entity/Usuario.php

use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints as DoctrineAssert;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Validator\Context\ExecutionContextInterface;

/**
 * @ORM\Entity()
 * @DoctrineAssert\UniqueEntity("email")
 * @Assert\Callback(callback = { "esDniValido" })
 */
class Usuario implements UserInterface
{
```

```
    // ...

    public function esDniValido(ExecutionContextInterface $context)
    {
        // ...
    }
}
```

Los métodos de validación no devuelven ningún valor, ni siquiera cuando falla la validación. Su funcionamiento se basa en añadir *violaciones* cuando se produzca algún error de validación. Estas *violaciones* se convierten después en los mensajes de error que se muestran al lado de cada campo del formulario. Para añadir las *violaciones* utiliza el objeto de tipo `ExecutionContextInterface` que Symfony pasa automáticamente como primer argumento del método propio de validación. No olvides importar la clase `ExecutionContextInterface` con la instrucción `use` tal y como muestra el código anterior.

En el caso del DNI se pueden producir dos tipos de *violaciones*: que no tenga el formato adecuado (*entre uno y ocho números seguidos de una letra*) o que la letra no corresponda al número. Así, el código completo del validador del DNI es el siguiente:

```
// src/AppBundle/Entity/Usuario.php

use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints as DoctrineAssert;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Validator\Context\ExecutionContextInterface;

/**
 * @ORM\Entity()
 * @DoctrineAssert\UniqueEntity("email")
 * @Assert\Callback(callback = { "esDniValido" })
 */
class Usuario implements UserInterface
{
    // ...

    public function esDniValido(ExecutionContextInterface $context)
    {
        $dni = $this->getDni();

        // Comprobar que el formato sea correcto
        if (0 === preg_match("/\d{1,8}[a-z]/i", $dni)) {
            $context->buildViolation('El DNI no tiene el formato correcto: entr
e 1 y 8 números seguidos de una letra (sin guiones y sin espacios)')
                ->atPath('dni')
                ->addViolation();
```

```
            return;
        }

        // Comprobar que la letra cumple con el algoritmo
        $numero = substr($dni, 0, -1);
        $letra = strtoupper(substr($dni, -1));
        if ($letra !== substr('TRWAGMYFPDXBNJZSQVHLCKE', strtr($numero, 'XYZ',
'012') % 23, 1)) {
            $context->buildViolation('La letra del DNI no es correcta para el nú
mero indicado.')
                ->atPath('dni')
                ->addViolation();
        }
    }
}
```

Las *violaciones* se crean con el método `buildViolation()` y se asocian con una propiedad determinada de la entidad mediante el método `atPath()`.

8.7.4.3 Métodos de validación *ad–hoc*

Symfony incluye otra forma más sencilla de crear validadores propios sin tener que definir una *constraint* propia con la anotación `Callback`. Su funcionamiento se basa en crear dentro de la entidad un nuevo método público cuyo nombre empiece por `is` o `get` y asociarle después una regla de validación de tipo `@Assert\True`.

Así por ejemplo podrías utilizar el siguiente código para asegurar que todos los usuarios que se registren sean mayores de edad:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity()
 * ...
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @Assert\True(message = "Debes tener al menos 18 años")
     */
    public function isMayorDeEdad()
    {
```

```
        return $this->fechaNacimiento <= new \DateTime('today - 18 years');
    }

    // ...
}
```

Los datos de una entidad sólo se consideran válidos si cumplen todas las reglas de validación, así que puedes incluir tantas validaciones como necesites utilizando el *truco* de los métodos `isXXX()` o `getXXX()`. El único inconveniente de este método es que los mensajes de error no se muestran al lado de cada campo sino de forma global en el formulario.

Para solucionar este problema, utiliza la opción `error_mapping` de los formularios de Symfony. A esta opción se le pasa como valor un array con pares `clave => valor` en los que la clave es el nombre del método de validación pero sin el prefijo `is` o `get` y el valor es el nombre del campo de formulario en el que se debe mostrar el mensaje de error:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Usuario',
            'error_mapping' => array(
                'mayorDeEdad' => 'fechaNacimiento'
            ),
        ));
    }

    // ...
}
```

8.8 Visualizando el perfil del usuario

Los usuarios registrados y *logueados* pueden visualizar toda la información de su perfil para actualizar cualquier información que deseen. Para añadir esta nueva funcionalidad en la aplicación, define una nueva acción llamada `perfilAction()` dentro del controlador `UsuarioController`:

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

/**
 * @Route("/usuario")
 */
class UsuarioController extends Controller
{
    /**
     * @Route("/perfil", name="usuario_perfil")
     */
    public function perfilAction(Request $request)
    {
        // ...
    }

    // ...
}
```

Después, actualiza el enlace de la opción *Ver mi perfil* en la plantilla `_caja_login.html.twig`:

```
{# app/Resources/views/usuario/_caja_login.html.twig #}
{% if is_granted('ROLE_USUARIO') %}
    {{ include('comun/_flashes.html.twig') }}

    <p>Conectado como {{ usuario.nombre ~ ' ' ~ usuario.apellidos }}</p>

    <a href="{{ path('usuario_perfil') }}">Ver mi perfil</a>
    <a href="{{ path('usuario_logout') }}">Cerrar sesión</a>
{% else %}
    {# ... #}
{% endif %}
```

A continuación se añade el código en la acción `perfilAction()` siguiendo esta lógica:

1. Obtener los datos del usuario logueado.
2. Crear un formulario de registro y rellenarlo con los datos del usuario.
3. Si la petición es GET, mostrar el formulario.
4. Si la petición es POST, actualizar la información del usuario con los nuevos datos obtenidos del formulario y redirigir a otra página.

El usuario *logueado* se obtiene directamente a través del atajo `getUser()`:

```
/** @Route("/perfil", name="usuario_perfil") */
public function perfilAction(Request $request)
```

```
{
    $usuario = $this->getUser();
}
```

Como el formulario que muestra los datos del perfil y el formulario de registro tienen los mismos campos, se reutiliza exactamente el mismo formulario. Esta es la gran ventaja de definir cada formulario en su propia clase. La única diferencia respecto al formulario de registro es que antes se pasaba un objeto vacío y ahora se pasa un objeto con toda la información del usuario:

```
/** @Route("/perfil", name="usuario_perfil") */
public function perfilAction(Request $request)
{
    $usuario = $this->getUser();
    $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);
}
```

Después utiliza la lógica habitual de Symfony para mostrar el formulario o actualizar la información en función del tipo de petición:

```
/** @Route("/perfil", name="usuario_perfil") */
public function perfilAction(Request $request)
{
    $usuario = $this->getUser();
    $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

    $formulario->handleRequest($request);

    if ($formulario->isValid()) {
        // actualizar el perfil del usuario
    }

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

Para probar que el formulario muestra correctamente los datos del usuario *logueado*, crea la plantilla `perfil.html.twig` basándote en el código de la anterior plantilla `registro.html.twig`:

```
{# app/Resources/views/usuario/perfil.html.twig #}
{% extends 'frontend.html.twig' %}

{% block id 'usuario' %}
{% block title %}Ver / Modificar mis datos{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>
```

```
{{ form_start(formulario) }}
    <div class="errors">
        {{ form_errors(formulario) }}
    </div>

    <div>
        {{ form_row(formulario.nombre) }}
    </div>

    <div>
        {{ form_row(formulario.apellidos) }}
    </div>

    {# ... #}

    <div>
        {{ form_errors(formulario.permiteEmail) }}
        {{ form_widget(formulario.permiteEmail) }}
        <span>Me gustaría recibir el boletín de ofertas de Cupon</span>
    </div>
{{ form_end(formulario) }}
{% endblock %}
```

Si pruebas ahora a cargar la página que muestra el perfil del usuario, verás que el formulario contiene un pequeño error. El título del botón para guardar los cambios se llama `Registrarme` en vez de `Guardar cambios`, ya que se está utilizando el mismo formulario que en el registro de usuarios.

Este es el motivo por el que algunos programadores defienden que los botones no deberían añadirse en la clase que define el formulario. Para solucionarlo, existe al menos dos alternativas. La primera solución consiste en eliminar el botón del formulario `UsuarioType` y añadir los botones necesarios en el controlador:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

/** @Route("/usuario") */
class UsuarioController extends Controller
{
    /** @Route("/registro", name="usuario_registro") */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);
        $formulario->add('registrarme', 'submit');
        // ...
    }
```

```
    /** @Route("/perfil", name="usuario_perfil") */
    public function perfilAction(Request $request)
    {
        $usuario = $this->getUser();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);
        $formulario->add('guardar', 'submit', array(
            'label' => 'Guardar cambios'
        ));

        // ...
    }

    // ...
}
```

La segunda solución consiste en **pasar una variable al formulario** para que añade el botón correcto en cada momento. En primer lugar, define en el formulario `UserType` una nueva opción llamada `accion` con un valor por defecto de `modificar_perfil`:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Usuario',
            'accion' => 'modificar_perfil',
        ));
    }

    // ...
}
```

A continuación, utiliza el valor de esta opción dentro del método `buildForm()` del formulario para añadir el botón más adecuado en cada momento:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            // ...
        ;

        if ('crear_usuario' === $options['accion']) {
            $builder->add('registrarme', 'submit', array(
                'label' => 'Registrarme',
            ));
        } elseif ('modificar_perfil' === $options['accion']) {
            $builder->add('guardar', 'submit', array(
                'label' => 'Guardar cambios',
            ));
        }
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Usuario',
            'accion' => 'modificar_perfil',
        ));
    }

    // ...
}
```

Por último, modifica las acciones `perfilAction()` y `registroAction()` para que pasen al formulario el valor correcto de la opción `accion`:

```
// src/AppBundle/Controller/UsuarioController.php
use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

/** @Route("/usuario") */
class UsuarioController extends Controller
```

```
{
    /** @Route("/registro", name="usuario_registro") */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario,
array(
            'accion' => 'crear_usuario',
        ));

        // ...
    }

    /** @Route("/perfil", name="usuario_perfil") */
    public function perfilAction(Request $request)
    {
        $usuario = $this->getUser();
        // en este caso no es obligatorio pasar la opción 'accion' porque su val
or
        // coincide con el valor por defecto definido en el formulario para est
a opción
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario,
array(
            'accion' => 'modificar_perfil',
        ));

        // ...
    }

    // ...
}
```

La última funcionalidad que falta por completar es la lógica que guarda los datos enviados, es decir, la lógica que actualiza el perfil del usuario. Si pruebas a hacer lo siguiente:

```
/** @Route("/perfil", name="usuario_perfil") */
public function perfilAction(Request $request)
{
    $usuario = $this->getUser();
    $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

    $formulario->handleRequest($request);

    if ($formulario->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($usuario);
        $em->flush();

        $this->addFlash('info', 'Los datos de tu perfil se han actualizado corre
```

```
ctamente');

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

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

El código anterior simplemente obtiene los nuevos datos del usuario a través del formulario y los guarda en la base de datos mediante el *entity manager* de Doctrine. Después crea un mensaje *flash* y redirige al usuario a la misma página del perfil para que vea los cambios. Si pruebas el código anterior, pronto descubrirás que tiene un error muy importante: cada vez que actualizas los datos de tu perfil se pierde la contraseña.

8.8.1 Gestionando el cambio de contraseña

Los formularios de la mayoría de aplicaciones web aplican la siguiente lógica para la acción de cambiar la contraseña de un usuario:

- El formulario muestra dos campos repetidos de tipo `password` para que el usuario pueda modificar su contraseña. Estos dos campos se muestran inicialmente vacíos.
- Si el usuario escribe un valor en uno de los campos y deja vacío el otro, se muestra un mensaje de error indicando que los dos campos deben ser iguales.
- Si el usuario escribe dos valores diferentes en cada uno de los campos, se muestra el mismo mensaje de error indicando que los dos campos deben ser iguales.
- Si el usuario no escribe nada en ningún campo de contraseña, se entiende que el usuario no quiere modificarla.
- Si el usuario escribe cualquier valor, pero idéntico en los dos campos, ese valor es su nueva contraseña.

Además, en la aplicación que se está desarrollando, el mismo formulario se reutiliza para registrar usuarios y para modificar su perfil. Por tanto, es necesario añadir una nueva condición:

- Si el usuario se está registrando, es obligatorio que indique su contraseña. Si está modificando su perfil, puede dejar la contraseña vacía para no cambiarla.

Symfony se encarga de comprobar que los dos campos de contraseña tengan el mismo valor y de mostrar un error de validación cuando no se cumpla esta condición. Sin embargo, el resto de condiciones deben ser implementadas por la aplicación.

A continuación se muestra la forma más recomendable de gestionar las contraseñas de los usuarios y sus modificaciones. En primer lugar, añade una nueva propiedad en la entidad `Usuario` llamada `$passwordEnClaro` (en esta entidad ya existe la propiedad `$password`):

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UsuarioRepository")
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @Assert\NotBlank()
     * @Assert\Length(min = 6)
     */
    private $passwordEnClaro;

    /**
     * @ORM\Column(name="password", type="string", length=255)
     */
    private $password;

    // ...

    public function getPasswordEnClaro()
    {
        // ...
    }

    public function setPasswordEnClaro($password)
    {
        // ...
    }
}
```

Observa que `$passwordEnClaro` no incluye ninguna anotación `@ORM\Column`. La razón es que solamente se trata de una propiedad PHP normal y corriente, por lo que su valor no se va a guardar en ninguna columna de la tabla de los usuarios. Esta es una de las grandes ventajas de que las entidades Doctrine sean clases PHP normales, ya que puedes añadir todas las propiedades, métodos y código PHP que necesites para resolver tus problemas.

El otro cambio es que la validación `@Assert\Length` y `@Assert\NotBlank` se ha pasado a la propiedad `$passwordEnClaro` y se ha eliminado de la propiedad `$password`. La idea es utilizar `$passwordEnClaro` como un almacén temporal de la contraseña en claro del usuario y guardar en

`$password` la contraseña codificada con `bcrypt` y que se guarda en la base de datos. Por tanto, ahora es necesario modificar los formularios, controladores y plantillas para que siempre hagan uso de esta propiedad `$passwordEnClaro` con la que el usuario puede establecer su contraseña.

El primer cambio consiste en actualizar el método `eraseCredentials()` de la entidad `Usuario`. Ahora, la propiedad que hay que borrar es `$passwordEnClaro`:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UsuarioRepository")
 */
class Usuario implements UserInterface
{
    // ...

    public function eraseCredentials()
    {
        $this->passwordEnClaro = null;
    }
}
```

Después, actualiza el formulario `UsuarioType` para añadir `passwordEnClaro` en vez de `password` como campo del formulario:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // Antes
        $builder
            // ...
            ->add('password', 'repeated', array(
                // ...
            ))
        ;
```

```
        // Después
        $builder
            // ...
            ->add('passwordEnClaro', 'repeated', array(
                // ...
            ))
        ;
    }
}
```

A continuación, actualiza la plantilla asociada a este formulario:

```
{# app/Resources/views/usuario/perfil.html.twig #}
{% extends 'frontend.html.twig' %}

{% block article %}
    <h1>{{ block('title') }}</h1>

    {{ form_start(formulario) }}

        {# Antes #}
        <div>
            {{ form_widget(formulario.password) }}
        </div>

        {# Después #}
        <div>
            {{ form_widget(formulario.passwordEnClaro) }}
        </div>

    {{ form_end(formulario) }}
{% endblock %}
```

Por último, es necesario actualizar el controlador asociado a la acción `perfilAction()`. Cuando el usuario deje la contraseña vacía, `$passwordEnClaro` vale `null` y no es necesario hacer nada más. Sin embargo, cuando el usuario quiera cambiar su contraseña, el valor de `$passwordEnClaro` ya no es `null` y hay que codificar su valor y guardarlo en la propiedad `$password` para que se actualice en la base de datos.

```
// src/AppBundle/Controller/UsuarioController.php
namespace AppBundle\Controller;

use AppBundle\Entity\Usuario;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

/**
 * @Route("/usuario")
```

```
 */
class UsuarioController extends Controller
{
    // ...

    /**
     * @Route("/perfil", name="usuario_perfil")
     */
    public function perfilAction(Request $request)
    {
        $usuario = $this->getUser();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

        $formulario->handleRequest($request);
        if ($formulario->isValid()) {
            if (null !== $usuario->getPasswordEnClaro()) {
                $encoder = $this->get('security.encoder_factory')
                                ->getEncoder($usuario);
                $passwordCodificado = $encoder->encodePassword(
                    $usuario->getPasswordEnClaro(),
                    null
                );
                $usuario->setPassword($passwordCodificado);
            }

            // ...
        }

        // ...
    }
}
```

Para finalizar la funcionalidad que gestiona las contraseñas, es necesario hacer un pequeño cambio en la gestión de los formularios. Observa el formulario de los usuarios:

```
// src/AppBundle/Entity/Usuario.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UsuarioRepository")
 */
class Usuario implements UserInterface
{
    // ...
```

```
    /**
     * @Assert\NotBlank()
     * @Assert\Length(min = 6)
     */
    private $passwordEnClaro;

    // ...
}
```

La propiedad `$passwordEnClaro` requiere que su valor no sea vacío y tenga al menos 6 caracteres de longitud. Cuando el usuario se registra, esta validación impide que se utilicen contraseñas demasiado cortas. Pero cuando el usuario actualiza su perfil sin modificar su contraseña, su valor será vacío y por tanto, esta validación fallará.

Resulta común que un mismo formulario (ej. `UsuarioType`) se utilice para diferentes funcionalidades (ej. registrar usuarios y actualizar el perfil). Así que Symfony define una funcionalidad llamada **grupos de validación**, que permite activar la validación de un formulario de forma selectiva.

En primer lugar, modifica la validación `@Assert\NotBlank` en la propiedad `$passwordEnClaro` para indicar que solo debe validarse cuando este formulario se utilice para registrar usuarios:

```
// src/AppBundle/Entity/Usuario.php
/**
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UsuarioRepository")
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @Assert\NotBlank(groups={"registro"})
     * @Assert\Length(min = 6)
     */
    private $passwordEnClaro;

    // ...
}
```

Después de este cambio, la validación `@Assert\NotBlank` no se tiene en cuenta a menos que se indique explícitamente que Symfony debe utilizar también el grupo de validación llamado `registro`.

A continuación, actualiza las acciones `registroAction()` y `perfilAction()` del controlador `UsuarioController` para indicar en cada momento qué grupos de validación deben tenerse en cuenta para validar la información del usuario. Esto se indica mediante la opción `validation_groups` al crear el formulario con el método `createForm()`:

```
// src/AppBundle/Controller/UsuarioController.php
/**
 * @Route("/usuario")
 */
class UsuarioController extends Controller
{
    // ...

    /**
     * @Route("/registro", name="usuario_registro")
     */
    public function registroAction(Request $request)
    {
        $usuario = new Usuario();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario,
array(
            'accion' => 'crear_usuario',
            'validation_groups' => array('default', 'registro'),
        ));

        // ...
    }

    /**
     * @Route("/perfil", name="usuario_perfil")
     */
    public function perfilAction(Request $request)
    {
        $usuario = $this->getUser();
        $formulario = $this->createForm('AppBundle\Form\UsuarioType', $usuario);

        // ...
    }
}
```

En la acción `perfilAction()` no se indica ningún grupo de validación, por lo que se aplicará el grupo por defecto, que se llama `default` y al que pertenecen todas las validación que no definen un grupo explícitamente. Por tanto, no se utiliza el grupo `registro`, lo que significa que la contraseña puede dejarse vacía sin que se produzca un error de validación.

Por su parte, la acción `registroAction()` indica que deben utilizarse los grupos `default` y `registro`, por lo que si se deja la contraseña vacía, se producirá un error de validación, que es justo lo que se quiere cuando un usuario se está registrando en la aplicación.

Esta página se ha dejado vacía a propósito

CAPÍTULO 9

RSS y los formatos alternativos

9.1 Formatos alternativos

Las aplicaciones web modernas están preparadas para adaptar sus contenidos a una variedad creciente de dispositivos (ordenadores, móviles y *tablets*) y de formas de consumir la información (RSS, API).

Todos los componentes de Symfony, desde el sistema de enrutamiento hasta las plantillas, están preparados para generar la misma información en diferentes formatos. Si observas por ejemplo el nombre de las plantillas de los capítulos anteriores (`xxx.html.twig`) verás que tienen dos extensiones, siendo la primera el formato en el que se genera la información.

La otra opción clave para soportar formatos alternativos es un parámetro especial del sistema de enrutamiento llamado `_format` (con el guión bajo por delante). El valor de este parámetro indica el formato en el que el usuario desea obtener la información. Cuando una ruta no incluye el parámetro `_format`, Symfony le asigna por defecto el valor `html`. Puedes comprobarlo obteniendo el formato solicitado dentro de cualquier controlador con el siguiente código:

```
public function portadaAction()
{
    $formato = $this->get('request')->getRequestFormat();

    // ...
}
```

Para ofrecer un mismo contenido en varios formatos, añade el parámetro `_format` a su ruta. El siguiente ejemplo modifica la ruta de la página que muestra las ofertas más recientes de una ciudad:

```
// src/AppBundle/Controller/CiudadController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class CiudadController extends Controller
{
    /**
     * @Route("/{ciudad}/recientes.{_format}", name="ciudad_recientes")
```

```
     */
    public function recientesAction(Request $request, $ciudad)
    {
        // ...
    }

    // ...
}
```

Si ahora accedes a alguna página que incluya un enlace con la ruta `ciudad_recientes` verás el siguiente mensaje de error de Twig: *"An exception has been thrown during the rendering of a template ("The "ciudad_recientes" route has some missing mandatory parameters ("_format").")"*. Para solucionar este problema asigna un valor por defecto al parámetro `_format` utilizando la opción `defaults` de la ruta:

```
// src/AppBundle/Controller/CiudadController.php
/**
 * @Route(
 *     "/{ciudad}/recientes.{_format}",
 *     "defaults" = { "_format" = "html" },
 *     name="ciudad_recientes"
 * )
 */
public function recientesAction(Request $request, $ciudad)
{
    // ...
}
```

Ahora que el parámetro `_format` tiene un valor por defecto, al generar la ruta con la función `path()` puedes omitirlo. No obstante, si lo omites, las URL resultantes serán diferentes:

```
{{ path('ciudad_recientes', { ciudad: 'nombre-ciudad' }) }}
{# Genera: /nombre-ciudad/recientes #}

{{ path('ciudad_recientes', { ciudad: 'nombre-ciudad', _format: 'html' }) }}
{# Genera: /nombre-ciudad/recientes.html #}

{{ path('ciudad_recientes', { ciudad: 'nombre-ciudad', _format: 'rss' }) }}
{# Genera: /nombre-ciudad/recientes.rss #}

{{ path('ciudad_recientes', { ciudad: 'nombre-ciudad', _format: 'pdf' }) }}
{# Genera: /nombre-ciudad/recientes.pdf #}
```

Asimismo, haciendo uso de la opción `requirements` de la ruta, puedes restringir los formatos disponibles para un determinado contenido. Siguiendo con el mismo ejemplo, se limitan los formatos disponibles a `html` y `rss`:

```
// src/AppBundle/Controller/CiudadController.php
/**
```

```
     * @Route(
     *     "/{ciudad}/recientes.{_format}",
     *     "defaults" = { "_format" = "html" },
     *     "requirements" = { "_format": "html|rss" },
     *     name="ciudad_recientes"
     * )
     */
    public function recientesAction(Request $request, $ciudad)
    {
        // ...
    }
```

Con la configuración anterior, el siguiente enlace generaría un error en las plantillas Twig, ya que el formato `pdf` no es un formato permitido.

```
{{ path('ciudad_recientes', { ciudad: 'nombre-ciudad', _format: 'pdf' }) }}
```

9.2 Generando el RSS de las ofertas recientes de una ciudad

Algunas páginas de la aplicación ofrecen sus contenidos en formato RSS. Como los enlaces a los archivos RSS se incluyen en la cabecera de las páginas (`<head>`), el primer paso consiste en crear un nuevo bloque llamado `rss` dentro de la sección `<head>` de la plantilla base:

```
{# app/Resources/views/base.html.twig #}

{# ... #}
<head>
    {# ... #}
    {% block rss %}{% endblock %}
</head>
{# ... #}
```

De esta forma, cuando cualquier página de la aplicación quiera enlazar con uno o más canales RSS, simplemente debe incluir los enlaces dentro de un bloque llamado `rss`. Si una página no dispone de RSS, no define ese bloque y la página no incluirá ningún enlace de ese tipo.

El primer canal RSS que se va a definir es el de las ofertas recientes de una ciudad. Abre su plantilla (`ciudad/recientes.html.twig`) y añade el siguiente bloque llamado `rss` para incluir el enlace:

```
{# app/Resources/views/ciudad/recientes.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Ofertas recientes en {{ ciudad.nombre }}{% endblock %}
{% block id 'recientes' %}

{% block rss %}
<link rel="alternate" type="application/rss+xml"
      title="RSS de las ofertas más recientes en {{ ciudad.nombre }}"
      href="{{ path('ciudad_recientes', { ciudad: ciudad.slug, _format: 'rss'
}) }}" />
```

```
{% endblock %}

{# ... #}
```

Si la ruta `ciudad_recientes` incluye el parámetro `_format` (como se ha explicado en la sección anterior), la URL del canal RSS será la siguiente:

```
/nombre-ciudad/recientes.rss
```

Si no has modificado la ruta `ciudad_recientes`, no es necesario que lo hagas. El sistema de enrutamiento de Symfony es tan flexible que si utilizas un parámetro que no está incluido en el patrón de la ruta, se añade en forma de *query string*. De esta forma, la URL del canal RSS en este caso sería:

```
/nombre-ciudad/recientes?_format=rss
```

Aunque las dos URL tienen un aspecto diferente, su funcionamiento es idéntico, así que sólo es cuestión de gustos la forma en la que se incluye el formato dentro de la URL.

Una vez creado el enlace al canal RSS, el siguiente paso consiste en refactorizar el controlador. Como la información es la misma en los dos casos (HTML y RSS) y sólo cambia la forma en la que se presenta, los cambios en el controlador son mínimos:

```
// src/AppBundle/Controller/CiudadController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class CiudadController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}/recientes", name="ciudad_recientes")
     */
    public function recientesAction(Request $request, $ciudad)
    {
        // ...

        $formato = $request->getRequestFormat();

        return $this->render(
            'ciudad/recientes.'.$formato.'.twig',
            array(...)
        );
    }
}
```

El controlador obtiene en primer lugar el formato de la petición mediante el método `getRequestFormat()`. Después, utiliza el formato para determinar la plantilla a renderizar: `recientes.html.twig` para las páginas HTML *normales* y `recientes.rss.twig` para crear el archivo RSS. El resto de la lógica del controlador permanece inalterada.

Para desarrollar la plantilla del archivo RSS, en primer lugar crea el archivo `recientes.rss.twig` y añade el siguiente código XML que forma la estructura básica de un archivo RSS formato 2.0:

```
<!-- app/Resources/views/ciudad/recientes.rss.twig -->
<?xml version="1.0"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
    <channel>
        <title>Ofertas recientes</title>
        <link>## URL absoluta de la página HTML original ##</link>
        <description>Las ofertas más recientes en ## CIUDAD ##</description>
        <language>## IDIOMA de los contenidos del RSS ##</language>
        <pubDate>## FECHA de publicación (formato RFC 2822) ##</pubDate>
        <lastBuildDate>## FECHA de actualización (RFC 2822) ##</lastBuildDate>
        <generator>Symfony</generator>
        <atom:link href="## URL absoluta de este archivo RSS ##"
                   rel="self" type="application/rss+xml" />
        <!-- Repetir lo siguiente para cada oferta incluida -->
        <item>
            <title>## TÍTULO de la oferta ##</title>
            <link>## URL absoluta de la página de la oferta ##</link>
            <description>## IMAGEN y DESCRIPCION de la oferta ##</description>
            <pubDate>## FECHA de publicación (formato RFC 2822) ##</pubDate>
            <guid>## URL absoluta de la página de la oferta ##</guid>
        </item>
        <!-- Fin de la repetición -->
    </channel>
</rss>
```

Sabiendo que el controlador pasa a la plantilla las variables `ciudad` y `ofertas` y haciendo uso de las funciones y filtros de Twig, es sencillo completar la plantilla. La única precaución que se debe tener en cuenta es que las URL de los enlaces que incluye el RSS siempre deben ser absolutas, ya que los contenidos RSS siempre se consumen fuera del sitio web.

```
{# app/Resources/views/ciudad/recientes.rss.twig #}
<?xml version="1.0"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
    <channel>
        <title>Ofertas recientes en {{ ciudad.nombre }}</title>
        <link>{{ url('oferta_recientes', { ciudad: ciudad.slug }) }}</link>
        <description>Las ofertas más recientes publicadas por Cupon en
                    {{ ciudad.nombre }}</description>
        <language>{{ app.request.locale }}</language>
        <pubDate>{{ 'now'|date('r') }}</pubDate>
        <lastBuildDate>{{ 'now'|date('r') }}</lastBuildDate>
```

```
        <generator>Symfony</generator>
        <atom:link rel="self" type="application/rss+xml"
                   href="{{ url('oferta_recientes', {
                       ciudad:  ciudad.slug,
                       _format: 'rss' }) }}" />

    {# ... #}
```

Observa cómo el código anterior utiliza la función `url()` en vez de `path()` para generar URL absolutas. Además, recuerda que el filtro `date()` de Twig soporta cualquier opción de formato de la función `date()` de PHP. Así que para generar fechas en formato RFC 2822 sólo es necesario indicar la letra `r` como formato. Por último, el idioma activo en la aplicación se obtiene mediante la sesión del usuario (`app.request.locale`).

La segunda parte de la plantilla es un bucle que recorre todas las ofertas y genera un elemento `<item>` para cada una:

```
{# app/Resources/views/ciudad/recientes.rss.twig #}
<?xml version="1.0"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
<channel>
    {# ... #}

    {% for oferta in ofertas %}
    <item>
        <title>{{ oferta.nombre }}</title>
        <link>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</link>
        <description><![CDATA[
            <img alt="" src="{{ asset(oferta.rutaFoto, 'ofertas') }}"/>
            {{ oferta.descripcion|mostrar_como_lista }}
            <a href="#">Comprar</a>
        ]]></description>
        <pubDate>{{ oferta.fechaPublicacion|date('r') }}</pubDate>
        <guid>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</guid>
    </item>
    {% endfor %}
</channel>
</rss>
```

Si ahora accedes a la página de ofertas recientes de una ciudad, verás que el navegador muestra el icono RSS indicando que la página dispone de al menos un canal RSS. Si pinchas sobre ese icono, verás correctamente los contenidos del archivo RSS.

En realidad, el archivo RSS sólo se ve bien en el ordenador en el que estás desarrollando la aplicación. En cualquier otro ordenador no se verán las imágenes, ya que la función `asset()` no genera URL absolutas. La solución consiste en utilizar la función `absolute_url()` de Twig, que convierte

la URL relativa que se le pasa en una URL absoluta (añadiendo el host, el `https://` si es necesario, etc.)

```
{# antes #}
<img alt="..." src="{{ asset(oferta.rutaFoto, 'ofertas') }}"/>

{# después #}
<img alt="..." src="{{ absolute_url(asset(oferta.rutaFoto, 'ofertas')) }}"/>
```

Juntando todo lo anterior, la plantilla `recientes.rss.twig` definitiva tiene el siguiente aspecto:

```
{# app/Resources/views/ciudad/recientes.rss.twig #}
<?xml version="1.0"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
    <channel>
        <title>Ofertas recientes en {{ ciudad.nombre }}</title>
        <link>{{ url('ciudad_recientes', { ciudad: ciudad.slug }) }}</link>
        <description>Las ofertas más recientes publicadas por Cupon en
                    {{ ciudad.nombre }}</description>
        <language>{{ app.request.locale }}</language>
        <pubDate>{{ 'now'|date('r') }}</pubDate>
        <lastBuildDate>{{ 'now'|date('r') }}</lastBuildDate>
        <generator>Symfony</generator>
        <atom:link rel="self" type="application/rss+xml"
                   href="{{ url('ciudad_recientes', {
                        ciudad:  ciudad.slug,
                        _format: 'rss' })
                    }}" />

        {% for oferta in ofertas %}
        <item>
            <title>{{ oferta.nombre }}</title>
            <link>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                     slug:   oferta.slug }) }}</link>
            <description><![CDATA[
                <img alt="" src="{{ absolute_url(asset(oferta.rutaFoto, 'oferta
s')) }}"/>
                {{ oferta.descripcion|mostrar_como_lista }}
                <a href="#">Comprar</a>
            ]]></description>
            <pubDate>{{ oferta.fechaPublicacion|date('r') }}</pubDate>
            <guid>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                     slug:   oferta.slug }) }}</guid>
        </item>
        {% endfor %}
    </channel>
</rss>
```

9.3 Generando el RSS de las ofertas recientes de una tienda

Cuando la aplicación genera varios canales RSS, es una buena idea disponer de una plantilla base de la que hereden todas las plantillas de RSS. Así que antes de crear el segundo canal RSS de la aplicación, crea una plantilla llamada `base.rss.twig` en el directorio `app/Resources/views`:

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
    <channel>
        <title>{% block title %}{% endblock %}</title>
        <link>{% block url %}{% endblock %}</link>
        <description>{% block descripcion %}{% endblock %}</description>

        <language>{% block idioma %}
            {{ app.request.locale }}
        {% endblock %}</language>

        <pubDate>{% block fechaPublicacion %}
            {{ 'now'|date('r') }}
        {% endblock %}</pubDate>

        <lastBuildDate>{% block fechaCreacion %}
            {{ 'now'|date('r') }}
        {% endblock %}</lastBuildDate>

        <generator>Symfony</generator>
        <atom:link href="{% block self %}{% endblock %}"
                   rel="self" type="application/rss+xml" />
        {% block items %}{% endblock %}
    </channel>
</rss>
```

Haciendo uso de esta plantilla base, la plantilla `recientes.rss.twig` generada en la sección anterior se puede refactorizar de la siguiente manera:

```
{# app/Resources/views/ciudad/recientes.rss.twig #}
{% extends 'base.rss.twig' %}

{% block title %}{% spaceless %}
    Cupon - Ofertas recientes en {{ ciudad.nombre }}
{% endspaceless %}{% endblock %}

{% block url %}{% spaceless %}
    {{ url('ciudad_recientes', { ciudad: ciudad.slug }) }}
{% endspaceless %}{% endblock %}

{% block descripcion %}{% spaceless %}
    Las ofertas más recientes publicadas por Cupon en {{ ciudad.nombre }}
{% endspaceless %}{% endblock %}
```

```
{% block self %}{% spaceless %}
    {{ url('ciudad_recientes', { ciudad: ciudad.slug, _format: 'rss' }) }}
{% endspaceless %}{% endblock %}

{% block items %}
    {% for oferta in ofertas %}
    <item>
        <title>{{ oferta.nombre }}</title>
        <link>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</link>
        <description><![CDATA[
            <img alt="" src="{{ absolute_url(asset(oferta.rutaFoto, 'ofertas'))
}}"/>
            {{ oferta.descripcion|mostrar_como_lista }}
            <a href="#">Comprar</a>
        ]]></description>
        <pubDate>{{ oferta.fechaPublicacion|date('r') }}</pubDate>
        <guid>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</guid>
    </item>
    {% endfor %}
{% endblock %}
```

Después de estos cambios, añadir un canal RSS para las ofertas recientes de una tienda es muy sencillo. Abre el controlador `TiendaController` y añade lo siguiente:

```
// src/AppBundle/Controller/TiendaController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class TiendaController extends Controller
{
    /**
     * @Route("/{ciudad}/tiendas/{tienda}", requirements={ "ciudad" = ".+" }, na
me="tienda_portada")
     */
    public function portadaAction($ciudad, $tienda)
    {
        // ...

        $formato = $this->get('request')->getRequestFormat();
        return $this->render(
            'tienda/portada.'.$formato.'.twig',
            array(...)
        );
    }
}
```

Después, añade un enlace al canal RSS en la plantilla `portada.html.twig`:

```
{# app/Resources/views/tienda/portada.html.twig #}
{% extends 'frontend.html.twig' %}

{% block title %}Tienda {{ tienda.nombre }}{% endblock %}
{% block id 'tienda' %}

{% block rss %}
<link rel="alternate" type="application/rss+xml"
      title="RSS de las ofertas más recientes de {{ tienda.nombre }}"
      href="{{ path('tienda_portada', { ciudad: tienda.ciudad.slug, tienda: tien
da.slug, _format: 'rss' }) }}" />
{% endblock %}

{# ... #}
```

Y por último, crea la plantilla `portada.rss.twig`:

```
{# app/Resources/views/tienda/portada.rss.twig #}
{% extends 'base.rss.twig' %}

{% block title %}{% spaceless %}
    Cupon - Las ofertas más recientes de {{ tienda.nombre }}
{% endspaceless %}{% endblock %}

{% block url %}{% spaceless %}
    {{ url('tienda_portada', { ciudad: tienda.ciudad.slug,
                               tienda: tienda.slug }) }}
{% endspaceless %}{% endblock %}

{% block descripcion %}{% spaceless %}
    Las ofertas más recientes de {{ tienda.nombre }}
{% endspaceless %}{% endblock %}

{% block self %}{% spaceless %}
    {{ url('tienda_portada', { ciudad:  tienda.ciudad.slug,
                               tienda:  tienda.slug,
                               _format: 'rss' }) }}
{% endspaceless %}{% endblock %}

{% block items %}
    {% for oferta in ofertas %}
    <item>
        <title>{{ oferta.nombre }}</title>
        <link>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</link>
        <description><![CDATA[
            <img alt="" src="{{ absolute_url(asset(oferta.rutaFoto, 'ofertas'))
```

```
}}"/>
            {{ oferta.descripcion|mostrar_como_lista }}
            <a href="#">Comprar</a>
        ]]></description>
        <pubDate>{{ oferta.fechaPublicacion|date('r') }}</pubDate>
        <guid>{{ url('oferta', { ciudad: oferta.ciudad.slug,
                                 slug:   oferta.slug }) }}</guid>
    </item>
    {% endfor %}
{% endblock %}
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 10

Internacionalizando el sitio web

La internacionalización o *i18n* es el conjunto de acciones encaminadas a traducir y adaptar el sitio web a diferentes idiomas y países. La combinación del idioma y país de un usuario se denomina *locale*. Gracias al *locale* las aplicaciones pueden soportar las variaciones idiomáticas, como sucede por ejemplo con el español (España, México, Argentina, Colombia, Venezuela, etc.) o el inglés (Reino Unido, Estados Unidos, Australia, etc.)

Los *locales* de Symfony utilizan el formato estándar que concatena mediante un guión bajo el código de dos letras del idioma (estándar ISO 639-1) y el código de dos letras del país (estándar ISO 3166). Ejemplos:

- `es_ES`, español de España
- `es_AR`, español de Argentina
- `fr_BE`, francés de Bélgica
- `en_AU`, inglés de Australia

En muchas ocasiones las aplicaciones web no diferencian por idioma y país, sino simplemente por idioma. En ese caso, el *locale* coincide con el código del idioma (`es`, `en`, `ca`, `de`, `ja`, etc.)

10.1 Configuración inicial

Symfony incluye tres opciones de configuración relacionadas con la internacionalización. La primera se define en el archivo `app/config/parameters.yml`:

```
# app/config/parameters.yml
parameters:
    # ...
    locale: es
```

Esta opción es la más importante de todas, ya que su valor se utiliza en otras partes y opciones de configuración de la aplicación. Las otras dos opciones se configuran en el archivo `app/config/config.yml`:

```
# app/config/config.yml

# ...
framework:
    translator:      { fallback: es }
```

```
    default_locale:  %locale%
    # ...
```

La opción `fallback` indica el idioma al que se traduce un contenido cuando el idioma solicitado por el usuario no está disponible. Si la aplicación utiliza por ejemplo `es_AR` como *locale*, el valor de la opción `fallback` podría ser `es`, para que el mensaje se muestre al menos en español.

La otra opción de configuración es `default_locale`, que por defecto toma el mismo valor que la opción `locale` del archivo `parameters.yml`. Esta opción indica el *locale* que se asigna al usuario cuando la aplicación no lo establece explícitamente utilizando el siguiente código:

```
class DefaultController extends Controller
{
    public function indexAction()
    {
        $this->getRequest()->setLocale('es_ES');

        // ...
    }
}
```

El código anterior establece `es_ES` como *locale* del usuario, por lo que se ignora la opción `default_locale`. Para determinar el *locale* del usuario activo en la aplicación, emplea el método `getLocale()`:

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function portadaAction(Request $request)
    {
        // ...

        $locale = $request->getLocale();
    }
}
```

En las plantillas Twig también puedes obtener el valor del *locale* con el siguiente código:

```
{% set locale = app.request.locale %}
```

10.2 Rutas internacionalizadas

Si la aplicación ofrece los mismos contenidos en varios idiomas, la ruta de una misma página debería ser diferente para cada idioma. Así, la página `/contacto` original debería transformarse en `/es_AR/contacto`, `/es_ES/contacto`, `/en/contacto`, etc.

Para facilitar al máximo esta tarea, el sistema de enrutamiento de Symfony incluye una variable especial llamada `_locale` (con un guión bajo por delante). Si la añades al patrón de la ruta, Symfony se encargará de asignarle el valor adecuado para cada usuario:

```
/**
 * @Route("/{_locale}/contacto", name="contacto")
 */
public function contactoAction()
{
    // ...
}
```

Si el *locale* del usuario es `en_US`, al generar la ruta con `{{ path('contacto') }}`, el resultado será `/en_US/contacto`. Si accede a la aplicación un usuario con el *locale* igual a `es`, la misma plantilla generará la ruta `/es/contacto`.

Además, la variable especial `_locale` también funciona *a la inversa*. El valor de `{_locale}` dentro de una URL se establece automáticamente como valor del *locale* del usuario. Así que si te encuentras en la página `/es/contacto` y modificas `/es/` por `/en/`, toda la aplicación se mostrará en inglés.

Internacionalizar todas las rutas de la aplicación implicaría añadir la variable `_locale` en todas las anotaciones `@Route()`. Para evitar todo este trabajo manual, puedes utilizar la opción `prefix` del sistema enrutamiento. Cuando importas rutas, esta opción indica el prefijo que se añade a todas las URL importadas. Así que puedes internacionalizar todas las rutas definidas como anotaciones en el *bundle* AppBundle de la siguiente manera:

```
# app/config/routing.yml
# ...
app:
    resource: '@AppBundle/Controller/'
    type:     annotation
    prefix:   /{_locale}
    defaults:
        _locale: '%locale%'
```

Si ahora pruebas a navegar por el sitio web, verás que todas las URL de la aplicación incluyen al principio el valor del `_locale`. Además, gracias a la opción `defaults`, las rutas se muestran por defecto en el idioma configurado en el parámetro `%locale%`.

10.2.1 Restringiendo los idiomas disponibles

Si la aplicación solamente soporta unos pocos idiomas o si algunas traducciones se encuentran a medias y por tanto no se pueden ver en producción, deberías restringir los posibles valores de `_locale` utilizando la opción `requirements`:

```
# app/config/routing.yml
# ...
app:
```

```
    resource: '@AppBundle/Controller/'
    type:     annotation
    prefix:   /{_locale}
    defaults:
        _locale: '%locale%'
    requirements:
        _locale: en|es
```

Si ahora tratas de acceder por ejemplo a la página `/fr/sitio/contacto` la aplicación mostrará el mensaje de error *"No route found for GET /fr/sitio/contacto"*

10.2.2 Actualizando la configuración de seguridad

Como sabes, la configuración de seguridad de las aplicaciones Symfony se basa en definir *firewalls* y restringir el acceso en función de las URL. Las partes más relevantes del archivo de configuración de la seguridad son las siguientes:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            pattern:     ^/
            anonymous:   ~
            form_login:  ~

    access_control:
        - { path: ^/usuario/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/registro, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/usuario/*, roles: ROLE_USUARIO }

    # ...
```

Como el *firewall* `frontend` cubre todas las URL de la aplicación mediante el patrón `^/`, no se ve afectado por los cambios introducidos por la internacionalización. Sin embargo, el control de acceso ya no funciona como debería, porque las URL ahora son `/{_locale}/usuario/*` en vez de `/usuario/*`.

Siguiendo con la misma configuración anterior en la que los únicos dos idiomas permitidos en la aplicación son `es` y `en`, los cambios necesarios serían los siguientes:

```
# app/config/security.yml
security:
    # ...

    access_control:
        - { path: ^/(es|en)/usuario/login,
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(es|en)/usuario/registro,
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(es|en)/usuario/*, roles: ROLE_USUARIO }
```

```
    # ...
```

Si el número de idiomas es muy grande o varía frecuentemente, es mejor utilizar una expresión regular:

```
# app/config/security.yml
security:
    # ...

    access_control:
        - { path: '^/[a-z]{2}/usuario/login',
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: '^/[a-z]{2}/usuario/registro',
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: '^/[a-z]{2}/usuario/*', roles: ROLE_USUARIO }

    # ...
```

10.2.3 Traduciendo las rutas de la aplicación

Lamentablemente, Symfony no permite traducir los patrones de las rutas. Así que aunque traduzcas el sitio web al inglés, la ruta de un oferta por ejemplo será `/en/{ciudad-en-español}/oferta/{slug-en-español}` y no `/en/{ciudad-en-inglés}/offer/{slug-en-inglés}`.

Si quieres traducir las URL de tus aplicaciones, utiliza el *bundle* BeSimpleI18nRoutingBundle (https://github.com/BeSimple/BeSimpleI18nRoutingBundle) .

10.2.4 Añadiendo un selector de idiomas

Cambiar el valor del *locale* en la URL de la página no es la forma más intuitiva de que los usuarios cambien el idioma del sitio web. Así que abre la plantilla base de la aplicación y añade el siguiente código para mostrar un selector de idioma en el pie de las páginas:

```
{# app/Resources/views/base.html.twig #}

{# ... #}

<footer>
    {# ... #}

    {% set locale = app.request.locale %}

    {% if locale == 'es' %}
        <span>Español</span>
        <a href="{{ path('portada', { _locale: 'en' }) }}">English</a>
    {% elseif locale == 'en' %}
        <a href="{{ path('portada', { _locale: 'es' }) }}">Español</a>
        <span>English</span>
```

```
    {% endif %}
</footer>

{# ... #}
```

10.3 Traduciendo contenidos estáticos

Después de actualizar las rutas, el siguiente elemento a traducir son los contenidos estáticos de las páginas del sitio web. Estos son los contenidos que no dependen de la información de la base de datos, como por ejemplo los menús de navegación, los nombres de las secciones, los formularios, las páginas estáticas, etc.

10.3.1 Traducciones en plantillas

La primera plantilla que se debe traducir es `frontend.html.twig`, de la que heredan todas las páginas del *frontend* y que incluye elementos tan importantes como el menú principal de navegación:

```
{# app/Resources/views/frontend.html.twig #}

{# ... #}
<ul>
    <li><a href="...">Oferta del día</a></li>
    <li><a href="...">Ofertas recientes</a></li>
    <li><a href="...">Mis ofertas</a></li>
</ul>
{# ... #}
```

La forma más sencilla de traducir los contenidos estáticos de una plantilla consiste en aplicar el filtro `trans` de Twig a cada cadena de texto que se quiere traducir:

```
{# app/Resources/views/frontend.html.twig #}
<ul>
    <li><a href="...">{{ "Oferta del día"|trans }}</a></li>
    <li><a href="...">{{ "Ofertas recientes"|trans }}</a></li>
    <li><a href="...">{{ "Mis ofertas"|trans }}</a></li>
</ul>
```

Si el texto es muy largo, resulta más cómodo utilizar la etiqueta `{% trans %}`:

```
{# app/Resources/views/frontend.html.twig #}
<ul>
    <li><a href="...">{% trans %}Oferta del día{% endtrans %}</a></li>
    <li><a href="...">{% trans %}Ofertas recientes{% endtrans %}</a></li>
    <li><a href="...">{% trans %}Mis ofertas{% endtrans %}</a></li>
</ul>
```

10.3.2 Catálogos de traducciones

Después de *marcar* las cadenas a traducir con el filtro `trans` o con la etiqueta `{% trans %}`, el siguiente paso consiste en crear las traducciones de los contenidos a los diferentes idiomas.

Las traducciones en Symfony se gestionan mediante catálogos, que no son más que archivos de texto en formato XLIFF, PHP o YAML. Estos archivos son los que contienen las traducciones a cada idioma de las diferentes cadenas de texto de las plantillas. Por defecto el nombre de los catálogos es `messages` seguido del valor del *locale* y del formato del archivo:

- `messages.en.xliff`, traducción al inglés en formato XLIFF.
- `messages.es.yml`, traducción al español en formato YAML.
- `messages.fr.php`, traducción al francés en formato PHP.

Por convención, los catálogos se guardan en el directorio `Resources/translations/` de cada *bundle* o si lo prefieres, en el directorio global `app/Resources/translations/`.

El formato XLIFF es el recomendado por Symfony para crear los catálogos y también es el formato más compatible con las herramientas que utilizan los servicios profesionales de traducción. Si el catálogo lo creas tú mismo, es recomendable utilizar el formato YAML por ser el más conciso.

En el siguiente ejemplo se traducen al inglés los contenidos estáticos de la portada del sitio. El archivo se crea en `app/Resources/translations/messages.en.xliff` pero también podría encontrarse en `src/AppBundle/Resources/translations/messages.en.xliff`:

```
<!-- app/Resources/translations/messages.en.xlf -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="es" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Oferta del día</source>
                <target>Daily deal</target>
            </trans-unit>
            <trans-unit id="2">
                <source>Ofertas recientes</source>
                <target>Recent offers</target>
            </trans-unit>
            <trans-unit id="3">
                <source>Mis ofertas</source>
                <target>My offers</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Al recargar la portada del sitio, seguirás viendo los mensajes en español. Pero si cambias el valor `es` por `en` en la ruta de la portada, verás cómo ahora el menú principal de navegación se muestra en inglés. Si utilizas la caché de HTTP (como se explica más adelante), recuerda que debes borrar la caché de la aplicación antes de poder probar los cambios en la internacionalización.

Aunque la traducción funciona correctamente, tiene una limitación que podría convertirse en un problema si traduces el sitio a muchos idiomas. Si quieres modificar por ejemplo el texto *Oferta*

del día por *Oferta diaria*, debes buscar y modificar el texto original en todas las plantillas. Además, también debes buscarlo y cambiarlo en todos los catálogos de traducción de todos los idiomas de la aplicación.

Para evitar este problema, puedes utilizar claves como texto de las plantillas:

```
<ul>
    <li><a href="...">{{ 'menu.dia'|trans }}</a></li>
    <li><a href="...">{{ 'menu.recientes'|trans }}</a></li>
    <li><a href="...">{{ 'menu.mias'|trans }}</a></li>
</ul>
```

No olvides encerrar las claves entre comillas para que Twig las interprete como cadenas de texto y no como objetos y propiedades. Ahora ya puedes utilizar las claves en cualquier catálogo de traducción:

```
<!-- app/Resources/translations/messages.en.xlf -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="es" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>menu.dia</source>
                <target>Daily deal</target>
            </trans-unit>
            <trans-unit id="2">
                <source>menu.recientes</source>
                <target>Recent offers</target>
            </trans-unit>
            <trans-unit id="3">
                <source>menu.mias</source>
                <target>My offers</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Obviamente, cuando se utilizan claves también hay que crear un catálogo que traduzca las claves a las cadenas de texto del idioma original (en este caso, el español):

```
<!-- app/Resources/translations/messages.es.xlf -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="es" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>menu.dia</source>
                <target>Oferta del día</target>
            </trans-unit>
```

```
            <trans-unit id="2">
                <source>menu.recientes</source>
                <target>Ofertas recientes</target>
            </trans-unit>
            <trans-unit id="3">
                <source>menu.mias</source>
                <target>Mis ofertas</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Después de reemplazar las cadenas de texto por claves, ya puedes modificar por ejemplo el texto *Oferta del día* cambiando una única traducción en un único catálogo, propagándose el cambio de forma instantánea en todas las plantillas de la aplicación.

Cuando se utiliza el formato YAML para los catálogos, es conveniente utilizar claves compuestas separadas por puntos (como en el ejemplo anterior), ya que simplifica mucho la creación del catálogo:

```
# app/Resources/translations/messages.es.yml
menu:
    dia:       Oferta del día
    recientes: Ofertas recientes
    mias:      Mis ofertas
```

Si la aplicación es muy compleja, puede ser necesario dividir el catálogo de traducción en diferentes archivos. Estos *trozos de catálogo* se llaman *dominios*. El dominio por defecto es `messages`, de ahí el nombre por defecto de los catálogos. Puedes crear tantos archivos como necesites y puedes nombrarlos como quieras, por ejemplo:

```
messages.en.xliff
menus.en.xliff
extranet.en.xliff
administracion.en.xliff
```

Si divides el catálogo en varios dominios, debes indicar siempre el dominio al traducir los contenidos de la plantilla:

```
{# Las traducciones se encuentran en
   app/Resources/translations/menus.en.xliff #}

<ul>
    <li><a href="...">{{ "Oferta del día"|trans({}, 'menus') }}</a></li>
    <li><a href="...">{{ "Ofertas recientes"|trans({}, 'menus') }}</a></li>
    <li><a href="...">{{ "Mis ofertas"|trans({}, 'menus') }}</a></li>
</ul>

<ul>
```

```
    <li><a href="...">
        {% trans from 'menus' %}Oferta del día{% endtrans %}
    </a></li>
    <li><a href="...">
        {% trans from 'menus' %}Ofertas recientes{% endtrans %}
    </a></li>
    <li><a href="...">
        {% trans from 'menus' %}Mis ofertas{% endtrans %}
    </a></li>
</ul>
```

La traducción siempre se realiza al *locale* de la petición actual o en su defecto, al valor definido en la opción `fallback` del servicio `translator`. No obstante, también puedes forzar la traducción a un determinado idioma indicándolo como tercer parámetro del filtro `trans()` o mediante la palabra clave `into` de la etiqueta `{% trans %}`:

```
{# Como filtro #}
{{ "Oferta del día"|trans({...}, 'menus', 'de_DE') }}

{# Como etiqueta #}
{% trans with {...} from 'menus' into 'de_DE' %}Oferta del día{% endtrans %}
```

10.3.3 Traducciones en controladores

La mayoría de traducciones de contenidos estáticos se realiza en las propias plantillas, pero en ocasiones también se necesitan traducir contenidos en los controladores. Todo lo explicado anteriormente es válido, pero la traducción se realiza a través del método `trans()` del servicio `translator`:

```
public function portadaAction($ciudad)
{
    // ...

    // Traducción de cadenas de texto
    $titulo = $this->get('translator')->trans('Oferta del día');

    // Traducción a través de claves
    $titulo = $this->get('translator')->trans('menu.dia');
}
```

El dominio o catálogo específico que se debe utilizar para la traducción se indica como tercer parámetro del método `trans()`. Por el momento añade un array vacío como segundo parámetro, ya que su utilidad se explicará más adelante:

```
public function portadaAction($ciudad)
{
    // ...

    $titulo = $this->get('translator')->trans(
        'Oferta del día', array(), 'menus'
```

```
    );
}
```

La traducción siempre se realiza al *locale* de la petición actual o en su defecto, al valor definido en la opción `fallback` del servicio `translator`. No obstante, también puedes indicar el *locale* explícitamente como cuarto parámetro del método `trans()`:

```
public function portadaAction($ciudad)
{
    // ...

    // La cadena se traduce al alemán
    $titulo = $this->get('translator')->trans(
        'Oferta del día', array(), 'messages', 'de_DE'
    );
}
```

10.3.4 Traducciones con variables

Si la cadena de texto contiene partes variables, la traducción no es posible con los métodos explicados en las secciones anteriores. Considera por ejemplo el siguiente código de una plantilla Twig que muestra cuánto tiempo falta para que caduque una oferta:

```
<strong>Faltan</strong>: {{ oferta.fechaExpiracion }}
```

Cuando la cadena a traducir tiene partes variables, se define una variable para cada una de ellas. El nombre de las variables sigue el formato `%nombre-variable%`, como muestra el siguiente código:

```
{# Utilizando el filtro trans() #}
{{ "<strong>Faltan</strong>: %fecha%"|trans(
    { '%fecha%': oferta.fechaExpiracion }
) }}

{# Utilizando la etiqueta {% trans %} #}
{% trans with { '%fecha%' : oferta.fechaExpiracion } %}
    <strong>Faltan</strong>: %fecha%
{% endtrans %}
```

La cadena de texto de este ejemplo, además de partes variables, contiene etiquetas HTML. Así que si utilizas el formato XLIFF para tus catálogos, no olvides encerrar el contenido de la cadena en una sección `CDATA`:

```
<!-- app/Resources/translations/messages.en.xlf -->
<trans-unit id="16">
    <source><![CDATA[<strong>Faltan</strong>: %fecha%]]></source>
    <target><![CDATA[%fecha% <strong>left</strong>]]></target>
</trans-unit>
```

En los controladores, la traducción con variables sigue una notación similar, pasando el valor de las variables en forma de array como segundo parámetro:

```
public function portadaAction($ciudad)
{
    // ...

    $cadena = $this->get('translator')->trans(
        'La oferta caduca el %fecha%',
        array('%fecha%' => $oferta->getFechaExpiracion())
    );
}
```

10.3.5 Traducciones con valores plurales

Los idiomas creados por humanos contienen numerosas excepciones e irregularidades. Una de las más importantes es el uso de los plurales. Si el texto a traducir por ejemplo es *"Has comprado N ofertas"*, cuando `N` sea `1`, el texto se debe sustituir por *"Has comprado 1 oferta"*.

Symfony se encarga automáticamente de estos detalles mediante la etiqueta `transchoice` en las plantillas y el método `transChoice()` en los controladores:

```
{% transchoice ofertas|length with { '%total%' : ofertas|length } %}
    Has comprado una oferta | Has comprado %total% ofertas
{% endtranschoice %}
```

```
$cadena = $this->get('translator')->transChoice(
    'Has comprado una oferta | Has comprado %total% ofertas',
    count($ofertas),
    array('%ofertas%' => count($ofertas))
);
```

Las cadenas que varían según el plural se indican con todas sus variantes separadas por una barra vertical `|`. Symfony se encarga de elegir la variante correcta en función del valor que se pasa como primer parámetro de `{% transchoice %}` o como segundo parámetro de `transChoice()`.

En el catálogo de traducciones, la cadena original se escribe tal y como se indica en la plantilla o en el controlador:

```
<!-- app/Resources/translations/messages.en.xlf -->
<trans-unit id="17">
    <source>Has comprado una oferta | Has comprado %total% ofertas</source>
    <target>One offer purchased | %total% offers purchased</target>
</trans-unit>
```

Algunos casos requieren más de dos variantes en función del plural, como por ejemplo para tratar de forma especial el valor `0` o los valores negativos. En tal caso se pueden indicar para qué valores se aplica cada variante:

```
{% transchoice ofertas|length with { '%total%' : ofertas|length %}
    {0} No has comprado ninguna oferta | {1} Has comprado una oferta | ]1,Inf] H
as comprado %total% ofertas
{% endtranschoice %}
```

```
$cadena = $this->get('translator')->transChoice(
    '{0} No has comprado ninguna oferta | {1} Has comprado una oferta | ]1,Inf]
Has comprado %total%',
    $numeroOfertas,
    array('%ofertas%' => $numeroOfertas)
);
```

La notación `{1}` indica que el valor debe ser exactamente `1`, mientras que `]1,Inf]` indica cualquier valor entero mayor que `1` y menor o igual que infinito. Esta notación se define en el estándar ISO 31-11 (http://en.wikipedia.org/wiki/Interval_%28mathematics%29#The_ISO_notation) .

Más allá del uso básico de plurales, esta notación permite variar los mensajes mostrados en función de alguna cantidad o valor almacenado en una variable:

```
{% set faltan = oferta.umbral - oferta.compras %}

{% if faltan > 0 %}

    {% transchoice faltan with { '%faltan%' : faltan } %}
    {1}  Una sola compra más activa la oferta!|[1, 9]  Sólo faltan %faltan% comp
ras para activar la oferta!|]9,Inf] Faltan %faltan% compras para activar la ofer
ta
    {% endtranschoice %}

{% else %}
    {# ... #}
{% endif %}
```

Y la traducción de la cadena de texto anterior en el catálogo:

```
<!-- app/Resources/translations/messages.en.xlf -->
<trans-unit id="17">
    <source>{1}  Una sola compra más activa la oferta!|[1, 9]  Sólo faltan %falt
an% compras para activar la oferta!|]9,Inf] Faltan %faltan% compras para activa
r la oferta</source>
    <target>{1} Just one more purchase needed to get the deal!|[1, 9] Just %falt
an% more needed to get the deal!|]9,Inf] %faltan% more needed to get the deal</t
arget>
</trans-unit>
```

10.4 Traduciendo contenidos dinámicos

Internacionalizar completamente un sitio web también requiere traducir todos los contenidos dinámicos almacenados en la base de datos. Aunque ni Symfony ni Doctrine incluyen esta carac-

terística, puedes hacer uso del *bundle* StofDoctri neExtensionsBundle (https://github.com/stof/StofDoctrineExtensionsBundle) .

10.5 Traduciendo páginas estáticas

Las páginas estáticas son aquellas páginas cuyos contenidos no se obtienen de una base de datos, sino que se incluyen en la propia plantilla. Como pueden incluir muchos contenidos, no es posible en la práctica utilizar el filtro `trans` o la etiqueta `{% trans %}` de Twig.

Resulta preferible crear una página/plantilla por cada idioma en el que se ofrezca el contenido. A continuación se muestra cómo traducir las páginas estáticas creadas en el capítulo 4 (página 49). Crea un directorio llamado `es/` dentro del directorio `app/Resources/views/sitio/` y copia en su interior todas las páginas estáticas:

```
app/
└─ Resources/
   └─ views/
      └─ sitio/
         └─ es/
            ├─ ayuda.html.twig
            ├─ contacto.html.twig
            ├─ privacidad.html.twig
            └─ sobre-nosotros.html.twig
```

Ahora crea otro directorio llamado `en/` dentro de `sitio/`, copia las plantillas de `es/` y traduce sus contenidos al inglés (pero manteniendo el nombre del archivo de cada plantilla):

```
app/
└─ Resources/
   └─ views/
      └─ sitio/
         ├─ es/
         │  ├─ ayuda.html.twig
         │  ├─ contacto.html.twig
         │  ├─ privacidad.html.twig
         │  └─ sobre-nosotros.html.twig
         │
         └─ en/
            ├─ ayuda.html.twig
            ├─ contacto.html.twig
            ├─ privacidad.html.twig
            └─ sobre-nosotros.html.twig
```

Después, modifica ligeramente el código del controlador que se encarga de mostrar las páginas estáticas:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;
```

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class SitioController extends Controller
{
    public function paginaAction($pagina, Request $request)
    {
        return $this->render(sprintf(
            'sitio/%s/%s.html.twig', $request->getLocale(), $pagina
        ));
    }
}
```

Con este cambio, cuando el usuario solicita una página estática, se obtiene el valor de su *locale* y se carga la plantilla que se encuentra en el directorio correspondiente a ese *locale*.

10.6 Traduciendo fechas

Las plantillas Twig pueden modificar el formato de las fechas antes de mostrarlas, pero no pueden traducirlas por defecto. Así que si incluyes el nombre del mes en las fechas, siempre se mostrará en inglés.

Afortunadamente, el proyecto Twig ha publicado una serie de extensiones y mejoras opcionales que incluyen la traducción completa de las fechas a cualquier idioma. En primer lugar, añade una nueva dependencia en el proyecto mediante Composer:

```
$ composer require twig/extensions
```

Después, añade la siguiente configuración en el archivo `app/config/services.yml` para activar la extensión:

```
# app/config/services.yml
services:
    # ...
    intl.twig.extension:
        class: Twig_Extensions_Extension_Intl
        tags: [{ name: 'twig.extension' }]
```

Ahora ya puedes utilizar el filtro `localizeddate` en cualquier plantilla de la aplicación. Internamente este filtro hace uso de la clase `IntlDateFormatter`, por lo que debes tener instalada y activada la extensión `intl` de PHP.

El filtro `localizeddate` permite elegir tanto el formato de la fecha como el de la hora. Si se supone por ejemplo que la fecha es el *21 de septiembre de 2016* y la hora son las *23:59:59 hora central europea*, los diferentes valores de `IntlDateFormatter` producen los siguientes resultados:

- `IntlDateFormatter::NONE`:
    - Fecha: no muestra nada.

    - Hora: no muestra nada.

- `IntlDateFormatter::SHORT`:
    - Fecha: `9/21/16` en inglés y `21/09/16` en español.
    - Hora: `11:59 PM` en inglés y `23:59` en español.
- `IntlDateFormatter::MEDIUM`: este es el valor por defecto que utiliza el filtro tanto para la fecha como para la hora.
    - Fecha: `Sep 21, 2016` en inglés y `21/09/2016` en español.
    - Hora: `11:59:59 PM` en inglés y `23:59:59` en español.
- `IntlDateFormatter::LONG`:
    - Fecha: `September 21, 2016` en inglés y `21 de septiembre de 2016` en español.
    - Hora: `11:59:59 PM GMT+02:00` en inglés y `23:59:59 GMT+02:00` en español.
- `IntlDateFormatter::FULL`:
    - Fecha: `Wednesday, September 21, 2016` en inglés y `miércoles 21 de septiembre de 2016` en español.
    - Hora: `11:59:59 PM Spain (Madrid)` en inglés y `11:59:59 p.m. España (Madrid)` en español.

El cuarto parámetro del filtro, cuyo valor por defecto es `null`, permite indicar el *locale* al que se traduce la fecha. Si no se indica ningún *locale*, el filtro utiliza el valor configurado por defecto.

Actualiza el código de todas las plantillas de la aplicación reemplazando el filtro `date()` básico por el filtro `localizeddate()`:

```
{# Antes #}
Finalizada el {{ oferta.fechaExpiracion|date }}

{# Ahora #}
Finalizada el {{ oferta.fechaExpiracion|localizeddate() }}
Finalizada el {{ oferta.fechaExpiracion|localizeddate('long') }}
Finalizada el {{ oferta.fechaExpiracion|localizeddate('medium', 'medium') }}
Finalizada el {{ oferta.fechaExpiracion|localizeddate('full', 'short') }}
Finalizada el {{ oferta.fechaExpiracion|localizeddate('none', 'long') }}
```

Si quieres formatear la fecha en un controlador, puedes hacer uso de la misma clase `IntlDateFormatter` de PHP que utiliza internamente la extensión:

```
class DefaultController extends Controller
{
    public function defaultAction()
    {
```

```
        // ...

        $formateador = \IntlDateFormatter::create(
            $this->getRequest()->getLocale(),
            \IntlDateFormatter::LONG,
            \IntlDateFormatter::NONE
        );

        $mensaje = sprintf(
            'Error: ya compraste esta misma oferta el día %s',
            $formateador->format($fechaCompra)
        );

        // ...
    }
}
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 11

Tests unitarios y funcionales

Los tests, también llamados *pruebas*, son imprescindibles para controlar la calidad del código de tu aplicación. Todos los programadores profesionales desarrollan tests para sus proyectos. Algunos incluso escriben sus tests antes que el código, lo que se conoce como *desarrollo basado en tests* o TDD (del inglés, *"test-driven development"*).

Tradicionalmente los tests se han dividido en dos tipos: unitarios y funcionales. Los tests unitarios prueban pequeñas partes del código, como por ejemplo una función o un método. Los tests funcionales prueban partes enteras de la aplicación, también llamados *escenarios*, como por ejemplo que la portada muestre una oferta activa o que el proceso de registro de usuarios funcione correctamente.

Cuando se utilizan tests, puedes añadir, modificar o eliminar partes de la aplicación con la certeza de saber que si rompes algo, te darás cuenta al instante. Si una refactorización estropea por ejemplo el formulario de registro de usuarios, cuando pases los tests se producirá un error en el test de registro de usuarios. Así sabrás rápidamente qué se ha roto y cómo arreglarlo.

El porcentaje de código de la aplicación para el que se han desarrollado tests se conoce como *code coverage*. Cuanto más alto sea este valor, más seguridad tienes de no romper nada al modificar la aplicación. El propio código fuente de Symfony dispone de miles de tests y su *code coverage* es muy elevado, siendo del 100% en sus componentes más críticos.

11.1 Primeros pasos

Symfony utiliza la librería PHPUnit (https://phpunit.de) para definir los tests, ya que se ha convertido en la herramienta estándar en el mundo PHP. De esta forma, los tests unitarios y funcionales de Symfony combinan la potencia de PHPUnit con las utilidades y facilidades proporcionadas por Symfony.

Antes de crear los primeros tests, asegúrate de tener instalada una versión reciente de PHPUnit ejecutando los siguientes comandos:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod +x phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

Ahora ya puedes ejecutar el comando global `phpunit` para ejecutar los tests:

```
$ phpunit --version

PHPUnit 5.5.0 by Sebastian Bergmann and contributors.
```

11.2 Tests unitarios

Los tests unitarios prueban que un pequeño *trozo de código* de la aplicación funciona tal y como debería hacerlo. Idealmente, los *trozos de código* son la parte más pequeña posible que se pueda probar. En la práctica suelen probarse clases enteras, a menos que sean muy complejas y haya que probar sus métodos por separado.

Por convención, cada test unitario y funcional de Symfony se define en una clase cuyo nombre acaba en `Test` y se encuentra dentro del directorio `Tests/` del *bundle*. Además, se recomienda utilizar dentro de `Tests/` la misma estructura de directorios del elemento que se quiere probar. Si se prueba por ejemplo el controlador por defecto de AppBundle, su test debería crearse en `src/AppBundle/Tests/Controller/DefaultControllerTest.php`.

Cuando se genera un *bundle*, Symfony crea automáticamente un pequeño test de ejemplo para el controlador por defecto:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/hello/Fabien');

        $this->assertTrue($crawler->filter('html:contains("Hello Fabien")')->cou
nt() > 0);
    }
}
```

Por el momento no te fijes mucho en el código del test, ya que además de que no es un test unitario, no funciona a menos que hagas cambios importantes. El primer test unitario que se va a desarrollar es el que prueba la extensión propia de Twig, que se encuentra en `src/AppBundle/Twig/Extension/CuponExtension.php`. Para ello, crea el siguiente archivo y copia su contenido:

```
// src/AppBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace AppBundle\Tests\Twig\Extension;

class TwigExtensionTest extends \PHPUnit_Framework_TestCase
{
```

```
    public function testDescuento()
    {
        $this->assertEquals(1, 1, "Probar que 1 es igual a 1");
    }
}
```

Este primer test de ejemplo simplemente prueba que `1` es igual a `1`. Para ello utiliza el método `assertEquals()` de PHPUnit, que comprueba que los dos primeros argumentos que se le pasan son iguales.

NOTA PHPUnit define un método muy similar a `assertEquals()` llamado `assertSame()`. La diferencia entre los dos es la misma que existe entre `==` y `===`. El método `assertEquals()` comprueba que dos valores sean iguales sin importar su tipo, por lo que `0` es igual a `false` e igual a una cadena vacía. Por su parte, el método `assertSame()` comprueba que tanto el valor como el tipo de los datos sean iguales, por lo que por ejemplo el número `2` no es igual que la cadena de texto `"2"`.

En este libro se utiliza `assertEquals()` porque es suficiente para el tipo de tests que se crean. No obstante, en tus aplicaciones web reales es recomendable que utilices siempre `assertSame()`.

Para ejecutar todos los tests de la aplicación y así poder probar el test que se acaba de crear, ejecuta el siguiente comando en la consola:

```
$ phpunit -c app
```

El resultado debería ser el siguiente (el tiempo y la memoria consumida varía de un ordenador a otro y de una ejecución a otra):

```
PHPUnit 5.5.0 by Sebastian Bergmann.

.

Time: 1 second, Memory: 1.25Mb

OK (1 test, 1 assertion)
```

Ahora cambia el código del test por lo que se muestra a continuación y vuelve a ejecutar los tests:

```
// Antes
$this->assertEquals(1, 1, "Probar que 1 es igual a 1");

// Ahora
$this->assertEquals(1, 2, "Probar que 1 es igual a 2");
```

Cuando ahora ejecutes los tests, se mostrará un error:

```
$ phpunit -c app

PHPUnit 5.5.0 by Sebastian Bergmann.

F

Time: 1 second, Memory: 1.50Mb

There was 1 failure:

1) AppBundle\Tests\Twig\Extension\TwigExtensionTest::testDescuento
Probar que 1 es igual a 2
Failed asserting that <integer:2> matches expected <integer:1>.

/.../src/AppBundle/Tests/Twig/Extension/CuponExtensionTest.php:10

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Cuando se produce un error, PHPUnit muestra el texto `FAILURES!` como resumen de la ejecución. Antes muestra el listado de todos los tests que han fallado, indicando para cada error la clase y método erróneos, el mensaje propio que se incluyó en el test e información adicional como el valor esperado y el valor obtenido.

Un test que no pasa satisfactoriamente es la mejor señal de que algo no funciona bien en la aplicación. Esta es la gran ventaja de los tests unitarios, que te avisan cada que vez *rompes* la aplicación. Así que una vez creados los tests de la aplicación, cada vez que completes una nueva funcionalidad, no olvides ejecutar los tests. Si has roto algo, gracias a la información de los tests erróneos, podrás arreglarlo fácilmente antes de subir la nueva funcionalidad al servidor de producción.

Aunque el código del test anterior es trivial, su estructura es la misma que la de los tests más avanzados. En primer lugar, las clases de los tests de PHPUnit siempre heredan de `PHPUnit_Framework_TestCase`. Su código puede contener tantas propiedades y métodos como quieras, pero los métodos que se ejecutan en el test siempre se llaman `testXXX()`.

A continuación se muestra el código completo del test que prueba la función `descuento()` de la extensión propia de Twig:

```php
// src/AppBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace AppBundle\Tests\Twig\Extension;

use AppBundle\Twig\Extension\CuponExtension;

class TwigExtensionTest extends \PHPUnit_Framework_TestCase
{
    public function testDescuento()
    {
        $extension = new CuponExtension();
```

```
        $this->assertEquals('-', $extension->descuento(100, null),
            'El descuento no puede ser null'
        );
        $this->assertEquals('-', $extension->descuento('a', 3),
            'El precio debe ser un número'
        );
        $this->assertEquals('-', $extension->descuento(100, 'a'),
            'El descuento debe ser un número'
        );

        $this->assertEquals('0%', $extension->descuento(10, 0),
            'Un descuento de cero euros se muestra como 0%'
        );
        $this->assertEquals('-80%', $extension->descuento(2, 8),
            'Si el precio de venta son 2 euros y el descuento sobre el precio
             original son 8 euros, el descuento es -80%'
        );
        $this->assertEquals('-33%', $extension->descuento(10, 5),
            'Si el precio de venta son 10 euros y el descuento sobre el precio
             original son 5 euros, el descuento es -33%'
        );
        $this->assertEquals('-33.33%', $extension->descuento(10, 5, 2),
            'Si el precio de venta son 10 euros y el descuento sobre el precio
             original son 5 euros, el descuento es -33.33% con dos decimales'
        );
    }
}
```

Los test unitarios se basan en comprobar que el código cumple una serie de condiciones, también llamadas *aserciones*. Además de probar su funcionamiento normal, es muy importante que el test incluya pruebas para todos los casos extremos (números negativos, valores `null`, etc.) El método más básico para establecer una condición es `assertEquals()` cuya definición es:

```
assertEquals($valorEsperado, $valorObtenido, $mensaje)
```

El primer parámetro es el valor que esperas que devuelva el método o función que estás probando. El segundo parámetro es el valor realmente obtenido al ejecutar ese método o función. El tercer parámetro es opcional y establece el mensaje que se muestra cuando los dos valores no coinciden y se produce un error en el test.

Si ahora ejecutas los tests, obtendrás el siguiente resultado:

```
$ phpunit -c app

PHPUnit 5.5.0 by Sebastian Bergmann.

.
```

```
Time: 0 seconds, Memory: 2.25Mb

OK (1 test, 7 assertions)
```

El mensaje `OK` indica que todos los tests se han ejecutado correctamente. También se indican los tests y las *aserciones* ejecutadas, que en este caso son 1 y 7 respectivamente. El número de tests no coincide con el número de clases o archivos de test sino con el número de métodos `testXXX()` incluidos en esas clases.

Después de probar la función `descuento()`, el siguiente test prueba la función `mostrarComoLista()`, que convierte los saltos de línea en elementos de una lista HTML (`<ul>` o `<ol>`). El código del test también es muy sencillo, pero no lo es tanto comprobar el contenido generado.

El problema es que se trata de una función que genera código HTML. Para probar que el contenido se genera bien, es importante tener en cuenta todos los espacios en blanco y los saltos de línea. Como puede ser difícil hacerlo dentro del código de una función PHP, es mejor guardar el contenido original y el esperado en archivos de texto y cargarlos en el test:

```php
// src/AppBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace AppBundle\Tests\Twig\Extension;

use AppBundle\Twig\Extension\CuponExtension;

class TwigExtensionTest extends \PHPUnit_Framework_TestCase
{
    // ...

    public function testMostrarComoLista()
    {
        $fixtures = __DIR__.'/fixtures/lista';
        $extension = new CuponExtension();

        $original = file_get_contents($fixtures.'/original.txt');

        $this->assertEquals(
            file_get_contents($fixtures.'/esperado-ul.txt'),
            $extension->mostrarComoLista($original)
        );

        $this->assertEquals(
            file_get_contents($fixtures.'/esperado-ol.txt'),
            $extension->mostrarComoLista($original, 'ol')
        );
    }
}
```

Los archivos auxiliares que necesitan los tests se guardan en el directorio `fixtures/` dentro del mismo directorio donde se encuentra el test. Además, se crea dentro otro directorio llamado `lista/` para separar los archivos de este test de los posibles archivos que necesiten los otros tests.

Y el contenido de los archivos es el que se muestra a continuación (presta especial atención a los espacios en blanco y los saltos de línea):

```
// src/AppBundle/Tests/Twig/Extension/fixtures/lista/original.txt
Primer elemento
Segundo elemento
Tercer elemento

// src/AppBundle/Tests/Twig/Extension/fixtures/lista/esperado-ul.txt
<ul>
  <li>Primer elemento</li>
  <li>Segundo elemento</li>
  <li>Tercer elemento</li>
</ul>

// src/AppBundle/Tests/Twig/Extension/fixtures/lista/esperado-ol.txt
<ol>
  <li>Primer elemento</li>
  <li>Segundo elemento</li>
  <li>Tercer elemento</li>
</ol>
```

11.2.1 Probando la validación de las entidades

Asegurar que la información creada por la aplicación sea correcta es crítico para su buen funcionamiento. Para ello, no basta con añadir reglas de validación a las entidades, sino que es imprescindible comprobar que todas se están cumpliendo escrupulosamente.

Los test unitarios son una buena herramienta para automatizar esta tarea tan tediosa. La clave reside en obtener el validador de Symfony, para lo cual puedes utilizar el siguiente código:

```
// src/AppBundle/Tests/Entity/OfertaTest.php
use Symfony\Component\Validator\Validation;

class OfertaTest extends \PHPUnit_Framework_TestCase
{
    public function testValidacion()
    {
        $oferta = ...

        $validador = Validation::createValidatorBuilder()
            ->enableAnnotationMapping()
            ->getValidator();

        $errores = $validador->validate($oferta);
```

```
        // ...
    }
}
```

Para que el código anterior funcione correctamente, no olvides importar la clase `Validation` con la instrucción `use` correspondiente. Como el validador es imprescindible en el test, lo mejor es inicializarlo en el método `setUp()` del test. Antes de ejecutar los tests, PHPUnit busca un método llamado `setUp()` dentro de la clase. Si existe, lo ejecuta antes que cualquier test, por lo que es el lugar ideal para inicializar y preparar cualquier elemento que necesiten los tests:

```
// src/AppBundle/Tests/Entity/OfertaTest.php
use Symfony\Component\Validator\Validation;

class OfertaTest extends \PHPUnit_Framework_TestCase
{
    private $validator;

    public function setUp()
    {
        $this->validator = Validation::createValidatorBuilder()
            ->enableAnnotationMapping()
            ->getValidator();
    }

    public function testValidacion()
    {
        $oferta = ...
        $errores = $this->validator->validate($oferta);

        // ...
    }
}
```

A continuación se muestra parte del código que comprueba la validación de la entidad `Oferta`:

```
// src/AppBundle/Tests/Entity/OfertaTest.php
namespace AppBundle\Tests\Entity;

use Symfony\Component\Validator\Validation;
use AppBundle\Entity\Oferta;

class OfertaTest extends \PHPUnit_Framework_TestCase
{
    private $validator;

    protected function setUp()
    {
        $this->validator = Validation::createValidatorBuilder()
            ->enableAnnotationMapping()
```

```
            ->getValidator();
    }

    public function testValidarSlug()
    {
        $oferta = new Oferta();

        $oferta->setNombre('Oferta de prueba');
        $slug = $oferta->getSlug();

        $this->assertEquals('oferta-de-prueba', $slug,
            'El slug se asigna automáticamente a partir del nombre'
        );
    }

    public function testValidarDescripcion()
    {
        $oferta = new Oferta();
        $oferta->setNombre('Oferta de prueba');

        $listaErrores = $this->validator->validate($oferta);
        $this->assertGreaterThan(0, $listaErrores->count(),
            'La descripción no puede dejarse en blanco'
        );

        $error = $listaErrores[0];
        $this->assertEquals('This value should not be blank.', $error->getMessag
e());
        $this->assertEquals('descripcion', $error->getPropertyPath());

        $oferta->setDescripcion('Descripción de prueba');

        $listaErrores = $this->validator->validate($oferta);
        $this->assertGreaterThan(0, $listaErrores->count(),
            'La descripción debe tener al menos 30 caracteres'
        );

        $error = $listaErrores[0];
        $this->assertRegExp("/This value is too short/", $error->getMessage());
        $this->assertEquals('descripcion', $error->getPropertyPath());
    }

    public function testValidarFechas()
    {
        $oferta = new Oferta();
        $oferta->setNombre('Oferta de prueba');
        $oferta->setDescripcion('Descripción de prueba - Descripción de prueba
- Descripción de prueba');
```

```
        $oferta->setFechaPublicacion(new \DateTime('today'));
        $oferta->setFechaExpiracion(new \DateTime('yesterday'));

        $listaErrores = $this->validator->validate($oferta);
        $this->assertGreaterThan(0, $listaErrores->count(),
            'La fecha de expiración debe ser posterior a la fecha de publicació
n'
        );

        $error = $listaErrores[0];
        $this->assertEquals('La fecha de expiración debe ser posterior a la fech
a de publicación', $error->getMessage());
        $this->assertEquals('fechaValida', $error->getPropertyPath());
    }

    public function testValidarPrecio()
    {
        $oferta = new Oferta();
        $oferta->setNombre('Oferta de prueba');
        $oferta->setDescripcion('Descripción de prueba - Descripción de prueba
- Descripción de prueba');
        $oferta->setFechaPublicacion(new \DateTime('today'));
        $oferta->setFechaExpiracion(new \DateTime('tomorrow'));
        $oferta->setUmbral(3);

        $oferta->setPrecio(-10);

        $listaErrores = $this->validator->validate($oferta);
        $this->assertGreaterThan(0, $listaErrores->count(),
            'El precio no puede ser un número negativo'
        );

        $error = $listaErrores[0];
        $this->assertRegExp("/This value should be .* or more/", $error->getMess
ageTemplate());
        $this->assertEquals('precio', $error->getPropertyPath());
    }

    // ...
}
```

Como puedes observar en el código anterior, se recomienda que cada clase incluya muchos tests pequeños. Cada test prueba una única funcionalidad independiente de las demás. El nombre de los métodos siempre empieza por `test` y continúa con una breve descripción de la funcionalidad probada. Esta descripción se escribe con la notación *"camel case"*, en el que se unen todas las palabras con su inicial en mayúscula (`testValidarFechas()`, `testValidarPrecio()`, etc.)

Cuando el objeto que se valida no cumple alguna condición, el método `validate()` del validador devuelve una clase de tipo `ConstraintViolationList` con todas las violaciones producidas. Así, para determinar si existe algún error de validación, sólo hay que comprobar que esta colección tenga más de un elemento mediante `assertGreaterThan(0, $listaErrores->count())`.

Como los tests anteriores comprueban una por una todas las validaciones de la entidad, en cada test sólo se necesitan los datos del primer error, que se obtiene mediante la instrucción `$error = $listaErrores[0]`. Por otra parte, cada error de validación dispone de los siguientes métodos:

- `getMessageTemplate()`, devuelve la plantilla utilizada para generar el mensaje de error. Puede contener variables de Twig como por ejemplo `{{ limit }}`.
- `getMessageParameters()`, devuelve un array con los parámetros que se pasan a la plantilla para generar el mensaje de error que finalmente se muestra al usuario. Ejemplo: `array('{{ limit }}' => 30)`.
- `getMessage()`, devuelve el mensaje de error completo que se mostraría al usuario.
- `getRoot()`, devuelve el objeto que se está validando, por lo que proporciona acceso a todas las propiedades del objeto original.
- `getPropertyPath()`, devuelve el nombre de la propiedad que ha producido el error de validación.
- `getInvalidValue()`, devuelve el valor de la propiedad que ha producido el error de validación.

Cuando el mensaje de error contiene partes variables, las aserciones no se pueden crear con el método `assertEquals()`. En su lugar, utiliza el método `assertRegExp()` que comprueba si el valor obtenido cumple con la expresión regular indicada como primer parámetro. Así también puedes comprobar partes significativas del mensaje de error en vez de comprobarlo entero.

Idealmente, cada test unitario debe ser independiente de los demás y estar completamente aislado de otras partes de la aplicación. Este requisito es realmente difícil de cumplir cuando la clase que se prueba hace uso de bases de datos, archivos, otros recursos externos o cuando necesita otros objetos complejos para probar su funcionalidad.

La solución a este problema son los *stubs* y los *mocks*. Un *stub* es un método falso creado para sustituir al método que realmente debería probarse. Imagina que en una prueba necesitas hacer uso de un método que crea un archivo en un servidor remoto y devuelve su contenido. Tan sólo tienes que crear un método falso con el mismo nombre dentro tu test y modificar su código para que simplemente devuelva un contenido de texto generado aleatoriamente. Aunque en este último caso no se crearían archivos en servidores remotos, la funcionalidad es la misma, por lo que la prueba es correcta y se evita todas las complicaciones innecesarias.

Igualmente, un *mock* es un objeto falso creado para sustituir al objeto que realmente debería utilizarse en una prueba. En el test que prueba la validación de un objeto de tipo `Oferta`, se necesita también un objeto de tipo `Tienda` y otro de tipo `Ciudad` para asociarlos con la oferta. Aunque debería hacerse una consulta a la base de datos para obtener los objetos `Tienda` y `Ciudad` reales, es mucho más cómodo crear objetos falsos pero correctos:

```
// src/AppBundle/Tests/Entity/OfertaTest.php

// ...
$ciudad = new Ciudad();
$ciudad->setNombre('Ciudad de Prueba');
$oferta->setCiudad($this->ciudad);

$this->assertEquals('ciudad-de-prueba', $oferta->getCiudad()->getSlug(),
    'La ciudad se guarda correctamente en la oferta'
);
```

El método `setUp()` del test es el lugar ideal para crear e inicializar los *stubs* y *mocks*:

```
// src/AppBundle/Tests/Entity/OfertaTest.php
class OfertaTest extends \PHPUnit_Framework_TestCase
{
    protected $tienda;
    protected $ciudad;

    protected function setUp()
    {
        $ciudad = new Ciudad();
        $ciudad->setNombre('Ciudad de Prueba');
        $this->ciudad = $ciudad;

        $tienda = new Tienda();
        $tienda->setNombre('Tienda de Prueba');
        $tienda->setCiudad($this->ciudad);
        $this->tienda = $tienda;
    }

    public function testValidacion()
    {
        // ...

        $oferta->setCiudad($this->ciudad);
        $this->assertEquals(
            'ciudad-de-prueba',
            $oferta->getCiudad()->getSlug(),
            'La ciudad se guarda correctamente en la oferta'
        );

        $oferta->setTienda($this->tienda);
        $this->assertEquals(
            $oferta->getCiudad()->getNombre(),
            $oferta->getTienda()->getCiudad()->getNombre(),
            'La tienda asociada a la oferta es de la misma ciudad
             en la que se vende la oferta'
        );
```

```
    }
}
```

Además del método `setUp()`, PHPUnit define un método equivalente pero contrario llamado `tearDown()`. Si una clase de tests incluye este método, PHPUnit lo ejecuta después de todos los tests, por lo que es ideal para borrar cualquier recurso creado expresamente para la prueba y que ya no se va a necesitar (archivos de prueba, registros en la base de datos, conexiones con otros servicios web, etc.)

11.3 Test funcionales

Los tests funcionales se diseñan para probar partes enteras de la aplicación, también llamados *escenarios*. El escenario que se va a probar a continuación es la generación de la portada del sitio web. Se considera que la portada es correcta si:

1. Muestra una única oferta activa, es decir, que todavía se pueda comprar.
2. Incluye al menos un enlace o botón para que los usuarios puedan registrarse.
3. Cuando la visita un usuario anónimo, se selecciona automáticamente la ciudad por defecto establecida en el archivo de configuración de la aplicación.
4. Cuando un usuario anónimo intenta comprar, se le redirige al formulario de *login*.

La portada del sitio se genera en el controlador por defecto del *bundle* `AppBundle`. Como se explicó anteriormente, cuando se genera un *bundle*, Symfony crea un test de prueba para su controlador por defecto. Así que abre el archivo `src/AppBundle/Tests/Controller/DefaultControllerTest.php` existente y sustituye su contenido por lo siguiente:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    /** @test */
    public function laPortadaSimpleRedirigeAUnaCiudad()
    {
        $client = static::createClient();
        $client->request('GET', '/');

        $this->assertEquals(302, $client->getResponse()->getStatusCode(),
            'La portada redirige a la portada de una ciudad (status 302)'
        );
    }

    // ...
}
```

Observa que el nombre del método anterior (`laPortadaSimpleRedirigeAUnaCiudad()`) no sigue la nomenclatura `testXXX()`. El motivo es que PHPUnit también permite añadir la anotación `@test` en un método para indicar que se trata de un test.

Para probar el test anterior, ejecuta como siempre el comando `phpunit -c app`, ya que esto hace que se ejecuten todos los test de la aplicación. Cuando tu aplicación crezca, ejecutar todos los test puede consumir mucho tiempo, por lo que puedes indicar a PHPUnit el nombre de un directorio para que solamente se ejecuten los test que contenga:

```
# Ejecuta sólo los test del bundle Oferta
$ phpunit -c app src/AppBundle/

# Ejecuta sólo los test del controlador del bundle Oferta
$ phpunit -c app src/AppBundle/Tests/Controller/

# Ejecuta sólo los test de la extensión de Twig
$ phpunit -c app src/AppBundle/Tests/Twig/
```

El aspecto de un test funcional es similar al de un test unitario, siendo la principal diferencia que su clase hereda de `WebTestCase` en vez de `PHPUnit_Framework_TestCase`. La mayoría de tests funcionales realizan primero una petición HTTP a una página y después analizan si la respuesta obtenida cumple con una determinada condición.

Las peticiones HTTP se realizan con un *cliente* o navegador especial creado con la instrucción:

```
$client = static::createClient();
```

Piensa en este *cliente* como un navegador que se puede manejar mediante programación, pero que puede hacer lo mismo que hace una persona con su navegador. Para realizar una petición HTTP, indica el método como primer argumento y la URL como segundo argumento. Así, para solicitar la portada del sitio, basta con indicar lo siguiente:

```
$client->request('GET', '/');
```

Además de peticiones, este cliente también puede recargar la misma página o avanzar/retroceder dentro del historial de navegación:

```
$client->reload();
$client->back();
$client->forward();
```

El navegador también incluye varios objetos con información útil sobre la última petición:

```
$request  = $client->getRequest();
$respuesta = $client->getResponse();
$historial = $client->getHistory();
$cookies   = $client->getCookieJar();
```

Comprobar que una petición ha tenido éxito es tan sencillo como comprobar que el código de estado HTTP de la respuesta devuelta por el servidor sea `200`. La respuesta que devuelve el navegador es un objeto del mismo tipo `Response` que las respuestas *normales* de Symfony, por lo que el código de estado se obtiene a través del método `getStatusCode()`:

```
$this->assertEquals(200, $client->getResponse()->getStatusCode(),
    'Status 200 en portada'
);
```

La aplicación *Cupon* no es tan sencilla, ya que la portada simple (cuya URL es `/`) redirige internamente a la portada completa, que incluye el nombre de la ciudad y el *locale* (por ejemplo, `/es/barcelona`). Así que en este caso el código de estado de la respuesta debería ser `302` (`Found`), que es el código más utilizado al realizar redirecciones:

```
$this->assertEquals(302, $client->getResponse()->getStatusCode(),
    'La portada redirige a la portada de una ciudad (status 302)'
);
```

Por defecto el navegador de los tests no sigue las redirecciones que recibe como respuesta. Para seguir la última redirección recibida, puedes utilizar el método `followRedirect()`:

```
$client->followRedirect();
```

Si quieres forzar a que el *cliente* siga todas las redirecciones, utiliza su método `followRedirects(true)`:

```
$client = static::createClient();
$client->followRedirects(true);
```

Además de crear los objetos que guardan la información sobre la petición y la respuesta, el método `request()` del *cliente* devuelve un objeto de tipo `DomCrawler`. Este objeto facilita mucho la navegación a través de los nodos DOM del código HTML de la respuesta obtenida. En otras palabras, permite extraer con mucha facilidad cualquier información del contenido de la página, lo que a su vez facilita mucho la comprobación de las condiciones del test.

Una de las condiciones establecidas anteriormente era *"La portada muestra una única oferta activa, es decir, que todavía se pueda comprar"*. Esto se puede probar fácilmente asegurando que la portada muestre un botón de tipo *Comprar* y sólo uno:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    // ...

    /** @test */
    public function laPortadaSoloMuestraUnaOfertaActiva()
```

```
    {
        $client = static::createClient();
        $client->followRedirects(true);

        $crawler = $client->request('GET', '/');

        $ofertasActivas = $crawler->filter(
            'article.oferta section.descripcion a:contains("Comprar")'
        )->count();

        $this->assertEquals(1, $ofertasActivas,
            'La portada muestra una única oferta activa que se puede comprar'
        );
    }
}
```

El método `filter()` del objeto `Crawler` devuelve la lista completa de nodos que cumplen con el selector CSS indicado como argumento. Symfony soporta todos los selectores de los estándares de CSS2 y CSS3. Además, también incluye algunos selectores propios tan útiles como por ejemplo `:contains()`, que selecciona todos los elementos cuyo contenido incluya el texto indicado.

Así que el selector `article.oferta section.descripcion a:contains("Comprar")` se interpreta como *"selecciona todos los enlaces cuyo texto contenga la palabra Comprar y que se encuentren en un elemento `<section>` cuyo atributo `class` sea `descripcion` y a su vez se encuentren en cualquier elemento `<article>` cuyo atributo `class` sea `oferta`"*.

Una vez filtrada la lista de nodos, se pueden contar las coincidencias con el método `count()`. Así que el test sólo debe asegurarse de que la cuenta sea exactamente `1`, lo que asegura que sólo hay un botón *Comprar* en la portada.

El objeto *crawler* es tan importante para las pruebas, que el *cliente*, además de devolverlo cada vez que se hace una petición, también lo almacena internamente en un objeto que se puede obtener mediante `$client->getCrawler()`.

La segunda condición que debe cumplir la portada es *"La portada incluye al menos un enlace o botón para que los usuarios puedan registrarse"*. Si suponemos que la guía de estilo del sitio web obliga a que los enlaces o botones para registrarse contengan el texto *Regístrate ya*, el test sólo debe comprobar que ese texto se encuentre en la página al menos una vez:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    // ...

    /** @test */
```

```
    public function losUsuariosPuedenRegistrarseDesdeLaPortada()
    {
        $client = static::createClient();
        $client->request('GET', '/');
        $crawler = $client->followRedirect();

        $numeroEnlacesRegistrarse = $crawler->filter(
            'html:contains("Regístrate ya")'
        )->count();

        $this->assertGreaterThan(0, $numeroEnlacesRegistrarse,
            'La portada muestra al menos un enlace o botón para registrarse'
        );
    }
}
```

Para comprobar que la página contenga un determinado texto, lo más sencillo es utilizar el selector `html:contains("texto-a-buscar")`. El texto a buscar puede contener incluso etiquetas HTML.

La tercera condición que debe cumplir la portada es mucho más interesante desde el punto de vista del test: *"Cuando un usuario anónimo visita la portada, se selecciona automáticamente la ciudad por defecto establecida en el archivo de configuración de la aplicación"*. La condición se puede probar de la siguiente manera:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    // ...

    /** @test */
    public function losUsuariosAnonimosVenLaCiudadPorDefecto()
    {
        $client = static::createClient();
        $client->followRedirects(true);

        $crawler = $client->request('GET', '/');

        $ciudadPorDefecto = $client->getContainer()->getParameter(
            'app.ciudad_por_defecto'
        );

        $ciudadPortada = $crawler->filter(
            'header nav select option[selected="selected"]'
        )->attr('value');

        $this->assertEquals($ciudadPorDefecto, $ciudadPortada,
```

```
            'Los usuarios anónimos ven seleccionada la ciudad por defecto'
        );
    }
}
```

Aunque no resulta habitual, en ocasiones los test deben hacer uso del contenedor de inyección de dependencias. Por ello, el navegador proporciona un acceso directo al contenedor a través del método `getContainer()`. Así es muy fácil obtener el valor del parámetro de configuración `app.ciudad_por_defecto`:

```
$ciudadPorDefecto = $client->getContainer()->getParameter(
    'app.ciudad_por_defecto'
);
```

Por otra parte, para obtener el elemento seleccionado en la lista desplegable de la portada, primero se accede a la lista mediante `header nav select`. Después, se obtiene la opción seleccionada buscando aquella que tenga el atributo `selected` mediante `option[selected="selected"]`. Una vez seleccionado el nodo que cumple la condición, se obtiene el contenido de su atributo `value` mediante el método `attr()` del `Crawler`, que devuelve el valor del atributo cuyo nombre se indica:

```
$ciudadPortada = $crawler->filter(
    'header nav select option[selected="selected"]'
)->attr('value');
```

La última condición que debe cumplir la portada es *"Cuando un usuario anónimo intenta comprar, se le redirige al formulario de login"*. Para intentar comprar la oferta de la portada, el usuario pulsa el botón *Comprar* que se muestra junto con los detalles de la oferta.

Aunque se puede utilizar el método `filter()` del *crawler* para buscar el botón, resulta más sencillo hacer uso del atajo `selectLink()`. Este método busca todos los enlaces de la página que contengan el texto que se pasa como primer parámetro. También busca todas las imágenes que sean *pinchables* y que contengan ese mismo texto dentro del atributo `alt`.

```
$enlacesComprar = $crawler->selectLink('Comprar');
```

A continuación, se utiliza el método `link()` para seleccionar el primer nodo de la lista y convertirlo en un objeto de tipo enlace.

```
$enlacesComprar = $crawler->selectLink('Comprar');
$primerEnlace = $enlacesComprar->link();
```

Por último, para simular el pinchazo del enlace, se emplea el método `click()` del navegador:

```
$enlacesComprar = $crawler->selectLink('Comprar');
$primerEnlace = $enlacesComprar->link();
$client->click($primerEnlace);
```

Una vez pinchado el enlace, el siguiente paso consiste en comprobar si la aplicación redirige al usuario al formulario de *login*. Para ello, utiliza el método `isRedirect()` del objeto que guarda la respuesta:

```
$this->assertTrue($client->getResponse()->isRedirect(),
    'Cuando un usuario anónimo intenta comprar, se le redirige al formulario
     de login'
);
```

Después de seguir la redirección, se muestra el formulario de *login*. Resulta fácil comprobarlo obteniendo el valor del atributo `action` del formulario de la página. Su valor debe contener obligatoriamente la cadena `/usuario/login_check`:

```
$crawler = $client->followRedirect();

$this->assertRegExp(
    '/.*\/usuario\/login_check/',
    $crawler->filter('article form')->attr('action'),
    'Después de pulsar el botón de comprar, el usuario anónimo ve el
     formulario de login'
);
```

Con todo lo anterior, el código completo del último test de la portada es el siguiente:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    // ...

    /** @test */
    public function losUsuariosAnonimosDebenLoguearseParaPoderComprar()
    {
        $client = static::createClient();
        $client->request('GET', '/');
        $crawler = $client->followRedirect();

        $comprar = $crawler->selectLink('Comprar')->link();
        $client->click($comprar);
        $crawler = $client->followRedirect();

        $this->assertRegExp(
            '/.*\/usuario\/login_check/',
            $crawler->filter('article form')->attr('action'),
            'Después de pulsar el botón de comprar, el usuario anónimo
             ve el formulario de login'
        );
```

```
    }
}
```

11.3.1 Comprobando el rendimiento

La mayoría de tests unitarios y funcionales se diseñan para asegurar que todas las características de la aplicación funcionan correctamente. No obstante, los tests también pueden comprobar que el rendimiento de la aplicación no se degrada a medida que se añaden nuevas funcionalidades.

Imagina que quieres asegurar el buen rendimiento de la portada. Para ello podrías establecer por ejemplo un tiempo máximo de generación de medio segundo y un límite de cuatro consultas a la base de datos para obtener la información requerida. Symfony facilita estas comprobaciones ya que el navegador utilizado para las pruebas tiene acceso al *profiler*, que almacena toda la información que muestra la barra de depuración web.

El *profiler* se obtiene a través del método `getProfile()` del *cliente* y la información se obtiene a partir de los diferentes *colectores* configurados en la aplicación.

En las versiones anteriores de Symfony el *profiler* estaba activado por defecto, pero a partir de Symfony 2.3 se ha desactivado para mejorar el rendimiento de los tests. Para activar el *profiler* en un determinado test, utiliza el método `enableProfiler()` del cliente:

```
$client = static::createClient();
$client->enableProfiler();
```

Realiza a continuación todas las peticiones que quieras con el cliente y accede a los datos del *profiler* a través del método `getProfile()`:

```
$profiler = $client->getProfile();

$numConsultasBD = count($profiler->getCollector('db')->getQueries());

// getDuration() devuelve el tiempo en milisegundos
$tiempoGeneracionPagina = $profiler->getCollector('time')->getDuration();
```

De esta forma, la prueba del test tendría el siguiente aspecto:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest.php
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    // ...

    /** @test */
    public function laPortadaRequierePocasConsultasDeBaseDeDatos()
    {
        $client = static::createClient();
        $client->enableProfiler();
```

```
        $client->request('GET', '/');

        if ($profiler = $client->getProfile()) {
            $this->assertLessThan(
                4,
                count($profiler->getCollector('db')->getQueries()),
                'La portada requiere menos de 4 consultas a la base de datos'
            );
        }
    }

    /** @test */
    public function laPortadaSeGeneraMuyRapido()
    {
        $client = static::createClient();
        $client->enableProfiler();
        $client->request('GET', '/');

        if ($profiler = $client->getProfile()) {
            $this->assertLessThan(
                500,
                $profiler->getCollector('time')->getDuration(),
                'La portada se genera en menos de medio segundo'
            );
        }
    }
}
```

Los *colectores* disponibles son los siguientes:

- `config`: devuelve información sobre la configuración de la aplicación: `token` (el *token* de XDebug), `symfony_version`, `name` (nombre del kernel), `env` (entorno de ejecución de la aplicación), `debug` (si está activada la depuración), `php_version`, `xdebug_enabled`, `eaccel_enabled`, `apc_enabled`, `xcache_enabled` y `bundles` (un array con el nombre y ruta de los *bundles* activados en la aplicación).
- `db`: devuelve un array con la lista de consultas a la base de datos necesarias para generar la página (`queries`), así como la lista de conexiones (`connections`) y de *entity managers* (`managers`) utilizados.
- `events`: devuelve la lista de todos los *listeners* ejecutados durante la generación de la página (`called_listeners`) y la lista de los *listeners* que no se han ejecutado (`not_called_listeners`).
- `exception`: devuelve toda la información sobre la excepción producida al generar la página (`exception`).
- `logger`: devuelve un array con todos los logs generados (`logs`) y el total de errores producidos (`error_count`).

- `memory`: devuelve el total de memoria reservada por PHP para generar la página, en bytes (`memory`).
- `request`: devuelve información tanto de la petición como de la respuesta:
  - Petición: `format`, `request_query` (parámetros enviados en la *query string*), `request_request` (el objeto de la petición), `request_headers`, `request_server` (parámetros de la variable `$_SERVER`), `request_cookies`, `request_attributes`.
  - Respuesta: `content`, `content_type`, `status_code`, `response_headers`, `session_attributes`.
- `security`: devuelve información del usuario, como su nombre (`user`), sus *roles* (`roles`), si está autenticado (`authenticated`) y si está habilitado (`enabled`).
- `swiftmailer`: devuelve la lista de emails enviados por la página (`messages`), el número total de emails (`messageCount`) y si se está usando un *spool* de mensajes (`isSpool`).
- `time`: devuelve el tiempo en milisegundos (`time`) que ha tardado la aplicación en generar la página.

11.3.2 Enviando formularios

El navegador utilizado en los tests no sólo puede pinchar enlaces, sino que también es capaz de rellenar formularios y enviarlos. De hecho, Symfony contiene varios atajos para simplificar al máximo esta tarea. En primer lugar, para obtener el formulario, se recomienda buscar su botón de envío mediante `selectButton()`:

```
$crawler->selectButton('Registrarme');
```

El método `selectButton()` busca elementos de tipo `<button>` o `<input type="submit">` cuyo atributo `value`, `id`, `name` o `alt` coincida con el parámetro que se le pasa. Si sientes curiosidad por saber cómo es capaz el *crawler* de encontrar siempre el botón adecuado, esta es la búsqueda Xpath que genera el código anterior:

```
//input[((@type="submit" or @type="button") and contains(concat(' ', normalize-s
pace(string(@value)), ' '), ' Registrarme ')) or (@type="image" and contains(con
cat(' ', normalize-space(string(@alt)), ' '), ' Registrarme ')) or @id="Registra
rme" or @name="Registrarme"] | //button[contains(concat(' ', normalize-space(str
ing(.)), ' '), ' Registrarme ') or @id="Registrarme" or @name="Registrarme"]
```

Una vez encontrado el botón de envío, el método `form()` devuelve el formulario en el que está incluido ese botón. Así, para buscar el formulario de registro, tan sólo hay que escribir lo siguiente:

```
$formulario = $crawler->selectButton('Registrarme')->form();
```

Rellenar un formulario con datos de prueba es igualmente sencillo, ya que basta con pasar un array de datos al método `form()` anterior. Finalmente, para enviarlo se emplea el método `submit()` disponible en el navegador:

```
$usuario = array(
    'nombre'    => 'Anónimo',
```

```
    'apellidos' => 'Apellido1 Apellido2'
);
$formulario = $crawler->selectButton('Registrarme')->form($usuario);

$client->submit($formulario);
```

El código anterior supone que el valor del atributo `name` de los campos del formulario es `nombre`, `apellidos`, etc. Si observas el código fuente de la página que muestra el formulario de registro, verás que en Symfony, el atributo `name` de cada campo del formulario sigue la nomenclatura `nombre-formulario[nombre-propiedad]`.

El nombre del formulario se establece con el método `getBlockPrefix()` de su clase. Para simplificar el código de los tests, es recomendable indicar un nombre corto y significativo, como por ejemplo `usuario`:

```
// src/AppBundle/Form/UsuarioType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        // ...
    }

    public function getBlockPrefix()
    {
        return 'usuario';
    }
}
```

Si utilizas el código anterior, el nombre de los campos del formulario de registro de usuarios sería:

```
usuario[nombre]
usuario[apellidos]
usuario[email]
...
```

Si el formulario permite introducir fechas, y estas se muestran como un *widget* de tres listas desplegables para elegir el día, mes y año, los campos se denominan:

```
usuario[fechaNacimiento][day]
usuario[fechaNacimiento][month]
usuario[fechaNacimiento][year]
```

Por su parte, a los campos de tipo `repeated` (como por ejemplo la contraseña) se les asigna los siguientes nombres:

```
usuario[passwordEnClaro][first]
usuario[passwordEnClaro][second]
```

11.3.3 Datos de prueba

Repetir un mismo test con varios datos de prueba es una tarea tediosa, pero muy habitual. Si el test anterior por ejemplo quiere probar el formulario de registro con cinco usuarios, el código resultante será complejo y repetitivo. Para estos casos, PHPUnit ha ideado el concepto de *data providers*.

Los *data providers* son métodos públicos disponibles en la propia clase del test y que generan datos de prueba para los diferentes tests:

```
// src/AppBundle/Tests/Twig/Extension/CuponExtensionTest.php
class DescuentoTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @dataProvider descuentos
     */
    public function testDescuento($precio, $descuento, $resultado)
    {
        $this->assertEquals($resultado, $precio - $descuento);
    }

    public function descuentos()
    {
        return array(
          array(10, 2, 8),
          array(5, 3, 2),
          array(-10, -2, -12),
          array(3, 6, -3)
        );
    }
}
```

Los métodos de tipo *data provider* siempre devuelven un array de arrays. Como el ejemplo anterior devuelve un array con cuatro arrays, el test `testDescuento()` se ejecuta cuatro veces seguidas, utilizando cada vez uno de los arrays. En cada ejecución, PHPUnit utiliza la información del array para asignar el valor de los parámetros que espera el test. Siguiendo el ejemplo anterior, en la primera ejecución `$precio = 10`, `$descuento = 2` y `$resultado = 8`.

Para utilizar los *data providers*, sólo es necesario crear un método que devuelva un array de arrays y añadir la anotación `@dataProvider` en todos los tests que quieran utilizarlo. A continuación se

muestra de forma resumida cómo podría probarse el formulario de registro con varios usuarios diferentes:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
namespace AppBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    public function generaUsuarios()
    {
        $usuario1 = array( 'nombre' => ..., 'apellidos' => ... );
        $usuario2 = array( 'nombre' => ..., 'apellidos' => ... );
        $usuario3 = array( 'nombre' => ..., 'apellidos' => ... );

        return array(
            array($usuario1),
            array($usuario2),
            array($usuario3),
        );
    }

    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $formulario = $crawler->selectButton('Registrarme')->form($usuario);
        $client->submit($formulario);

        // ...
    }
}
```

11.3.4 Creando el test para el registro de usuarios

El registro de nuevos usuarios en el sitio web es una de las partes críticas de la aplicación. Su funcionamiento debe ser siempre correcto, por lo que resulta necesario crear un test funcional para este escenario.

Además de probar el formulario de registro, el test que se va a desarrollar comprueba que el usuario ha sido realmente dado de alta en la base de datos y que se ha creado una sesión para el usuario recién *logueado*.

Comienza creando la estructura básica del test y, al menos, un usuario de prueba:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
namespace AppBundle\Tests\Controller;
```

```
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...
    }

    public function generaUsuarios()
    {
        return array(
            array(
                array(
                    'usuario[nombre]'    => 'Anónimo',
                    'usuario[apellidos]' => 'Apellido1 Apellido2',
                    'usuario[email]'     =>
                        'anonimo'.uniqid().'@localhost.localdomain',
                    'usuario[passwordEnClaro][first]'  => 'anonimo1234',
                    'usuario[passwordEnClaro][second]' => 'anonimo1234',
                    'usuario[direccion]'        => 'Calle ...',
                    'usuario[dni]'              => '11111111H',
                    'usuario[numero_tarjeta]'   => '123456789012345',
                    'usuario[ciudad]'           => '1',
                    'usuario[permiteEmail]'    => '1'
                )
            )
        );
    }
}
```

Como el usuario de prueba realmente se va a registrar en el sitio web, su email no puede coincidir con el de ningún otro usuario registrado. Para ello se genera una dirección de correo electrónico aleatoria mediante:

```
'anonimo'.uniqid().'@localhost.localdomain'
```

El resto de datos del usuario serán siempre los mismos en todas las ejecuciones del test. Lo primero que debe hacer el test es cargar la portada, pinchar el enlace *Regístrate ya* y comprobar que se carga la página con el formulario de registro:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
```

```
     */
    public function testRegistro($usuario)
    {
        $client = static::createClient();
        $client->followRedirects(true);

        $crawler = $client->request('GET', '/');

        $enlaceRegistro = $crawler->selectLink('Regístrate ya')->link();
        $crawler = $client->click($enlaceRegistro);

        $this->assertGreaterThan(
            0,
            $crawler->filter(
                'html:contains("Regístrate gratis como usuario")'
            )->count(),
            'Después de pulsar el botón Regístrate de la portada, se carga la
             página con el formulario de registro'
        );
    }

    // ...
}
```

Empleando los métodos `selectLink()`, `link()` y `click()` es muy fácil buscar y pinchar el enlace cuyo contenido es *Regístrate ya*. Después, una vez pinchado el enlace, se comprueba que la página cargada contenga el texto *"Regístrate gratis como usuario"*, que es lo que se escribió en la plantilla.

El siguiente paso consiste en registrarse en el sitio web utilizando el formulario:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $formulario = $crawler->selectButton('Registrarme')->form($usuario);
        $crawler = $client->submit($formulario);

        $this->assertTrue($client->getResponse()->isSuccessful());

        $this->assertRegExp(
            '/(\d|[a-z])+/',
            $client->getCookieJar()->get('MOCKSESSID', '/', 'localhost')->getVal
ue(),
```

```
            'La aplicación ha enviado una cookie de sesión'
        );
    }

    // ...
}
```

Utilizando los métodos `selectButton()` y `form()` explicados anteriormente, junto con los datos de prueba que genera el *data provider*, resulta muy sencillo rellenar el formulario de registro:

```
$formulario = $crawler->selectButton('Registrarme')->form($usuario);
```

Una vez enviado el formulario con el método `submit()`, se comprueba que la respuesta devuelta por el servidor sea correcta. Para ello puedes comprobar el código de estado de la respuesta o puedes utilizar como atajo el método `isSuccessful()`:

```
$this->assertTrue($client->getResponse()->isSuccessful());
```

El método `assertTrue()` de PHPUnit comprueba que el parámetro que se le pasa sea el valor *booleano* `true`. El método `isSuccessful()` devuelve `true` si el código de estado es mayor o igual que `200` y menor que `300`.

Cuando un usuario se registra con éxito en el sitio, además de ser redirigido a la portada, la aplicación le *loguea* automáticamente. Por tanto, si el registro ha sido correcto, el navegador tendrá ahora una *cookie* de sesión.

Las *cookies* del navegador se obtienen a través del método `getCookieJar()`. Para obtener los datos de una *cookie* específica, se utiliza el método `get()` pasándole como parámetro el nombre de la *cookie*. Como en los tests por defecto la *cookie* de sesión se llama `MOCKSESSID`, el código para obtener su valor es el siguiente:

```
$cookie = $client->getCookieJar()->get('MOCKSESSID', '/', 'localhost');
$contenidoCookie = $cookie->getValue();
```

Utilizando una expresión regular es posible comprobar que el navegador tiene una *cookie* de sesión válida:

```
$this->assertRegExp(
    '/(\d|[a-z])+/',
    $client->getCookieJar()->get('MOCKSESSID', '/', 'localhost')->getValue(),
    'La aplicación ha enviado una cookie de sesión'
);
```

Como el método `get()` devuelve `null` cuando la *cookie* no existe, el código anterior se puede simplificar por lo siguiente:

```
$this->assertTrue(
    $client->getCookieJar()->get('MOCKSESSID', '/', 'localhost'),
    'La aplicación ha enviado una cookie de sesión'
);
```

La clase `CookieJar`también incluye los métodos `clear()` para borrar todas las *cookies* y `set()` para añadir una nueva *cookie* al navegador.

La existencia de la *cookie* de sesión es una prueba suficiente de que el registro ha sido correcto. Aún así, si quieres comprobarlo de manera irrefutable, puedes hacer que el navegador pinche en el enlace *Ver mi perfil* que se muestra en la zona lateral de las páginas de los usuarios *logueados*:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $perfil = $crawler->filter('aside section#login')->selectLink(
            'Ver mi perfil'
        )->link();

        $crawler = $client->click($perfil);
    }

    // ...
}
```

Una vez cargada la página del perfil, comprueba que la dirección de email que se muestra es la misma que la que se utilizó al registrarse. Como esta dirección se genera aleatoriamente al ejecutar el test, si los dos valores coinciden es completamente seguro que el registro funciona bien:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $perfil = $crawler->filter('aside section#login')->selectLink(
            'Ver mi perfil'
        )->link();

        $crawler = $client->click($perfil);

        $this->assertEquals(
            $usuario['usuario[email]'],
```

```
            $crawler->filter(
                'form input[name="usuario[email]"]'
            )->attr('value'),
            'El usuario se ha registrado correctamente y sus datos se han
             guardado en la base de datos'
        );
    }

    // ...
}
```

Por último, para dejar la base de datos tal y como se encontraba antes de ejecutar los tests, no olvides borrar el usuario aleatorio creado anteriormente:

```
// src/AppBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $usuario = $this->em->getRepository('AppBundle:Usuario')
            ->findOneByEmail($usuario['usuario[email]']);

        $this->em->remove($usuario);
        $this->em->flush();
    }

    // ...
}
```

Otra posible solución para evitar modificar la base de datos cada vez que se ejecutan los tests consiste en utilizar otra base de datos de pruebas para los tests. Para ello sólo tienes que crear una nueva base de datos y configurar su acceso en el archivo `app/config/config_test.yml`, que es el archivo de configuración utilizado en los tests.

11.4 Configurando PHPUnit en Symfony

Al ejecutar PHPUnit en todos los ejemplos anteriores, se ha utilizado el comando `phpunit -c app`. La opción `-c` indica el nombre del directorio en el que se encuentra el archivo de configuración de PHPUnit. Si al ejecutar las pruebas lo haces desde el directorio `app/` del proyecto Symfony, puedes ejecutar simplemente el comando `phpunit`.

En cualquier caso, PHPUnit busca un archivo llamado `phpunit.xml` en el directorio actual o en el directorio indicado mediante `-c`. Si no lo encuentra, busca un archivo llamado `phpunit.xml.dist`.

Symfony ya incluye un archivo de configuración adecuado en `app/phpunit.xml.dist` con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- https://phpunit.de/manual/current/en/appendixes.configuration.html -->
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.8/phpunit.xs
d"
         backupGlobals="false"
         colors="true"
         bootstrap="autoload.php"
>
    <php>
        <ini name="error_reporting" value="-1" />
        <!--
            <server name="KERNEL_DIR" value="/path/to/your/app/" />
        -->
    </php>

    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/*/Bundle/*Bundle/Tests</directory>
            <directory>../src/*Bundle/Tests</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
            <directory>../src</directory>
            <exclude>
                <directory>../src/*Bundle/Resources</directory>
                <directory>../src/*Bundle/Tests</directory>
                <directory>../src/*/*Bundle/Resources</directory>
                <directory>../src/*/*Bundle/Tests</directory>
                <directory>../src/*/Bundle/*Bundle/Resources</directory>
                <directory>../src/*/Bundle/*Bundle/Tests</directory>
            </exclude>
        </whitelist>
    </filter>
</phpunit>
```

La explicación detallada de todas las opciones de configuración se encuentra en el manual de PHPUnit, pero básicamente la configuración anterior indica que se deben ejecutar todos los test que se encuentren en el directorio `Tests/` de cualquier *bundle* de la aplicación. La sección `<filter>` indica las partes de la aplicación que no se deben tener en cuenta para el *code coverage*.

Una buena práctica recomendada consiste en copiar el archivo `phpunit.xml.dist` de Symfony y renombrarlo a `phpunit.xml` para configurar PHPUnit según tus necesidades. Si utilizas un repositorio común de código tipo Git o Subversion, no olvides excluir este nuevo archivo para no interferir en los tests de los demás programadores del proyecto.

El siguiente ejemplo muestra cómo restringir las pruebas que se ejecutan un único *bundle* llamado `AppBundle`:

```
<!-- app/phpunit.xml -->
...

<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/AppBundle/Tests</directory>
    </testsuite>
</testsuites>
```

11.4.1 Configurando el informe de resultados

Los resultados que muestra PHPUnit en la consola de comandos resultan en ocasiones demasiado concisos:

```
$ phpunit -c app
PHPUnit 5.5.0 by Sebastian Bergmann.

F....

Time: 3 seconds, Memory: 21.50Mb
```

Por cada test ejecutado, se muestra un punto si el test ha sido exitoso, una letra `F` cuando falla alguna *aserción*, una letra `E` cuando se produce algún error o excepción de PHP, una letra `S` cuando se ha saltado el test y una letra `I` cuando el test está marcado como incompleto.

Para obtener resultados más detallados, añade la opción `--tap`:

```
$ phpunit -c app --tap

TAP version 13
ok 1 - AppBundle\Tests\Controller\DefaultControllerTest::testPortada
ok 2 - AppBundle\Tests\Entity\OfertaTest::testValidacion
ok 3 - AppBundle\Tests\Twig\Extension\TwigExtensionTest::testDescuento
ok 4 - AppBundle\Tests\Twig\Extension\TwigExtensionTest::testMostrarComoLista
ok 5 - AppBundle\Tests\Controller\DefaultControllerTest::testRegistro with data
set #0 (array('Anónimo', 'Apellido1 Apellido2', 'anonimo4e672de965aff@localhos
t.localdomain', 'anonimo1234', 'anonimo1234', 'Mi calle, Mi ciudad, 01001', '0
1', '01', '1970', '11111111H', '123456789012345', '1', '1'))
1..5
```

También puedes hacer uso de la opción `--testdox`:

```
$ phpunit -c app --testdox

PHPUnit 5.5.0 by Sebastian Bergmann.

AppBundle\Tests\Controller\DefaultController
 [x] Portada

AppBundle\Tests\Entity\Oferta
 [x] Validacion

AppBundle\Tests\Twig\Extension\TwigExtension
 [x] Descuento
 [x] Mostrar como lista

AppBundle\Tests\Controller\DefaultController
 [x] Registro
```

PHPUnit es una herramienta tan completa y dispone de tantas utilidades que si no lo has hecho ya, es muy recomendable repasar su documentación, disponible en https://phpunit.de/manual/current/en/index.html

Sección 3

Extranet

Esta página se ha dejado vacía a propósito

CAPÍTULO 12

Planificación

Las tiendas de la aplicación disponen de una zona privada para gestionar sus datos y añadir nuevas ofertas. Esta zona privada, denominada *extranet*, es completamente independiente de la parte pública (o *frontend*) y de la parte de administración de la aplicación (o *backend*).

Cada tienda accede a su *extranet* mediante un nombre de usuario y una contraseña. Una vez dentro, la portada le muestra un listado de todas sus ofertas. Si la oferta ya ha sido publicada o ha sido revisada por los administradores del sitio web, no se pueden modificar sus datos. Si la oferta ha generado alguna venta, se puede ver el listado detallado de los compradores.

Por último, en la *extranet* se incluye un formulario para añadir nuevas ofertas y otro formulario para ver y/o modificar los datos de la tienda.

12.1 Bundles

El concepto de *bundle* en Symfony es tan flexible, que la *extranet* se puede crear de varias maneras:

1. Crear un *bundle* específico llamado `ExtranetBundle` que englobe todos los elementos relacionados con la *extranet*.
2. Incluir todos los elementos de la *extranet* dentro del *bundle* `AppBundle`, ya que la *extranet* no deja de ser la parte de administración de las tiendas.
3. Incluir los elementos de la *extranet* dentro de un gran *bundle* llamado `BackendBundle` que comprenda toda la parte de administración, tanto el *backend* como la *extranet*.

En este libro se ha optado por la segunda propuesta. El primer motivo es que la *extranet* y el *backend* son conceptos relacionados, pero demasiado diferentes como para mezclarlos en el mismo *bundle*. El segundo motivo es que la *extranet* es tan sencilla que no es necesario crear un *bundle* para guardar los poquísimos archivos necesarios.

12.1.1 Controlador, plantillas y rutas

Para definir la *extranet* completa son necesarios los siguientes elementos:

- Crear un nuevo controlador `ExtranetController` dentro del directorio `Controller/` del *bundle* `AppBundle`.
- Crear un nuevo directorio llamado `extranet/` dentro del directorio `app/Resources/views/` para definir las plantillas
- Crear en el directorio `app/Resources/views/` un nuevo layout llamado `extranet.html.twig` del que hereden todas las plantillas de la *extranet*.

- Crear un repositorio propio para agrupar las nuevas consultas relacionadas con la entidad `Tienda`.

12.2 Enrutamiento

La funcionalidad completa de la *extranet* se puede realizar con las siguientes cinco rutas:

- `extranet_portada`, muestra la portada de la *extranet* de la tienda.
- `extranet_oferta_nueva`, muestra el formulario para crear una nueva oferta.
- `extranet_oferta_editar`, muestra el formulario para modificar los datos de una oferta.
- `extranet_oferta_ventas`, muestra un listado detallado con las ventas de una oferta.
- `extranet_perfil`, muestra un formulario con la información de la tienda y permite modificar cualquier dato.

Con todo esto, ya puedes incluir las cicno acciones vacías dentro del nuevo controlador `ExtranetController` y añadir las correspondientes anotaciones para las rutas:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * @Route("/extranet")
 */
class ExtranetController extends Controller
{
    /**
     * @Route("/", name="extranet_portada")
     */
    public function portadaAction() { }

    /**
     * @Route("/oferta/ventas/{id}", name="extranet_oferta_ventas")
     */
    public function ofertaVentasAction() { }

    /**
     * @Route("/oferta/nueva", name="extranet_oferta_nueva")
     */
    public function ofertaNuevaAction() { }

    /**
     * @Route("/oferta/editar/{id}", name="extranet_oferta_editar")
     */
    public function ofertaEditarAction() { }
```

```
    /**
     * @Route("/perfil", name="extranet_perfil")
     */
    public function perfilAction() { }
}
```

Observa cómo el patrón de las rutas individuales no incluye el valor `/extranet`. Cuando todas las rutas comienzan de la misma manera, es mejor añadir la anotación `@Route()` en la clase del controlador para definir ese prefijo común.

12.3 Layout

Si recuerdas la sección *[herencia de plantillas a tres niveles](#herencia-a -tres-niveles)* del capítulo 7, todas las páginas de la *extranet* heredan de una plantilla llamada `extranet.html.twig`, que a su vez hereda de la plantilla `base.html.twig`.

Crea el archivo `app/Resources/views/extranet.html.twig` y copia en su interior el siguiente código Twig:

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

{% block stylesheets %}
    <link href="{{ asset('css/normalizar.css') }}" />
    <link href="{{ asset('css/comun.css') }}" />
    <link href="{{ asset('css/extranet.css') }}" />
{% endblock %}

{% block body %}
<header>
    <h1><a href="{{ path('extranet_portada') }}">CUPON EXTRANET</a></h1>

    <nav>
        <ul>
            <li><a href="{{ path('extranet_portada') }}">Ofertas</a></li>
            <li><a href="{{ path('extranet_perfil') }}">Mis datos</a></li>
            <li><a href="#">Cerrar sesión</a></li>
        </ul>
    </nav>

    <p>Teléfono de atención al cliente <strong>902 XXX XXX</strong></p>
</header>

<article>
    {% block article %}{% endblock %}
</article>

<aside>
    {% block aside %}
```

```
    <a class="boton" href="{{ path('extranet_oferta_nueva') }}">Añadir ofert
a</a>

    <section id="faq">
        <h2>Preguntas frecuentes</h2>

        <dl>
            <dt>¿Lorem ipsum dolor sit amet?</dt>
            <dd>Consectetur adipisicing elit, sed do eiusmod tempor.</dd>
            <dt>¿Ut enim ad minim veniam?</dt>
            <dd>Quis nostrud exercitation ullamco laboris nisi.</dd>
            <dt>¿Excepteur sint occaecat cupidatat non proident?</dt>
            <dd>Sunt in culpa qui officia deserunt mollit anim laborum.</dd>
        </dl>
    </section>
    {% endblock %}
</aside>
{% endblock %}
```

La primera instrucción de la plantilla indica que hereda de la plantilla `base.html.twig` que se encuentra directamente en `app/Resources/views/`:

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

Una vez declarada la herencia, la plantilla `extranet.html.twig` ya no puede incluir contenidos propios, sino que solamente puede rellenar los bloques definidos por la plantilla base. El primer bloque importante se llama `stylesheets` y define las hojas de estilo CSS que se enlazan en todas las páginas de la *extranet*:

```
{% block stylesheets %}
    <link href="{{ asset('css/normalizar.css') }}" />
    <link href="{{ asset('css/comun.css') }}" />
    <link href="{{ asset('css/extranet.css') }}" />
{% endblock %}
```

Además de las hojas de estilo comunes `normalizar.css` y `comun.css`, las páginas de la *extranet* definen sus propios estilos en un archivo llamado `extranet.css` que debes crear en el directorio `web/css/`.

NOTA Si estás desarrollando la aplicación a medida que lees el libro, puedes copiar los contenidos de la hoja de estilos `extranet.css` que se encuentra en: https://github.com/javiereguiluz/Cupon/blob/2.8/web/css/extranet.css

El siguiente bloque definido en la plantilla es `body`, que incluye todos los contenidos que se muestran en la página. El primer elemento de este bloque es la cabecera de la página (`<header>`), mucho más simple que la del *frontend*:

```
{% block body %}
<header>
    <h1><a href="{{ path('extranet_portada') }}">CUPON EXTRANET</a></h1>

    <nav>
        <ul>
            <li><a href="{{ path('extranet_portada') }}">Ofertas</a></li>
            <li><a href="{{ path('extranet_perfil') }}">Mis datos</a></li>
            <li><a href="#">Cerrar sesión</a></li>
        </ul>
    </nav>

    <p>Teléfono de atención al cliente <strong>902 XXX XXX</strong></p>
</header>

{# ... #}
{% endblock %}
```

El enlace *Cerrar sesión* no se define por el momento porque hasta que no se configure la seguridad de la *extranet* todavía no existe una ruta para desconectar al usuario. Todas las páginas de la *extranet* se estructuran en dos columnas de contenidos. Así que la plantilla `extranet.html.twig` define dos nuevos bloques llamados `article` y `aside` para los contenidos principales y secundarios de la página:

```
{% block body %}
{# ... #}

<article>
    {% block article %}{% endblock %}
</article>

<aside>
    {% block aside %}
    <a class="boton" href="{{ path('extranet_oferta_nueva') }}">Añadir ofert
a</a>

    <section id="faq">
        <h2>Preguntas frecuentes</h2>

        <dl>
            <dt>¿Lorem ipsum dolor sit amet?</dt>
            <dd>Consectetur adipisicing elit, sed do eiusmod tempor.</dd>
            <dt>¿Ut enim ad minim veniam?</dt>
            <dd>Quis nostrud exercitation ullamco laboris nisi.</dd>
            <dt>¿Excepteur sint occaecat cupidatat non proident?</dt>
            <dd>Sunt in culpa qui officia deserunt mollit anim laborum.</dd>
        </dl>
    </section>
```

```
    {% endblock %}
</aside>
{% endblock %}
```

El bloque `article` no define ningún contenido por defecto porque cada página incluirá contenidos muy diferentes. El bloque `aside` muestra por defecto el botón para añadir nuevas ofertas y un listado de preguntas frecuentes, ya que se considera que estos contenidos se repiten en varias páginas de la *extranet*.

CAPÍTULO 13

Seguridad

La *extranet* de la aplicación se encuentra bajo la ruta `/extranet/*`. Cualquier intento de acceso a una página de la *extranet* redirige al usuario al formulario de login, que es la única página de la *extranet* que puede ser vista por usuarios anónimos.

Además, como solamente las tiendas pueden acceder a la *extranet*, es necesario crear un nuevo tipo de usuario. Siguiendo la recomendación de crear *roles* con nombres auto-descriptivos, se define un nuevo *role* llamado `ROLE_TIENDA`.

Por otra parte, la aplicación debe comprobar que cada tienda sólo pueda modificar sus propias ofertas. Para asegurar que siempre se cumpla esta condición, se va a crear un *security voter* propio.

13.1 Definiendo la nueva configuración de seguridad

La seguridad de las aplicaciones Symfony se configura en el archivo `app/config/security.yml`. Después de los cambios realizados en los capítulos anteriores, su contenido actual es el siguiente:

```
# app/config/security.yml
security:
    firewalls:
        frontend:
            pattern:        ^/*
            provider:       usuarios
            anonymous:      ~
            form_login:
                login_path: usuario_login
                check_path: usuario_login_check
            logout:
                path:       usuario_logout
            remember_me:
                lifetime:   604800    # 7 * 24 * 3600 = 604.800 = 1 semana

    access_control:
        - { path: ^/(es|en)/usuario/login,
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(es|en)/usuario/registro,
            roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(es|en)/usuario/*, roles: ROLE_USUARIO }

    providers:
        usuarios:
            entity: { class: AppBundle\Entity\Usuario, property: email }
```

```
    encoders:
        AppBundle\Entity\Usuario: bcrypt
```

Para cumplir los requerimientos de la *extranet* es necesario definir un nuevo *firewall*. Como el orden de los *firewalls* es importante y el *frontend* abarca todas las URL de la aplicación, para que el nuevo *firewall* llamado `extranet` tenga efecto, tienes que definirlo antes que el *firewall* `frontend`:

```
# app/config/security.yml

security:
    firewalls:
        extranet:
            pattern:        ^/extranet
            provider:       tiendas
            anonymous:      ~
            form_login:
                login_path: /extranet/login
                check_path: /extranet/login_check
            logout:
                path:       extranet_logout

        frontend:
            pattern:        ^/*
            provider:       usuarios
            # ...

    # ...
```

Como todas las URL de la *extranet* incluyen el prefijo `/extranet`, la opción `pattern` simplemente es `^/extranet`. A continuación se indica que los usuarios de este *firewall* se obtienen del proveedor `tiendas`, que se definirá a continuación.

La opción `anonymous` permite que los usuarios anónimos puedan acceder a una o más de sus URL (en función de la configuración que después se realice en la opción `access_control`). Por último, se incluyen las opciones `form_login` y `logout` para solicitar el usuario y contraseña mediante un formulario de login. El valor de estas opciones coincide con el patrón de las rutas `login` y `logout` de la *extranet*, que se definirán posteriormente.

A continuación, añade las reglas de control de acceso del nuevo *firewall*:

```
# app/config/security.yml
security:
    firewalls:
        extranet:
            pattern:        ^/extranet
            # ...

    access_control:
```

```
        # ...
        - { path: ^/extranet/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/extranet/*, roles: ROLE_TIENDA }

    # ...
```

Las reglas definidas en la opción `access_control` son muy sencillas porque todas las URL de la *extranet* exigen ser un usuario con el *role* `ROLE_TIENDA`, salvo la página del formulario de login, que puede ser accedida por cualquier usuario. El orden en el que se incluyen las reglas de control de acceso también es importante. No obstante, en este caso las reglas de la *extranet* no colisionan con las del *frontend*, por lo que puedes definirlas en cualquier orden.

El *firewall* `extranet` obtiene los usuarios de un proveedor llamado `tiendas`. Defínelo añadiendo sus opciones bajo la clave `providers`:

```
# app/config/security.yml
security:
    firewalls:
        extranet:
            pattern:        ^/extranet
            provider:       tiendas
            # ...

    access_control:
        # ...
        - { path: ^/extranet/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/extranet/*, roles: ROLE_TIENDA }

    providers:
        usuarios:
            entity: { class: AppBundle\Entity\Usuario, property: email }
        tiendas:
            entity: { class: AppBundle\Entity\Tienda, property: login }

    # ...
```

El proveedor `tiendas` crea los usuarios a partir de la entidad `Tienda` del *bundle* `AppBundle` y utiliza su propiedad `login` como nombre de usuario. Más adelante se añaden los cambios necesarios para que la entidad `Tienda` pueda crear usuarios de tipo `ROLE_TIENDA`.

Por último, define en la clave `encoders` que las contraseñas de los usuarios relacionados con las tiendas se codifican con el algoritmo `bcrypt`:

```
# app/config/security.yml
security:
    firewalls:
        extranet:
            pattern:        ^/extranet
            provider:       tiendas
```

```
            # ...

    access_control:
        # ...
        - { path: ^/extranet/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/extranet/*, roles: ROLE_TIENDA }

    providers:
        # ...
        tiendas:
            entity: { class: AppBundle\Entity\Tienda, property: login }

    encoders:
        AppBundle\Entity\Usuario: bcrypt
        AppBundle\Entity\Tienda:  bcrypt
```

13.2 Preparando el proveedor de usuarios de las tiendas

Si recuerdas la sección *[Creando proveedores de usuarios](#proveedores-de- usuarios)* del capítulo 8, convertir una entidad de Doctrine en un proveedor de usuarios es tan sencillo como implementar la inferfaz `UserInterface`.

Abre el archivo de la entidad `Tienda` del *bundle* `AppBundle` y añade los siguientes cambios:

```
// src/AppBundle/Entity/Tienda.php
use Symfony\Component\Security\Core\User\UserInterface;

class Tienda implements UserInterface
{
    public function eraseCredentials()
    {
    }

    public function getSalt()
    {
        return null;
    }

    public function getRoles()
    {
        return array('ROLE_TIENDA');
    }

    public function getUsername()
    {
        return $this->getLogin();
    }

    // El método getPassword() ya existía en la entidad
```

```
    // ...
}
```

El método `getRoles()` devuelve un array con el *role* `ROLE_TIENDA` porque todas las tiendas son del mismo tipo. El método `getUsername()` devuelve el valor de la propiedad `login`, que es la que se utiliza como nombre de usuario. Por último, no se añaden el método `getPassword()` porque ya existía, ya que la entidad `Tienda` dispone de las propiedad `password`.

13.2.1 Actualizando los usuarios de prueba

Una vez definida la nueva configuración de seguridad de la *extranet* y después de actualizar la entidad `Tienda`, actualiza el archivo de *fixtures* que crea usuarios de prueba de tipo `Tienda`.

El principal cambio es que ahora debes codificar la contraseña con el mismo algoritmo y condiciones que las que se definen en el archivo `security.yml`. Para ello es necesario obtener primero el contenedor de inyección de dependencias:

```
// src/AppBundle/DataFixtures/ORM/Tiendas.php
namespace AppBundle\DataFixtures\ORM;

// ...
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class Tiendas extends AbstractFixture implements OrderedFixtureInterface, Contai
nerAwareInterface
{
    private $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }

    public function load(ObjectManager $manager)
    {
        // ...

        foreach ($ciudades as $ciudad) {
            for ($j=1; $j<=rand(2, 5); $j++) {
                $tienda = new Tienda();

                // ...

                $tienda->setLogin('tienda'.$i);

                $encoder = $this->container->get('security.encoder_factory')
                                           ->getEncoder($tienda);
```

```
                $passwordEnClaro = 'tienda'.$i;
                $passwordCodificado = $encoder->encodePassword(
                    $passwordEnClaro,
                    $tienda->getSalt()
                );
                $tienda->setPassword($passwordCodificado);

                // ...

                $manager->persist($tienda);
            }
        }

        $manager->flush();
    }

    // ...
}
```

Después de actualizar el archivo de *fixtures*, vuelve a cargarlos en la base de datos con el comando:

```
$ php app/console doctrine:fixtures:load
```

13.3 Creando el formulario de login

Cuando un usuario trate de acceder a alguna página de la *extranet*, la configuración del control de acceso hará que el *firewall* `extranet` le solicite que se identifique mediante un formulario de login.

Como ya se explicó detalladamente en el capítulo 8, Symfony se encarga de gestionar la lógica que comprueba el usuario y contraseña introducidos (`login_check`) y la desconexión del usuario (`logout`). Así que sólo es necesario crear el código de la acción de login y su formulario asociado.

Añade en primer lugar las tres rutas y acciones relacionadas con el login dentro del controlador `ExtranetController`:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;
use use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExtranetController extends Controller
{
    /**
     * @Route("/login", name="extranet_login")
     */
    public function loginAction()
    {
        $authUtils = $this->get('security.authentication_utils');
```

```
        return $this->render('extranet/login.html.twig', array(
            'last_username' => $authUtils->getLastUsername(),
            'error' => $authUtils->getLastAuthenticationError(),
        ));
    }

    /**
     * @Route("/login_check", name="extranet_login_check")
     */
    public function loginCheckAction()
    {
        // el "login check" lo hace Symfony automáticamente, por lo que
        // no hay que añadir ningún código en este método
    }

    /**
     * @Route("/logout", name="extranet_logout")
     */
    public function logoutAction()
    {
        // el logout lo hace Symfony automáticamente, por lo que
        // no hay que añadir ningún código en este método
    }

    // ...
}
```

Después, crea el formulario de login en la plantilla `extranet/login.html.twig`:

```
{# app/Resources/views/extranet/login.html.twig #}
{% extends 'frontend.html.twig' %}

{% block id 'login' %}
{% block title %}Administra tu tienda{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('extranet_login_check') }}" method="post">
    <div>
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="_username" />
    </div>

    <div>
```

```
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="_password" />
    </div>

    <input class="boton" type="submit" name="login" value="Entrar" />
</form>
{% endblock %}

{% block aside %}
{% endblock %}
```

Como cualquier usuario puede acceder a la página del formulario de login, este hereda de `frontend.html.twig` en vez de `extranet.html.twig` para no revelar públicamente ningún tipo de información interna de la *extranet*.

Si ahora tratas de acceder a la URL `http://127.0.0.1:8000/app_dev.php/extranet` la aplicación te redirige al formulario de login de la *extranet*. Si introduces las credenciales de cualquier tienda de prueba, puedes acceder a la *extranet* pero se muestra un error porque todavía no se ha creado ninguna página.

13.3.1 Refactorizando el evento asociado al *login*

La sección *Ejecutando código después del login* (página 200) del capítulo 8 utiliza el evento `security.interactive_login` para redirigir a cada usuario a la portada de su ciudad después del login:

```
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\Routing\Router;
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    private $router, $ciudad = null;

    public function __construct(Router $router)
    {
        $this->router = $router;
    }

    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
        $token = $event->getAuthenticationToken();
        $this->ciudad = $token->getUser()->getCiudad()->getSlug();
    }
```

```
    public function onKernelResponse(FilterResponseEvent $event)
    {
        if (null === $this->ciudad) {
            return;
        }

        $urlPortada = $this->router->generate('portada', array(
            'ciudad' => $this->ciudad
        ));
        $event->setResponse(new RedirectResponse($urlPortada));
    }
}
```

El problema del código anterior es que se aplica a todos los usuarios de la aplicación, sin importar del tipo que sean. Cuando una tienda haga login en la *extranet*, el código anterior le redirigirá a la portada del *frontend* correspondiente a la ciudad a la que pertenece la tienda.

Por tanto, es necesario refactorizar este código para tener en cuenta el tipo de usuario. Si es un usuario con el *rol* `ROLE_USUARIO`, la lógica se mantiene. Si es un usuario de tipo `ROLE_TIENDA`, se le redirige a la portada de la *extranet*.

La comprobación del tipo de usuario se realiza mediante el método `isGranted()` del componente de seguridad de Symfony:

```
if ($this->get('security.authorization_checker')->isGranted('ROLE_TIENDA')) {
    // El usuario es de tipo Tienda
}
elseif ($this->get('security.authorization_checker')->isGranted('ROLE_USUARI
O')) {
    // El usuario es de tipo Usuario
}
```

Como el método `isGranted()` requiere acceder al contexto de seguridad, primero debes modificar la definición del servicio `app.login_listener` del *bundle* `AppBundle` para inyectar el servicio `@security.authorization_checker` como argumento:

```
# src/AppBundle/Resources/config/services.yml
services:
    app.login_listener:
        class:     AppBundle\Listener\LoginListener
        arguments: ['@security.authorization_checker', '@router']
        tags:
            - { name: kernel.event_listener,
                event: security.interactive_login }
            - { name: kernel.event_listener, event: kernel.response }
```

A continuación, modifica el código del *listener* para que obtenga el nuevo argumento de tipo `AuthorizationChecker` y añade la lógica necesaria para determinar la página a la que se redirecciona al usuario en función de su tipo:

```php
// src/AppBundle/Listener/LoginListener.php
namespace AppBundle\Listener;

use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\Routing\Router;
use Symfony\Component\Security\Core\Authorization\AuthorizationChecker;
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    private $checker, $router, $ciudad = null;

    public function __construct(AuthorizationChecker $checker, Router $router)
    {
        $this->checker = $checker;
        $this->router = $router;
    }

    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
    {
        $token = $event->getAuthenticationToken();
        $this->ciudad = $token->getUser()->getCiudad()->getSlug();
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        if (null === $this->ciudad) {
            return;
        }

        if($this->checker->isGranted('ROLE_TIENDA')) {
            $urlPortada = $this->router->generate('extranet_portada');
        }
        else {
            $urlPortada = $this->router->generate('portada', array(
                'ciudad' => $this->ciudad
            ));
        }

        $event->setResponse(new RedirectResponse($urlPortada));
        $event->stopPropagation();
    }
}
```

13.4 Creando un *security voter* propio

Uno de los requerimientos de la *extranet* es que permita modificar la información de las ofertas que todavía no han sido publicadas o revisadas. Comprobar que el usuario que quiere modificar una

oferta tenga el *role* `ROLE_TIENDA` no es suficiente, ya que una tienda sólo debe poder modificar sus propias ofertas.

El componente de seguridad de Symfony utiliza *voters* para tomar todas las decisiones relacionadas con la parte de la autorización. Cuando un usuario solicita acceso a un recurso, el componente de seguridad organiza una *votación*, en la que participan todos los *voters* registrados en la aplicación.

El componente de seguridad entrega a cada *voter* la información sobre el usuario y el recurso solicitado y cada uno emite su veredicto (permitir o denegar el acceso) o se abstiene. Después, Symfony tiene en consideración todas esas resoluciones para tomar la decisión final de permitir o no el acceso.

El proceso de decisión es más ágil de lo que puede parecer, ya que no todos los *voters* opinan sobre todos los casos que se le presentan. En la siguiente sección se va a crear un *security voter* encargado exclusivamente de decidir si una tienda tiene permiso para modificar la oferta solicitada.

NOTA Para las versiones de Symfony anteriores a la 2.3, este libro recomendaba y explicaba cómo utilizar ACL (*access control lists*) para solucionar este problema.

No obstante, ahora se utilizan *security voters* porque las ACL de Symfony son extremadamente difíciles de utilizar y porque los propios creadores del *framework* recomiendan los *voters* para resolver estos problemas.

13.4.1 La interfaz VoterInterface

Los *voters* de seguridad son clases PHP que implementan la interfaz `VoterInterface`. A partir de Symfony 2.8, los *voters* se han simplificado gracias a la introducción de una nueva clase `Voter` de la que pueden extender tus *voters*. Esta clase es abstracta y requiere que implementes los dos siguientes métodos:

```
use use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class CreadorOfertaVoter extends Voter
{
    protected function supports($attribute, $subject);
    protected function voteOnAttribute($attribute, $subject, TokenInterface $tok
en);
}
```

El método `supports()` comprueba si este *voter* debe aplicarse al `$attribute` y `$subject` que le pasas. El `$attribute` o *atributo* normalmente es el nombre del permiso que está solicitando el usuario (ej. `ROLE_EDITAR_OFERTA`). El `$subject` o *sujeto* es el elemento sobre el que está pidiendo el permiso; normalmente es un objeto (ej. una instancia concreta de la entidad `Oferta`) pero no es obligatorio que lo sea (ej. puedes pasar una cadena de texto que representa la ruta de un recurso que se quiere descargar).

Si `supports()` devuelve `true`, Symfony aplica este *voter* cuando se llama a la función `isGranted()`. Devuelve `false` para que no se aplique.

El método `voteOnAttribute()` es el más importante, ya que incluye toda la lógica propia del *voter* para decidir si se concede o deniega el permiso solicitado. Esta lógica puede ser tan sencilla o tan compleja como necesites y puede hacer uso de los servicios de la aplicación.

El primer argumento del método `voteOnAttribute()` es el `$attribute` o *atributo* (ej. el *rol* `ROLE_EDITAR_OFERTA`); el segundo argumento es el `$subject` o *sujeto* sobre el que se solicita el permiso (ej. un objeto de tipo `Oferta`); el tercer argumento es el *token* que representa al usuario que está solicitando el permiso.

La respuesta devuelta por `voteOnAttribute()` debe ser un valor *booleano* que es `true` cuando el permiso solicitado se concede y `false` cuando se deniega.

13.4.2 Creando un *security voter* propio

Por convención, los *voters* se crean en el directorio `Security/` de los *bundles* y sus nombres acaban en `Voter`. Así que para definir el *voter* que decide si una oferta puede ser editada por su creador, crea el archivo `src/AppBundle/Security/CreadorOfertaVoter.php` con el siguiente contenido:

```
// src/AppBundle/Security/CreadorOfertaVoter.php
namespace AppBundle\Security;

use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class CreadorOfertaVoter extends Voter
{
    public function supports($attribute, $subject)
    {
    }

    protected function voteOnAttribute($attribute, $subject, TokenInterface $tok
en)
    {
    }
}
```

El contenido del método `supports()` es muy sencillo para este *voter*, ya que sólo se ocupa de un permiso llamado `ROLE_EDITAR_OFERTA` y de los objetos de tipo `Oferta`:

```
// src/AppBundle/Security/CreadorOfertaVoter.php
namespace AppBundle\Security;

use AppBundle\Entity\Oferta;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class CreadorOfertaVoter extends Voter
{
    public function supports($attribute, $subject)
```

```
    {
        return $subject instanceof Oferta && 'ROLE_EDITAR_OFERTA' === $attribut
e;
    }

    // ...
}
```

En el caso del *voter* que se está desarrollando, la lógica consiste en comparar si el `id` de la tienda que solicita el permiso coincide con el `id` asociado a la tienda de la oferta que se quiere modificar:

```
// src/AppBundle/Security/CreadorOfertaVoter.php
namespace AppBundle\Security;

use AppBundle\Entity\Oferta;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class CreadorOfertaVoter extends Voter
{
    // ...

    protected function voteOnAttribute($attribute, $subject, TokenInterface $tok
en)
    {
        $tienda = $token->getUser();

        return $tienda instanceof Tienda && $subject->getTienda()->getId() ===
$tienda->getId();
    }
}
```

A continuación se muestra el código completo del *voter* resultante:

```
// src/AppBundle/Security/CreadorOfertaVoter.php
namespace AppBundle\Security;

use AppBundle\Entity\Oferta;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class CreadorOfertaVoter extends Voter
{
    public function supports($attribute, $subject)
    {
        return $subject instanceof Oferta && 'ROLE_EDITAR_OFERTA' === $attribut
e;
    }
```

```
    protected function voteOnAttribute($attribute, $subject, TokenInterface $tok
en)
    {
        $tienda = $token->getUser();

        return $tienda instanceof Tienda && $subject->getTienda()->getId() ===
$tienda->getId();
    }
}
```

El último paso para activar este *voter* propio consiste en definir un nuevo servicio y etiquetarlo con la etiqueta especial `security.voter`:

```
# app/config/services.yml
services:
    # ...

    app.security.oferta_voter:
        class: AppBundle\Security\CreadorOfertaVoter
        tags:
            - { name: security.voter }
```

Si la lógica del *voter* requiriera el acceso a la base de datos o a otros servicios y parámetros del contenedor, añade estos servicios como argumentos del servicio anterior:

```
# app/config/services.yml
services:
    # ...

    app.security.oferta_voter:
        class: AppBundle\Security\CreadorOfertaVoter
        arguments: ['@doctrine.orm.entity_manager', '@service_container']
        tags:
            - { name: security.voter }
```

13.4.3 Estableciendo la estrategia de decisión de los *voters*

En las aplicaciones Symfony resulta habitual que varios *voters* intervengan en el proceso de autorización. Así que aunque en esta aplicación sólo se ha definido un *voter* propio, en realidad Symfony ejecuta varios *voters* internos.

Tomar la decisión sobre si el usuario tiene la autorización solicitada no siempre es sencillo, ya que algunos *voters* pueden decir que sí y otros que no. Por eso el componente de seguridad de Symfony te permite elegir la estrategia a utilizar para tomar la decisión:

- `affirmative`: basta con que un solo *voter* diga que sí para que el usuario esté autorizado.
- `unanimous`: todos los *voters* tienen que decir que sí para autorizar al usuario.
- `consensus`: si una mayoría simple de *voters* dice que sí, el usuario tiene autorización.

La estrategia que emplea Symfony por defecto es `affirmative`, lo cual no es muy adecuado para el tipo de *voter* que se ha diseñado en la sección anterior. Para cambiar la estrategia a `unanimous`, añade la siguiente configuración en el archivo `security.yml`:

```
# app/config/security.yml
security:
    access_decision_manager:
        strategy: unanimous
```

13.4.4 Comprobando los permisos

Una vez creado y activado el *security voter* propio, resula trivial comprobar si una tienda tiene permiso para modificar la oferta solicitada. Para ello, utiliza el método `isGranted()` del componente de seguridad, que comprueba si el usuario *logueado* en la aplicación dispone del permiso necesario sobre el objeto solicitado.

Para ello, pasa como primer argumento una cadena de texto con el nombre del permiso solicitado (`ROLE_EDITAR_OFERTA`) y pasa como segundo argumento el objeto de la oferta que se quiere modificar. El método `isGranted()` devuelve `true` cuando el usuario dispone del permiso y `false` en cualquier otro caso.

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// ...

public function ofertaEditarAction($id)
{
    $oferta = $em->getRepository('AppBundle:Oferta')->find($id);

    if (false === $this->isGranted('ROLE_EDITAR_OFERTA', $oferta)) {
        throw new AccessDeniedException();
    }

    // ...
}
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 14

Creando la parte de administración

La *extranet* de la aplicación se divide en tres partes bien diferenciadas:

- La portada que muestra el listado de ofertas de la tienda y la página de detalle con las ventas de una oferta.
- El formulario que muestra y permite modificar la información sobre la tienda.
- El formulario que permite crear una nueva oferta o modificar una oferta existente que no haya sido ni publicada ni revisada.

En este capítulo se desarrollan las dos primeras partes y el siguiente capítulo explica detalladamente cómo crear la tercera parte.

14.1 Creando la portada de la *extranet*

La portada de la *extranet* muestra el listado de todas las ofertas de la tienda que está *logueada*, sin importar si han sido aprobadas o no. El listado no incluye un paginador porque se considera que el número de ofertas de una misma tienda no es demasiado grande.

Abre el controlador de la *extranet* y añade la siguiente acción `portadaAction()` para responder a la ruta `extranet_portada`:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExtranetController extends Controller
{
    // ...

    public function portadaAction()
    {
        $em = $this->getDoctrine()->getManager();

        $tienda = $this->getUser();
        $ofertas = $em->getRepository('AppBundle:Tienda')
                      ->findOfertasRecientes($tienda->getId());

        return $this->render('extranet/portada.html.twig', array(
```

```
            'ofertas' => $ofertas
        ));
    }
}
```

Para que el controlador sea más conciso, se crea una búsqueda propia en el repositorio de la entidad `Tienda`. Así que añade el siguiente método `findOfertasRecientes()` que admite como primer argumento el `id` de la tienda y como segundo argumento el número de ofertas que se devuelven:

```
// src/AppBundle/Repository/TiendaRepository.php
namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;

class TiendaRepository extends EntityRepository
{
    // ...

    public function findOfertasRecientes($tienda_id, $limite = null)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t FROM AppBundle:Oferta o JOIN o.tienda t
             WHERE o.tienda = :id
          ORDER BY o.fechaExpiracion DESC
        ');
        $consulta->setParameter('id', $tienda_id);

        if (null != $limite) {
            $consulta->setMaxResults($limite);
        }

        return $consulta->getResult();
    }
}
```

Por último, crea la plantilla `portada.html.twig` para mostrar el listado de ofertas que obtiene el controlador:

```
{# app/Resources/views/extranet/portada.html.twig #}
{% extends 'extranet.html.twig' %}

{% block id 'portada' %}
{% block title %}Administración de {{ app.user.nombre }}{% endblock %}

{% block article %}
<h1>Todas tus ofertas</h1>
```

```
<table>
    <thead>
        <tr>
            <th>Revisada</th>
            <th>Se publica</th>
            <th>Finaliza</th>
            <th>Nombre</th>
            <th>Ventas</th>
            <th>Acciones</th>
        </tr>
    </thead>

    <tbody>
        {% for oferta in ofertas %}
        <tr>
            <td>{{ oferta.revisada ? 'si' : 'no' }}</td>

            {% if oferta.revisada %}
            <td>{{ oferta.fechaPublicacion|localizeddate('medium', 'short')
}}</td>
            <td>{{ oferta.fechaExpiracion|localizeddate('medium', 'short') }}</t
d>
            {% else %}
            <td colspan="2">Pendiente de revisión</td>
            {% endif %}

            <td>{{ oferta.nombre }}</td>
            <td>{{ oferta.compras }}</td>
            <td>
                <ul>
                    {% if oferta.compras > 0 %}
                    <li>
                        <a href="{{ path('extranet_oferta_ventas',
                                    { id: oferta.id }) }}">
                            Lista de ventas
                        </a>
                    </li>
                    {% endif %}

                    {% if not oferta.revisada %}
                    <li>
                        <a href="{{ path('extranet_oferta_editar',
                                    { id: oferta.id }) }}">
                            Modificar
                        </a>
                    </li>
                    {% endif %}
```

```
                </ul>
            </td>
        </tr>
        {% endfor %}
    </tbody>
</table>
{% endblock %}
```

La plantilla obtiene el nombre de la tienda directamente a través de la variable especial de Twig `app.user`, que guarda la información del usuario actualmente conectado. Así no es necesario crear un nueva variable en el controlador y después pasarla a la plantilla:

```
{% block title %}Administración de {{ app.user.nombre }}{% endblock %}
```

El siguiente elemento destacable es la forma en la que se muestra el contenido de la propiedad *booleana* `oferta.revisada`:

```
<td>{{ oferta.revisada ? 'si' : 'no' }}</td>
```

Aunque mostrar `si` y `no` es suficiente, en ocasiones este tipo de campos se muestran con gráficos que indican más claramente si el valor es `true` o `false`. Gracias a las entidades HTML puedes mostrar estos símbolos gráficos sin necesidad de crear imágenes:

```
<td>{{ oferta.revisada ? '&#10004;' : '&#10008;' }}</td>
```

Observa también cómo las fechas de publicación y expiración se muestran mediante el filtro `localizeddate` de la extensión `intl` de Twig que se activó en los capítulos anteriores y que permite mostrar la fecha en varios idiomas y con varios formatos predefinidos:

```
{% if oferta.revisada %}
<td>{{ oferta.fechaPublicacion|localizeddate('medium', 'short') }}</td>
<td>{{ oferta.fechaExpiracion|localizeddate('medium', 'short') }}</td>
{% else %}
<td colspan="2">Pendiente de revisión</td>
{% endif %}
```

El enlace para ver el listado de ventas de una oferta sólo se muestra si la oferta tiene alguna venta. Esto es fácil de comprobar gracias a la propiedad `compras` de la oferta:

```
{% if oferta.compras > 0 %}
<li>
    <a href="{{ path('extranet_oferta_ventas', { id: oferta.id }) }}">
        Lista de ventas
    </a>
</li>
{% endif %}
```

Por último, las tiendas sólo pueden modificar sus ofertas si estas no han sido todavía revisadas por un administrador:

```
{% if not oferta.revisada %}
<li>
    <a href="{{ path('extranet_oferta_editar', { id: oferta.id }) }}">
        Modificar
    </a>
</li>
{% endif %}
```

14.2 Mostrando las ventas de una oferta

Al pulsar el enlace *Lista de ventas* sobre una oferta de la portada de la *extranet*, se muestra un listado detallado con la fecha de compra, el nombre y apellidos y el DNI de cada comprador.

El controlador necesario para obtener la lista de ventas es muy similar al controlador desarrollado en la sección anterior. El motivo es que la parte de administración de un sitio web es muy repetitivo, ya que casi todas las páginas son listados de elementos:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExtranetController extends Controller
{
    // ...

    public function ofertaVentasAction($id)
    {
        $em = $this->getDoctrine()->getManager();

        $ventas = $em->getRepository('AppBundle:Oferta')
                     ->findVentasByOferta($id);

        return $this->render('extranet/ventas.html.twig', array(
            'oferta' => $ventas[0]->getOferta(),
            'ventas' => $ventas
        ));
    }
}
```

Para que la plantilla pueda obtener fácilmente los datos de la oferta, el controlador crea la variable `oferta` a partir del primer elemento del array de ventas. Como es habitual, la consulta a la base de datos se realiza mediante un método propio en el repositorio de la entidad. Abre el archivo del repositorio `Oferta` y añade el siguiente método `findVentasByOferta()`:

```
// src/AppBundle/Repository/OfertaRepository.php
namespace AppBundle\Entity;
use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
```

```
{
    // ...

    public function findVentasByOferta($oferta)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT v, o, u
              FROM AppBundle:Venta v
              JOIN v.oferta o JOIN v.usuario u
             WHERE o.id = :id
          ORDER BY v.fecha DESC
        ');
        $consulta->setParameter('id', $oferta);

        return $consulta->getResult();
    }
}
```

Con la información que le pasa el controlador, la plantilla resultante es realmente sencilla:

```
{# app/Resources/views/extranet/ventas.html.twig #}
{% extends 'extranet.html.twig' %}

{% block id 'oferta' %}
{% block title %}Ventas de la oferta {{ oferta.nombre }}{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<table>
    <thead>
        <tr>
            <th>DNI</th>
            <th>Nombre y apellidos</th>
            <th>Fecha venta</th>
        </tr>
    </thead>

    <tbody>
        {% for venta in ventas %}
        <tr>
           <td>{{ venta.usuario.dni }}</td>
           <td>{{ venta.usuario.nombre ~ ' ' ~ venta.usuario.apellidos }}</td>
           <td>{{ venta.fecha | localizeddate('long', 'medium') }}</td>
        </tr>
        {% endfor %}
        <tr>
```

```
            <th>TOTAL</th>
            <td>{{ ventas|length * oferta.precio }} &euro;</td>
            <td>{{ ventas|length }} ventas</td>
        </tr>
    </tbody>
</table>
{% endblock %}
```

14.3 Mostrando el perfil de la tienda

Cuando una tienda está *logueada* en la *extranet* y accede a la ruta `extranet_perfil`, se muestra un formulario con todos sus datos. Este formulario también permite modificar cualquier dato y guardar los cambios.

14.3.1 El formulario

Los formularios de las páginas internas de la aplicación normalmente son sencillos de crear porque contienen todos los campos de la entidad que modifican. Además, al contrario de los formularios que se muestran en el *frontend*, no suele ser necesario realizar muchos ajustes en su aspecto.

Para crear rápidamente este tipo de formularios, Symfony dispone de un comando llamado `doctrine:generate:form` que genera un formulario con todos los campos de la entidad que se indica como argumento:

```
$ php app/console doctrine:generate:form AppBundle:Tienda

The new TiendaType.php class file has been created under
.../src/AppBundle/Form/TiendaType.php.
```

Si observas el formulario `TiendaType` generado automáticamente, verás que su código es muy sencillo:

```
// src/AppBundle/Form/TiendaType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class TiendaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('slug')
            ->add('login')
            ->add('password')
            ->add('salt')
```

```
            ->add('descripcion')
            ->add('direccion')
            ->add('ciudad')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Tienda'
        ));
    }
}
```

A continuación, ajusta los tipos y opciones de los campos y añade el botón para poder enviar el formulario:

```
// src/AppBundle/Form/TiendaType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class TiendaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('login', 'text', array('read_only' => true))

            ->add('password', 'repeated', array(
                'type' => 'password',
                'invalid_message' => 'Las dos contraseñas deben coincidir',
                'first_options'   => array('label' => 'Contraseña'),
                'second_options'  => array('label' => 'Repite Contraseña'),
                'required'        => false
            ))

            ->add('descripcion')
            ->add('direccion')
            ->add('ciudad')

            ->add('guardar', 'submit', array('label' => 'Guardar cambios'))
        ;
    }
```

```
    // ...
}
```

Como las tiendas no pueden modificar su nombre de usuario, el primer cambio necesario es añadir la opción `read_only` a `true` sobre el campo `login`. Al ser un campo de sólo lectura, la tienda podrá ver su login, pero no podrá modificarlo.

El otro cambio necesario en el formulario es el del campo `password`. Además de indicar que es de tipo `password` (para que sus contenidos no se vean por pantalla) se transforma en un campo `repeated` para que se muestre con dos cuadros de texto repetidos:

- Si el usuario escribe en uno de los campos y deja vacío el otro o si escribe dos valores diferentes, se muestra un mensaje de error indicando que los dos campos deben ser iguales.
- Si el usuario no escribe nada en ningún campo de contraseña, se entiende que el usuario no quiere modificar su contraseña.
- Si el usuario escribe cualquier valor idéntico en los dos campos, ese valor es la nueva contraseña del usuario.

NOTA La gestión de las contraseñas de las tiendas es exactamente igual que la que se explicó con detalle para los usuarios en el Capítulo 8. Resulta recomendable resolver esta funcionalidad de la misma manera añadiendo una nueva propiedad `$passwordEnClaro` en la entidad `Tienda` que permita gestionar más fácilmente los cambios de contraseña.

14.3.2 El controlador

El controlador asociado a la ruta `extranet_perfil` sigue el esquema habitual de los controladores que muestran un formulario y también se encargan de procesar los datos enviados por el usuario:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExtranetController extends Controller
{
    // ...

    public function perfilAction(Request $request)
    {
        $tienda = $this->getUser();
        $formulario = $this->createForm('AppBundle\Form\TiendaType', $tienda);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            // procesar formulario ...
```

```
        }

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

La tienda actualmente *logueada* se obtiene a través del atajo `getUser()` del controlador (que es equivalente a ejecutar el siguiente código: `$this->get('security.token_storage')->getToken()->getUser()`). Después, se crea un formulario de tipo `TiendaType` y se rellena con los datos de la tienda:

```
$tienda = $this->getUser();
$formulario = $this->createForm('AppBundle\Form\TiendaType', $tienda);
```

Después, si la petición es de tipo `POST` se procesa la información enviada por el usuario y se guardan los cambios en la base de datos. Si no, se muestra directamente el formulario, preparándolo para la plantilla con el método `$formulario->createView()`:

```
return $this->render('extranet/perfil.html.twig', array(
    'tienda'     => $tienda,
    'formulario' => $formulario->createView()
));
```

El procesado de la información enviada por la tienda es muy sencillo gracias a las utilidades que proporciona Symfony:

```
// src/AppBundle/Controller/ExtranetController.php
use Symfony\Component\HttpFoundation\Request;

class ExtranetController extends Controller
{
    // ...

    public function perfilAction(Request $request)
    {
        $tienda = $this->getUser();
        $formulario = $this->createForm('AppBundle\Form\TiendaType', $tienda);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($tienda);
            $em->flush();

            $this->addFlash('info', 'Los datos de tu perfil se han actualizado c
```

```
orrectamente');

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

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

14.3.3 La plantilla

La plantilla asociada al controlador que se acaba de desarrollar simplemente muestra el formulario con los datos de la tienda, por lo que podría ser tan sencilla como lo siguiente:

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

{% block title %}Ver / Modificar mis datos{% endblock %}

{% block article %}
    {{ form(formulario) }}
{% endblock %}
```

No obstante, como se quiere modificar el título o *label* de algunos campos y como se quiere añadir algún mensaje de ayuda, es mejor mostrar el formulario con las funciones especiales para formularios:

```
{# app/Resources/views/extranet/perfil.html.twig #}
{% extends 'extranet.html.twig' %}

{% block id 'tienda' %}
{% block title %}Ver / Modificar mis datos{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

{{ form_start(formulario) }}
<div>
    {{ form_errors(formulario) }}

    <div>
        {{ form_row(formulario.nombre) }}
    </div>

    <div>
        {{ form_label(formulario.login, 'Nombre de usuario') }}
        {{ form_errors(formulario.login) }}
        {{ form_widget(formulario.login) }}
```

```
    </div>

    <div>
        {{ form_widget(formulario.password) }}

        <p class="ayuda">Si quieres cambiar la contraseña, escríbela dos
        veces. Si no quieres cambiarla, deja su valor vacío.</p>
    </div>

    <div>
        {{ form_label(formulario.descripcion, 'Descripción') }}
        {{ form_errors(formulario.descripcion) }}
        {{ form_widget(formulario.descripcion) }}
    </div>

    <div>
        {{ form_row(formulario.ciudad) }}
    </div>

    <div>
        {{ form_label(formulario.ciudad) }}
        {{ form_errors(formulario.ciudad) }}
        {{ form_widget(formulario.ciudad) }}
    </div>
</div>
{{ form_end(formulario) }}
{% endblock %}
```

CAPÍTULO 15

Administrando las ofertas

15.1 Creando ofertas

Las tiendas conectadas a la *extranet* pueden añadir nuevas ofertas pulsando el botón *Añadir oferta* que se muestra en la zona lateral de todas las páginas. Las ofertas añadidas no se publican directamente en el sitio web, sino que primero deben revisarlas los administradores.

La base de datos almacena para cada oferta mucha información que la tienda no puede modificar, como la fecha de publicación y expiración, las compras, el *slug* y si ha sido revisada o no. Por eso es mejor crear a mano el formulario que utilizan las tiendas para añadir y modificar ofertas. De esta forma, se tiene un control más preciso sobre los campos que se añaden.

15.1.1 El formulario

Crea un archivo llamado `OfertaType.php` en el directorio `Form/` de AppBundle con el siguiente contenido:

```
// src/AppBundle/Form/OfertaType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('descripcion')
            ->add('condiciones')
            ->add('rutaFoto', 'file', array('required' => false))
            ->add('precio', 'money')
            ->add('descuento', 'money')
            ->add('umbral')
            ->add('guardar', 'submit', array('label' => 'Guardar cambios'))
        ;
    }
```

```
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Oferta'
        ));
    }

    public function getBlockPrefix()
    {
        return 'oferta';
    }
}
```

Como Symfony no es capaz de adivinar que los campos `precio` y `descuento` sirven para introducir cifras que representan dinero, se indica explícitamente que son campos de tipo `money`. Igualmente, el campo `foto` es de tipo `string` en la entidad `Oferta`, por lo que es necesario convertirlo en un campo de tipo `file` que permita subir fotos.

15.1.2 Validación de información

Cuando una tienda envía información a través de un formulario, antes de guardarla en la base de datos el controlador comprueba que la información sea válida. Symfony permite establecer la información de validación en formato YAML, XML, PHP y con anotaciones.

Normalmente las anotaciones son el método más conveniente de añadir esta información, ya que se definen en el mismo archivo de la entidad. Así que añade lo siguiente en la entidad `Oferta`:

```
// AppBundle/Entity/Oferta.php

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

class Oferta {
    // ...

    /**
     * @ORM\Column(type="string")
     *
     * @Assert\NotBlank()
     */
    protected $nombre;

    /**
     * @ORM\Column(type="string")
     *
     * @Assert\NotBlank()
     */
    protected $slug;
```

```
    /**
     * @ORM\Column(type="text")
     *
     * @Assert\NotBlank()
     * @Assert\Length(min = 30)
     */
    protected $descripcion;

    /**
     * @ORM\Column(type="text")
     */
    protected $condiciones;

    /**
     * @ORM\Column(type="string")
     */
    protected $rutaFoto;

    /**
     * @ORM\Column(type="decimal", scale=2)
     *
     * @Assert\Range(min = 0)
     */
    protected $precio;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $descuento;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     *
     * @Assert\DateTime
     */
    protected $fechaPublicacion;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     *
     * @Assert\DateTime
     */
    protected $fechaExpiracion;

    /**
     * @ORM\Column(type="integer")
     */
    protected $compras;
```

```
    /**
     * @ORM\Column(type="integer")
     *
     * @Assert\Range(min = 0)
     */
    protected $umbral;

    // ...
}
```

Además de las reglas de validación de cada propiedad, Symfony permite añadir validaciones dinámicas, tal como se explicó en la sección *Métodos de validación ad-hoc* (página 237) del capítulo 8.

Este tipo de validaciones son muy útiles cuando se necesita comparar el valor de dos o más propiedades. En la entidad `Oferta` por ejemplo, es importante asegurarse de que la fecha de expiración sea posterior a la fecha de publicación.

Para ello, crea en la entidad un nuevo método llamado `isFechaValida()` y añade como información de validación `@Assert\True` (incluyendo opcionalmente el mensaje que se muestra cuando se produce un error de validación). El código del método simplemente compara las dos fechas de la oferta y devuelve `false` cuando la fecha de expiración sea anterior a la fecha de publicación:

```
// AppBundle/Entity/Oferta.php
class Oferta {
    // ...

    /**
     * @Assert\True(message = "La fecha de expiración debe ser posterior a
     *                         la fecha de publicación")
     */
    public function isFechaValida()
    {
        if ($this->fechaPublicacion == null
            ||
            $this->fechaExpiracion == null) {
            return true;
        }

        return $this->fechaExpiracion > $this->fechaPublicacion;
    }
}
```

15.1.3 El controlador

Después de todo lo explicado en los capítulos anteriores, resulta muy sencillo crear el controlador que se encarga de mostrar y de procesar el formulario creado anteriormente:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;
```

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use AppBundle\Entity\Oferta;

class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction(Request $request)
    {
        $oferta = new Oferta();
        $formulario = $this->createForm('AppBundle\Form\OfertaType', $oferta);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($oferta);
            $em->flush();

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

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

Si pruebas el controlador anterior, se producen varios errores al guardar los datos, ya que en la entidad falta información importante que no se puede establecer mediante el formulario (propiedades `tienda`, `ciudad`, `compras`). Así que completa el controlador con el siguiente código que establece el valor inicial de algunas propiedades importantes de la entidad:

```
// src/AppBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction()
    {
        $oferta = new Oferta();

        // ...

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
```

```
            $tienda = $this->getUser();

            $oferta->setCompras(0);
            $oferta->setTienda($tienda);
            $oferta->setCiudad($tienda->getCiudad());

            // ...
        }

        // ...
    }
}
```

15.1.4 Subiendo fotos

Antes de dar por finalizado el controlador es necesario añadir la lógica que procesa la foto de la oferta. La entidad dispone de una propiedad llamada `rutaFoto` que no guarda la imagen sino su ruta. Por tanto, debemos asegurarnos de que la foto se copia en algún directorio preparado para ello y que en la entidad se guarda solamente su ruta.

El truco consiste en añadir una nueva propiedad *virtual* tanto a la entidad `Oferta` como al formulario que se utiliza para modificarla. Esta nueva propiedad, llamada `foto`, permitirá subir fotos con el formulario y almacenará temporalmente la foto hasta que se copie definitivamente en algún directorio. Se dice que esta propiedad es *virtual* porque su información no se va a persistir en la base de datos.

Abre en primer lugar el archivo de la entidad `Oferta` y añade una nueva propiedad llamada `foto` con la siguiente configuración:

```
// src/AppBundle/Entity/Oferta.php

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity(repositoryClass="AppBundle\Repository\OfertaRepository")
 */
class Oferta
{
    // ...

    /**
     * @Assert\Image(maxSize = "500k")
     */
    protected $foto;

    /**
     * @param UploadedFile $foto
```

```
     */
    public function setFoto(UploadedFile $foto = null)
    {
        $this->foto = $foto;
    }

    /**
     * @return UploadedFile
     */
    public function getFoto()
    {
        return $this->foto;
    }
}
```

Observa que la nueva propiedad `foto` no define una anotación de tipo `@ORM\Column(...)`, lo que significa que su información no se va a guardar en la base de datos. En cambio, esta propiedad sí que define una anotación de tipo `@Assert\Image(...)`, para asegurar que el archivo subido sea una foto y su tamaño no supere los 500 KB.

Por otra parte, el atributo que el formulario pasará automáticamente al *setter* de la propiedad `foto` es de tipo `UploadedFile`, que permite manipular y extraer fácilmente información del archivo subido. No olvides importar la clase `UploadedFile` con la instrucción `use` correspondiente.

Finalmente, actualiza el formulario con el que se manipula la información de la oferta. Elimina el campo `rutaFoto` y sustitúyelo por un nuevo campo llamado `foto` y de tipo `file`:

```
// src/AppBundle/Form/OfertaType.php
class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('foto', 'file', array('required' => false))
        ;
    }
}
```

Después de realizar estos cambios, el formulario ya permite subir fotos y la variable `$formulario` del controlador ya tiene acceso a la foto subida por el usuario. El siguiente paso consiste en guardar la foto en el directorio de las imágenes y actualizar convenientemente el valor de la propiedad `rutaFoto`, que es realmente la que se guarda en la base de datos.

Como la lógica descrita anteriormente se puede necesitar en más de un controlador de la aplicación, es mejor incluirla en la propia entidad. Así que en el controlador solamente es necesario añadir una llamada al método `subirFoto()` que se creará posteriormente en la entidad:

```
// src/AppBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction()
    {
        $oferta = new Oferta();

        // ...

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            $tienda = $this->getUser();

            $oferta->setCompras(0);
            $oferta->setTienda($tienda);
            $oferta->setCiudad($tienda->getCiudad());

            $oferta->subirFoto();

            // ...
        }

       // ...
    }
}
```

La gran ventaja de las entidades de Doctrine es que son clases PHP normales. Por eso puedes añadir tantos métodos como necesites para encapsular la lógica relacionada con la información que gestiona esa entidad. Así que abre el archivo de la entidad `Oferta` y crea un nuevo método llamado `subirFoto()` con el siguiente contenido:

```
// src/AppBundle/Entity/Oferta.php
class Oferta
{
    // ...

    public function subirFoto()
    {
        if (null === $this->foto) {
            return;
        }

        $directorioDestino = __DIR__.'/../../../../web/uploads/images';
        $nombreArchivoFoto = uniqid('cupon-').'-foto1.jpg';

        $this->foto->move($directorioDestino, $nombreArchivoFoto);
```

```
            $this->setRutaFoto($nombreArchivoFoto);
        }
    }
```

El método `subirFoto()` genera un nombre único para la foto concatenando el prefijo `cupon-`, una cadena aleatoria y el sufijo `-foto1.jpg`. Si en tu aplicación prefieres mantener el nombre original de la foto, utiliza el método `getClientOriginalName()`:

```
$nombreArchivoFoto = $this->foto->getClientOriginalName();
```

Si lo necesitas, también dispones de los métodos `getClientMimeType()` para obtener el tipo MIME original de la foto, `getClientSize()` para determinar el tamaño original de la foto en bytes y `getClientOriginalExtension()` para obtener la extensión del archivo original de la foto.

Una vez definido el nombre con el que se guarda la foto y su directorio de destino, ya puedes guardar el archivo de la foto en ese directorio utilizando el método `move()` de la clase `UploadedFile`:

```
$this->foto->move($directorioDestino, $nombreArchivoFoto);
```

Por último, se actualiza el valor de la propiedad `rutaFoto` para almacenar en la base de datos la ruta correcta a la foto que se acaba de mover:

```
$this->setRutaFoto($nombreArchivoFoto);
```

Una mejora que se puede introducir en el método `subirFoto()` de la entidad es evitar el uso de rutas de directorios escritas directamente en el código de la entidad:

```
public function subirFoto()
{
    $directorioDestino = __DIR__.'/../../../../web/uploads/images';

    // ...
}
```

Para facilitar el mantenimiento de la aplicación, este tipo de información debería incluirse en un archivo de configuración. Así que abre el archivo `config.yml` de la aplicación y crea un nuevo parámetro llamado `cupon.directorio.imagenes`:

```
# app/config/config.yml

# ...
parameters:
    cupon.directorio.imagenes: %kernel.root_dir%/../web/uploads/images/
```

Las imágenes subidas desde la *extranet* se guardan en el directorio `uploads/images/` dentro del directorio público de la aplicación (`web/`). Una buena práctica en los archivos de configuración consiste en no utilizar rutas absolutas, ya que esto dificulta instalar la aplicación en diferentes servidores.

Lamentablemente Symfony no define un parámetro especial para la ruta del directorio `web/`, pero su valor se puede obtener fácilmente a partir del parámetro `%kernel.root_dir%`, que corresponde al directorio `app/`.

El siguiente paso consiste en obtener el valor del parámetro `cupon.directorio.imagenes` desde la entidad `Oferta`. Como las entidades de Doctrine son clases PHP normales y corrientes, no puedes utilizar directamente el contenedor de inyección de dependencias dentro de su código. Así que cuando los métodos de una entidad necesitan acceder a los parámetros del contenedor, la mejor solución consiste en pasar el parámetro desde el propio controlador:

```
// Controlador: src/AppBundle/Controller/ExtranetController.php
public function ofertaNuevaAction()
{
    // ...

    $oferta->subirFoto(
        $this->container->getParameter('cupon.directorio.imagenes')
    );

    // ...
}

// Entidad: src/AppBundle/Entity/Oferta.php
class Oferta
{
    // ...

    public function subirFoto($directorioDestino)
    {
        if (null === $this->foto) {
            return;
        }

        $nombreArchivoFoto = uniqid('cupon-').'-foto1.jpg';
        $this->foto->move($directorioDestino, $nombreArchivoFoto);
        $this->setFoto($nombreArchivoFoto);
    }
}
```

15.1.5 La plantilla

La plantilla necesaria para mostrar el formulario que crea las ofertas es muy sencilla gracias al uso de las funciones `form_*` de Twig:

```
{# app/Resources/views/extranet/oferta.html.twig #}
{% extends 'extranet.html.twig' %}

{% block id 'oferta' %}
{% block title %}Añadir una nueva oferta{% endblock %}
```

```
{% block article %}
<h1>{{ block('title') }}</h1>

{{ form_start(formulario) }}
<div>
    {{ form_errors(formulario) }}

    <div>
        {{ form_row(formulario.nombre) }}
    </div>

    <div>
        {{ form_label(formulario.descripcion, 'Descripción') }}
        {{ form_errors(formulario.descripcion) }}
        {{ form_widget(formulario.descripcion) }}

        <p class="ayuda">Escribe cada característica en una línea.</p>
    </div>

    <div>
        {{ form_row(formulario.condiciones) }}
    </div>

    <div>
        {{ form_label(formulario.foto, 'Fotografía') }}
        {{ form_errors(formulario.foto) }}
        {{ form_widget(formulario.foto) }}

        <p class="ayuda">Tamaño máximo: 500 KB. Formato preferido: JPEG.</p>
    </div>

    <div>
        {{ form_row(formulario.precio) }}
    </div>

    <div>
        {{ form_row(formulario.descuento) }}
    </div>

    <div>
        {{ form_label(formulario.umbral, 'Compras necesarias') }}
        {{ form_errors(formulario.umbral) }}
        {{ form_widget(formulario.umbral) }}
    </div>
</div>
{{ form_end(formulario) }}
{% endblock %}
```

```
{% block aside %}
    {# ... #}
{% endblock %}
```

15.1.6 Campos de formulario que no pertenecen al modelo

Cuando se crea o modifica una entidad a través de un formulario, lo normal es que todos los campos del formulario correspondan con alguna propiedad de la entidad. Sin embargo, en ocasiones el formulario debe incluir campos cuyo valor sería absurdo almacenar en la entidad.

Imagina el caso del formulario con el que las tiendas añaden nuevas ofertas desde la *extranet*. Legalmente puede ser necesario obligar a las tiendas a aceptar una serie de condiciones al añadir la oferta. Estas condiciones legales se pueden mostrar en forma de *checkbox* que sea obligatorio activar al crear una oferta:

Figura 15.1 Campo de formulario adicional para que las tiendas acepten las condiciones legales

Este *checkbox* no corresponde a ninguna propiedad de la entidad `Oferta`, por lo que si lo añades al formulario, se producirá un error. La solución es sencilla gracias a la opción `mapped`, que le dice al *form builder* que un determinado campo no se corresponde con ninguna propiedad de la entidad:

```
// src/AppBundle/Form/OfertaType.php
use Symfony\Component\Validator\Constraints\IsTrue;

class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('descripcion')
            // ...
            ->add('acepto', 'checkbox', array(
                'mapped'   => false,
                'constraints' => new IsTrue(array(
                    'message' => 'Debes aceptar las condiciones legales antes d
e añadir una oferta',
                ))
            ))
        ;
    }
```

```
    // ...
}
```

Cuando el valor de la opción `mapped` es `false`, el campo de formulario no se asocia a ninguna propiedad de la entidad, por lo que ya no se producirá ningún error al enviar el formulario.

Además, como el campo `acepto` no es una propiedad de la entidad, no se puede validar añadiéndole una anotación de tipo `@Assert`. La solución consiste en utilizar la opción `constraints` para definir la validación al mismo tiempo que se añade el campo de formulario. En este caso, bastaría con un simple `IsTrue()`, que solamente será válido cuando la casilla del *checkbox* esté activada.

A continuación, añade el nuevo campo en la plantilla del formulario:

```
{# app/Resources/views/extranet/oferta.html.twig #}

{# ... #}
<div>
  {{ form_errors(formulario.acepto) }}
  {{ form_widget(formulario.acepto) }} <span>Declaro que toda la información
     de esta oferta es correcta, que me comprometo a cumplir las condiciones
     prometidas y que dispongo de los medios necesarios para hacerlo.</span>
</div>
```

15.2 Modificando las ofertas

Las acciones *crear* y *modificar* de la parte de administración de un sitio web suelen ser tan parecidas que en ocasiones se fusionan en una única acción o plantilla. En esta sección se crea el controlador de la acción *modificar*, pero se reutiliza la misma plantilla de la acción *crear* desarrollada en la sección anterior.

15.2.1 El controlador

El esqueleto básico del controlador de la acción que modifica los datos de una oferta es el que se muestra a continuación:

```
// src/AppBundle/Controller/ExtranetController.php
namespace AppBundle\Controller;

// ...
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use AppBundle\Form\OfertaType;

class ExtranetController extends Controller
{
    public function ofertaEditarAction(Request $request, $id)
    {
```

```
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('AppBundle:Oferta')->find($id);

        if (!$oferta) {
            throw $this->createNotFoundException('La oferta no existe');
        }

        if (false === $this->isGranted('ROLE_EDITAR_OFERTA', $oferta)) {
            throw new AccessDeniedException();
        }

        if ($oferta->getRevisada()) {
            $this->addFlash('error', 'La oferta no se puede modificar porque ya
ha sido revisada');

            return $this->redirect($this->generateUrl('extranet_portada'));
        }

        $formulario = $this->createForm('AppBundle\Form\OfertaType', $oferta);

        $formulario->handleRequest($request);

        if ($formulario->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($oferta);
            $em->flush();

            return $this->redirect(
                $this->generateUrl('extranet_portada')
            );
        }

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

La primera parte del controlador realiza tres comprobaciones básicas relacionadas con la modificación de una oferta. La primera consiste en comprobar que la oferta solicitada existe:

```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

// ...

$oferta = $em->getRepository('AppBundle:Oferta')->find($id);
```

```
if (!$oferta) {
    throw $this->createNotFoundException('La oferta no existe');
}
```

La segunda comprobación es la más importante, ya que gracias al *security voter* propio desarrollado en el capítulo anterior, se comprueba que la tienda tenga permiso para modificar la oferta solicitada:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// ...

if (false === $this->isGranted('ROLE_EDITAR_OFERTA', $oferta)) {
    throw new AccessDeniedException();
}
```

La última comprobación asegura que la oferta no haya sido revisada por los administradores del sitio web, en cuyo caso ya no se puede seguir modificando. Si así fuera, se redirige al usuario a la portada de la *extranet* y se le muestra un mensaje de error creado mediante los *mensajes flash*:

```
if ($oferta->getRevisada()) {
    $this->addFlash('error', 'La oferta no se puede modificar porque ya ha sido
revisada');

    return $this->redirectToRoute('extranet_portada');
}
```

El resto del controlador sigue el flujo habitual que procesa los datos de las peticiones `POST` o muestra el formulario con los datos de la oferta a modificar. Si pruebas el código anterior, pronto verás que existe un grave problema con la foto de la oferta. Si la tienda no la cambia, se pierde la foto. Si la cambia, no se actualiza correctamente. El siguiente código muestra los cambios necesarios para manejar correctamente la foto:

```
// src/AppBundle/Controller/ExtranetController.php
use Symfony\Component\HttpFoundation\Request;

class ExtranetController extends Controller
{
    public function ofertaEditarAction(Request $request, $id)
    {
        // ...

        $formulario = $this->createForm('AppBundle\Form\OfertaType', $oferta);

        $rutaFotoOriginal = $formulario->getData()->getRutaFoto();

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

```
        if ($formulario->isValid()) {
            if (null == $oferta->getFoto()) {
                // La foto original no se modifica, recuperar su ruta
                $oferta->setRutaFoto($rutaFotoOriginal);
            } else {
                // La foto de la oferta se ha modificado
                $directorioFotos = $this->container->getParameter(
                    'cupon.directorio.imagenes'
                );

                $oferta->subirFoto($directorioFotos);

                // Borrar la foto anterior
                unlink($directorioFotos.$fotoOriginal);
            }

            // ...
        }

        // ...
    }
}
```

Después de la llamada al método `handleRequest()` se pierden los datos originales que guardaba el objeto `$oferta` y se sustituyen por los datos enviados a través del formulario, sean válidos o no. Así que primero se guarda la ruta original de la foto:

```
$rutaFotoOriginal = $formulario->getData()->getRutaFoto();
$formulario->handleRequest($request);
// ...
```

Si los datos del formulario son válidos, el siguiente paso consiste en comprobar si la foto se ha modificado o no. Cuando no se modifica, el formulario le asigna el valor `null` a la propiedad `foto`. En este caso, simplemente se vuelve a asignar la ruta original guardada previamente:

```
if ($formulario->isValid()) {
    if (null == $oferta->getFoto()) {
        $oferta->setRutaFoto($rutaFotoOriginal);
    } else {
        // ...
    }

    // ...
}
```

Si la foto se ha modificado, se copia la nueva foto en el directorio de las fotos subidas (mediante el método `subirFoto()` de la entidad `Oferta`) y se borra la foto anterior:

```
if ($formulario->isValid()) {
    if (null == $oferta->getFoto()) {
        $oferta->setRutaFoto($rutaFotoOriginal);
    } else {
        $directorioFotos = $this->container->getParameter(
            'cupon.directorio.imagenes'
        );

        $oferta->subirFoto($directorioFotos);
        unlink($directorioFotos.$fotoOriginal);
    }

    // ...
}
```

TRUCO Si no quieres encargarte de la gestión de las fotos subidas (copiarlas en el directorio destino, borrar las fotos que se han modificado, etc.) puedes hacer que Doctrine se encargue de todas esas tareas gracias a los *Lifecycle Callbacks*. Consulta el artículo *How to handle File Uploads with Doctrine* (http://symfony.com/doc/current/cookbook/doctrine/file_uploads.html#using-lifecycle-callbacks) de la documentación oficial de Symfony para obtener más información.

15.2.2 La plantilla

La plantilla de la acción *modificar* es muy similar a la de la acción *crear*, así que utiliza la misma plantilla Twig para las dos acciones. El primer cambio que debes hacer es pasar desde el controlador una nueva variable que indique si la `accion` es `crear` o `editar`:

```
// src/AppBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    public function ofertaNuevaAction($id)
    {
        // ...

        return $this->render('extranet/formulario.html.twig',
            array(
                'accion'     => 'crear',
                'formulario' => $formulario->createView()
            )
        );
    }

    public function ofertaEditarAction($id)
    {
        // ...

        return $this->render('extranet/formulario.html.twig',
```

```
            array(
                'accion'     => 'editar',
                'oferta'     => $oferta,
                'formulario' => $formulario->createView()
            )
        );
    }
```

Ahora ya puedes empezar a refactorizar la plantilla `formulario.html.twig` creada en las secciones anteriores:

```
{# app/Resources/views/extranet/oferta.html.twig #}
{# ... #}

{% block title %}{{ accion == 'crear'
    ? 'Añadir una nueva oferta'
    : 'Modificar la oferta ' ~ oferta.nombre
}}{% endblock %}

{# ... #}
```

El cambio más importante en la plantilla está relacionado con la foto. Cuando se crea una oferta, sólo se muestra un campo de formulario para seleccionar el archivo de la foto. Cuando se modifica una oferta, también se muestra una miniatura de la foto actual:

```
{# app/Resources/views/extranet/oferta.html.twig #}
{# ... #}
<div>
    {{ form_label(formulario.foto, 'Fotografía') }}
    {{ form_errors(formulario.foto) }}

    {% if accion == 'editar' %}
        <img src="{{ asset(oferta.rutaFoto, 'ofertas') }}" />

        {{ form_label(formulario.foto, 'Modificar foto') }}
    {% endif %}

    {{ form_widget(formulario.foto) }}
</div>

{# ... #}
```

Las condiciones legales sólo se deben aceptar al crear la oferta, por lo que no es necesario mostrarlas cada vez que se modifica la oferta:

```
{# app/Resources/views/extranet/oferta.html.twig #}
{# ... #}

{% if accion == 'crear' %}
```

```
<div>
    {{ form_errors(formulario.acepto) }}
    {{ form_widget(formulario.acepto) }} <span>Declaro que toda la información d
e esta oferta es correcta, que soy consciente de la obligación de cumplir las co
ndiciones prometidas y que dispongo de los medios necesarios para hacerlo.</spa
n>
</div>
{% endif %}

{# ... #}
```

Para dar por finalizada la acción que modifica ofertas, es necesario realizar un último cambio en el formulario. El campo adicional `acepto` sólo se debe incluir cuando se añade una oferta, no cuando se modifica. Para ello, puedes comprobar el valor de la propiedad `id` de la entidad: si vale `null`, la entidad todavía no se ha guardado en la base de datos y por tanto se está creando:

```
// src/AppBundle/Form/OfertaType.php
class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...

        if (null == $options['data']->getId()) {
            $builder->add('acepto', 'checkbox', array(
                'mapped'   => false,
                'required' => false
            ));

            $listener = new OfertaTypeListener();
            $builder->addEventListener(
                FormEvents::PRE_SUBMIT,
                array($listener, 'preSubmit')
            );
        }
    }
}
```

Sección 4

Backend

Esta página se ha dejado vacía a propósito

CAPÍTULO 16

Planificación

El *backend* es la parte de administración del sitio web. Su acceso está restringido para usuarios del *frontend* y para las tiendas de la *extranet*. Al *backend* sólo pueden acceder un nuevo grupo de usuarios de tipo administrador.

Los administradores pueden ver y modificar cualquier información sobre ofertas, tiendas, usuarios y ciudades. La gestión de la información se divide en función de cada entidad. Para cada una están disponibles las siguientes operaciones:

- Crear (*create*): crea una nueva entidad (equivalente a añadir una fila en la tabla de la base de datos).
- Ver (*read*): muestra todos los datos de una entidad. Se utiliza para ver el detalle de la información sin tener que entrar en el formulario que modifica sus datos.
- Actualizar (*update*): actualiza una o más propiedades de una entidad.
- Borrar (*delete*): borra una entidad completa (equivalente a borrar una fila en la tabla de la base de datos).

Estas cuatro operaciones son tan comunes que es habitual referirse a ellas mediante el acrónimo *CRUD*, formado por las iniciales en inglés de las cuatro operaciones.

16.1 Bundles

Como se explicó al desarrollar la *extranet* el concepto de *bundle* en Symfony es muy flexible, lo que significa que suele haber varias soluciones a una misma necesidad:

1. Crear un único *bundle* llamado `BackendBundle` que incluya todos los controladores, plantillas y rutas de la parte de administración.
2. Crear varios *bundles* de administración, cada uno dedicado exclusivamente a una entidad: `CiudadBackendBundle`, `OfertaBackendBundle`, `TiendaBackendBundle`, etc.
3. Crear la parte de administración de cada entidad dentro de los *bundles* ya existentes. Así por ejemplo dentro de `AppBundle` se puede añadir el controlador `Controller/BackendController.php`.

No obstante, el *backend* de la aplicación *Cupon* se no desarrolla de ninguna de estas maneras. En su lugar, se utiliza un *bundle* de terceros que sigue su propia filosofía de trabajo, tal y como se explicará con detalle en el próximo capítulo.

16.2 Seguridad

El *backend* del sitio web es una zona de acceso restringido a la que sólo pueden acceder los usuarios de tipo administrador. Así que en primer lugar se define un nuevo *rol* llamado `ROLE_ADMIN` para distinguir a estos usuarios de los del *rol* `ROLE_USUARIO` o `ROLE_TIENDA`.

A continuación, se añade un nuevo *firewall* en el archivo `security.yml` y se restringe el acceso al *backend* a los usuarios que sean de tipo `ROLE_ADMIN`:

```
# app/config/security.yml
security:
    firewalls:
        backend:
            pattern:        ^/backend
            provider:       administradores
            http_basic:     ~

        extranet:
            pattern:        ^/extranet
            # ...

        frontend:
            pattern:        ^/*
            # ...

    access_control:
        # ...
        - { path: ^/backend/*, roles: ROLE_ADMIN }

    # ...
```

Recuerda que el orden en el que se definen los *firewalls* es importante, ya que la expresión regular de un *firewall* puede *solaparse* con la expresión regular de los otros *firewalls*. En este caso, el nuevo *firewall* `backend` debe definirse antes que el *firewall* `frontend`.

Los usuarios del *firewall* `backend` se crean mediante un proveedor llamado `administradores` que se definirá más adelante. Los usuarios del *frontend* y de la *extranet* utilizan un formulario de login para introducir sus credenciales. En el caso de los administradores se puede simplificar todavía más este procedimiento utilizando la autenticación básica de HTTP.

Añadiendo la opción `http_basic` en la configuración del *firewall*, se consigue que la aplicación solicite el usuario y contraseña mediante la *caja de login* del propio navegador, en vez de tener que definir un nuevo formulario de login.

Una vez configuradas la autenticación y la autorización, el otro elemento que se debe configurar es el proveedor de usuarios `administradores`. Los usuarios del *frontend* se crean con la entidad `Usuario` y los usuarios de la *extranet* con la entidad `Tienda`. Así que la primera opción sería crear en la aplicación una nueva entidad `Administrador` para crear estos usuarios y guardar su información en la base de datos.

Otra opción más avanzada sería aprovechar la entidad `Usuario` para crear todo tipo de usuarios. Para ello habría que refactorizar su código y añadir al menos una propiedad que indique el tipo de usuario mediante los *roles* definidos en la aplicación.

La última opción consiste en aprovechar las características del componente de seguridad de Symfony para crear los administradores directamente en el archivo `security.yml`, sin crear ni modificar ninguna entidad de Doctrine:

```
# app/config/security.yml
security:
    firewalls:
        # ...

    access_control:
        # ...

    providers:
        # ...
        administradores:
            memory:
                users:
                    admin: { password: 1234, roles: ROLE_ADMIN }

    encoders:
        # ...
        Symfony\Component\Security\Core\User\User: plaintext
```

El proveedor `administradores` no utiliza la opción `entity` para indicar la entidad de la que surgen los usuarios, sino que utiliza la opción `memory` para crear los usuarios directamente en la memoria del servidor. De esta forma se pueden crear tantos usuarios como sean necesarios, cada uno con su propia contraseña y *roles*:

```
# app/config/security.yml
security:
    # ...
    providers:
        # ...
        administradores:
            memory:
                users:
                  admin:  { password: 1234, roles: ROLE_ADMIN }
                  jose:   { password: secreto, roles: ['ROLE_ADMIN','ROLE_MANAGE
R'] }
                  editor: { password: s4jdi8Sp, roles: ['ROLE_ADMIN', 'ROLE_EDIT
OR'] }
```

La contraseña de estos usuarios es directamente la indicada en su opción `password` porque en la configuración anterior se establece que las contraseñas de los usuarios creados por Symfony se guardan en claro (opción `plaintext`):

```
# app/config/security.yml
security:
    # ...

    encoders:
        # ...
        Symfony\Component\Security\Core\User\User: plaintext
```

Guadar en claro las contraseñas de los administradores no es una buena práctica de seguridad. Por eso también puedes codificar estas contraseñas con el algoritmo `bcrypt`:

```
# app/config/security.yml
security:
    # ...

    providers:
        # ...
        administradores:
            memory:
                users:
                    admin: { password: '$2y$13$w700yeSs8FxMLrEXaCATgehosZf9vx09R
bwKddswl9LN/22dKL2qO', roles: ROLE_ADMIN }
                    # ...

    encoders:
        # ...
        Symfony\Component\Security\Core\User\User: bcrypt
```

La contraseña del administrador sigue siendo `1234`, pero ahora es *imposible* de adivinar ni aún accediendo a los contenidos del archivo `security.yml`.

Por último, con los usuarios de tipo administrador puede resultar muy útil la opción `role_hierarchy` que permite definir una jerarquía para los *roles* de la aplicación indicando qué otros *roles* comprende cada *role*. El siguiente ejemplo indica que cualquier usuario con *role* `ROLE_ADMIN` también dispone de los *roles* `ROLE_USUARIO` y `ROLE_TIENDA`:

```
# app/config/security.yml
security:
    firewalls:
        # ...

    access_control:
        # ...

    encoders:
```

```
        # ...

    providers:
        # ...

    role_hierarchy:
        ROLE_ADMIN: [ROLE_TIENDA, ROLE_USUARIO]
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 17

Admin generator

Se denomina *admin generator* al conjunto de utilidades que facilitan la creación de la parte de administración del sitio web. Las versiones antiguas de Symfony incluían un completo *admin generator* o que permitía crear con un solo comando un *backend* de tipo *CRUD* para crear, editar, buscar y borrar cualquier entidad de la base de datos.

A partir de Symfony 2 esta funcionalidad desapareció y en su lugar Symfony propone varias alternativas:

- **Generar el *backend* a mano**, no se aconseja para los *backends* sencillos porque existen soluciones automáticas mejores. Solo sería aconsejable cuando el *backend* es extremadamente complejo o tiene un funcionamiento muy diferente al *CRUD* tradicional.
- SensioGeneratorBundle (https://github.com/sensiolabs/SensioGeneratorBundle) . Se desaconseja su uso porque los *backends* generados son extremadamente simples y apenas ahorra trabajo.
- SonataAdminBundle (https://github.com/sonata-project/SonataAdminBundle) . El generador de *backends* más completo para Symfony, pero muy difícil de aprender porque apenas cuenta con documentación.
- EasyAdminBundle (https://github.com/javiereguiluz/EasyAdminBundle) . Una alternativa menos potente que SonataAdminBundle, pero mucho más fácil de usar y de aprender. Además, está completamente documentado.

En este capítulo se muestran los *bundles* SensioGeneratorBundle y EasyAdminBundle.

17.1 SensioGeneratorBundle

Symfony dispone de un generador de código que simplifica varias tareas comunes durante el desarrollo de las aplicaciones web. A lo largo de los capítulos anteriores se han utilizado varios de sus comandos para generar código: `generate:bundle`, `generate:doctrine:entity`, etc.

Además de estas tareas, Symfony incluye otra que genera código y que está relacionada con los *admin generators*: `generate:doctrine:crud`, que genera todo el código necesario para administrar los datos de una entidad de Doctrine mediante las habituales operaciones *CRUD*.

Aunque no se trata de un *admin generator* completo, esta tarea genera un controlador, un archivo de rutas y cuatro plantillas listas para utilizar con la entidad indicada. Así, para crear un administrador de ofertas, basta con ejecutar el siguiente comando:

```
$ php app/console generate:doctrine:crud --entity=AppBundle:Oferta
  --route-prefix=oferta --with-write --format=yml --no-interaction
```

Las opciones incluidas en el comando son las siguientes:

- `--entity`, indica (con *notación bundle*) la entidad para la que se crea el administrador. En este caso, la entidad `Oferta` del *bundle* `AppBundle`.
- `--route-prefix`, prefijo que se añade a todas las rutas generadas. El prefijo se incluye mediante la opción `prefix` al importar el archivo de rutas generado.
- `--with-write`, si no añades esta opción, sólo se generan las acciones para mostrar información (`list` y `show`). Con esta opción, el comando genera todas las acciones del *CRUD*, incluyendo `create`, `edit` y `delete`.
- `--format`, formato de los archivos de configuración. En concreto es el formato en el que se definen las rutas. Los formatos `yml`, `xml` y `php` definen las rutas en un nuevo archivo. El formato `annotation` define las rutas en el propio controlador, por lo que no se genera un archivo de rutas.
- `--no-interaction`, para que el comando complete todas sus operaciones sin realizar ninguna pregunta por consola.

Si ejecutas el comando anterior, Symfony genera los siguientes archivos y realiza los siguientes cambios:

- Crea el controlador `AppBundle/Controller/OfertaController.php` con todas las acciones del *CRUD* de las ofertas.
- Crea el formulario `AppBundle/Form/OfertaType.php` con tantos campos como propiedades tenga la entidad `Oferta`.
- Crea las plantillas `index.html.twig`, `new.html.twig`, `edit.html.twig` y `show.html.twig` en? el directorio `AppBundle/Resources/views/Oferta/`.
- Crea el archivo de enrutamiento `oferta.yml` en el directorio `AppBundle/Resources/config/routing/`.
- Importa el anterior archivo `oferta.yml` desde el archivo `AppBundle/Resources/config/routing.yml`

Si se produce algún error o el comando no puede completar alguna de sus operaciones, se muestra un mensaje de error muy completo que explica la causa del error y describe con precisión la forma en la que puedes solucionarlo. El siguiente ejemplo es el mensaje de error que muestra el comando cuando no puede importar el archivo de rutas generado:

```
$ php app/console generate:doctrine:crud --entity=AppBundle:Oferta
  --route-prefix=oferta --with-write --format=yml --no-interaction

CRUD generation
```

```
  Generating the CRUD code:  OK
  Generating the Form code:  OK
  Importing the CRUD routes: FAILED

The command was not able to configure everything automatically.
You must do the following changes manually:

  Import the bundle's routing resource in the bundle routing file
  (.../src/AppBundle/Resources/config/routing.yml).

    AppBundle_oferta:
        resource: "@AppBundle/Resources/config/routing/oferta.yml"
        prefix:   /oferta
```

17.1.1 Refactorizando los archivos generados

Después de ejecutar el comando anterior, ya puedes probar la parte de administración generada automáticamente por Symfony para las ofertas. Accede a la URL `http://127.0.0.1:8000/app_dev.php/oferta` para ver el listado de todas las ofertas:

Oferta list

Id	Nombre	Slug	Descripcion	Condiciones	Foto	Precio	Descuento	Fecha_publicacion	Fecha_expiracion	Compras	Umbral	Revisada	Actions
1	Oferta natoque sitamet lorem at nulla vel libero	oferta-natoque-sitamet-lorem-at-nulla-vel-libero-	Nulla scelerisque blandit ligula eget hendrerit. Maecenas non erat eu justo rutrum condimentum...	No disponible para llevar. Debe consumirse en el propio local.	foto14.jpg	94.01	54.53	2011-11-16 23:59:59	2011-11-17 23:59:59	0	75		• show • edit
2	Oferta imperdiet ut imperdiet aut	oferta-imperdiet-ut-imperdiet-aut-	Nam rhoncus lorem sed libero hendrerit accumsan. Sed malesuada, enim sit amet ultricies semper...	Oferta válida si se realizan consumiciones adicionales por valor de 50 euros.	foto20.jpg	69.41	44.42	2011-11-15 23:59:59	2011-11-16 23:59:59	2	56	1	• show • edit
3	Oferta sed eget facilisis et tincidunt sitamet malesuada	oferta-sed-eget-facilisis-et-tincidunt-sitamet-malesuada-	Mauris ultricies nunc nec sapien tincidunt facilisis. Pellentesque ultricies erat ac lorem pharetra vulputate...	Válido para cualquier día entre semana.	foto11.jpg	88.33	20.32	2011-11-14 23:59:59	2011-11-15 23:59:59	1	88	1	• show • edit
4	Oferta imperdiet lacinia sitamet enimsit facilisis	oferta-imperdiet-lacinia-sitamet-enimsit-facilisis-	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris ultricies nunc nec sapien tincidunt facilisis...	Válido para cualquier día entre semana. Válido solamente para comidas, no para cenas.	foto2.jpg	26.21	14.15	2011-11-13 23:59:59	2011-11-14 23:59:59	0	58		• show • edit

Figura 17.1 Aspecto de la portada de la parte de administración generada por Symfony

El listado permite ver los detalles de cualquier oferta pinchando en el enlace de su clave primaria o en el enlace *show*. También permite modificar los datos de cualquier oferta pinchando en el enlace *edit*. Por último, puedes crear nuevas ofertas pinchando el enlace *Create a new entry* que se muestra al final del listado.

El primer problema de las plantillas generadas es que su aspecto es bastante feo por no disponer de ningún estilo. No obstante, gracias a la flexibilidad de Twig, puedes mejorar fácilmente su aspecto haciendo que herede de alguna plantilla de la aplicación.

El siguiente problema de las plantillas generadas es que los listados incluyen todas las propiedades de cada entidad, lo que los hace inmanejables. En el caso concreto del listado de ofertas, basta con incluir la información esencial de cada oferta. Así que puedes simplificar la plantilla generada sustituyendo su código por lo siguiente:

```
{# src/AppBundle/Resources/views/Oferta/index.html.twig #}
{# ... #}

{% block article %}
<h1>Oferta list</h1>

<table class="records_list">
    <thead>
        <tr>
            <th>ID</th>
            <th>Nombre</th>
            <th>Precio</th>
            <th>Descuento</th>
            <th>Fecha_publicacion</th>
            <th>Fecha_expiracion</th>
            <th>Compras</th>
            <th>Umbral</th>
            <th>Revisada</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
    {% for entity in entities %}
        <tr>
            <td><a href="{{ path('backend_oferta_show', { 'id': entity.id })
}}">{{ entity.id }}</a></td>
            <td>{{ entity.nombre }}</td>
            <td>{{ entity.precio }}</td>
            <td>{{ entity.descuento }}</td>
            <td>
                {% if entity.fechapublicacion %}
                    {{ entity.fechapublicacion|date('Y-m-d H:i:s') }}
                {% endif%}
            </td>
            <td>
                {% if entity.fechaexpiracion %}
                    {{ entity.fechaexpiracion|date('Y-m-d H:i:s') }}
                {% endif%}
            </td>
            <td>{{ entity.compras }}</td>
            <td>{{ entity.umbral }}</td>
            <td>{{ entity.revisada }}</td>
            <td>
                <ul>
                    <li><a href="{{ path('backend_oferta_show',
                                { id: entity.id }) }}">show</a></li>
                    <li><a href="{{ path('backend_oferta_edit',
                                { id: entity.id }) }}">edit</a></li>
```

```
                </ul>
            </td>
        </tr>
    {% endfor %}
    </tbody>
</table>

<ul>
    <li><a href="{{ path('backend_oferta_new') }}">Create a new entry</a></li>
</ul>
{% endblock %}
```

Al margen de los retoques estéticos y de los ajustes en sus contenidos, las plantillas generadas sufren carencias básicas para la parte de administración de un sitio:

- No se incluye un paginador, ni siquiera en los listados con cientos o miles de filas.
- No se pueden reordenar los listados pinchando en el título de cada columna.
- No se incluyen filtros que permitan restringir los listados o realizar búsquedas entre los contenidos.
- No se contempla la posibilidad de que existan campos especiales en los formularios, como campos de contraseña, de fotos o archivos, etc.

17.2 EasyAdminBundle

Esta sección se completará próximamente.

Esta página se ha dejado vacía a propósito

CAPÍTULO 18

Comandos de consola

Los usuarios registrados en la aplicación *Cupon* reciben a diario un email con la información sobre la *oferta del día* en su ciudad. Si no desean recibir el email, pueden indicarlo en su perfil gracias a la propiedad `permiteEmail` de la entidad `Usuario`.

La mejor forma de desarrollar esta funcionalidad es mediante una tarea programada. Suponiendo que el tráfico del sitio web disminuya por las noches, esta tarea podría ejecutarse de madrugada para que los usuarios reciban los emails a primera hora de la mañana.

Este tipo de tareas se realizan en Symfony mediante **comandos de consola**. Los comandos son *scripts* PHP que se ejecutan en la consola y que permiten automatizar tareas pesadas, de mantenimiento o que simplemente se deben ejecutar con una determinada periodicidad.

18.1 Creando comandos de consola

Como ya se ha visto en los capítulos anteriores, Symfony incluye decenas de comandos que se pueden listar ejecutando `php app/console`. Los comandos propios de tu aplicación se crean con las mismas herramientas que los comandos estándar y Symfony los trata exactamente igual que a sus comandos.

Por convención, los comandos se definen en clases PHP cuyo nombre acaba en `Command` y se guardan en el directorio `Command/` del *bundle*. Así que para desarrollar el comando que envía el email diario, crea un archivo llamado `EmailOfertaDelDiaCommand.php` en el directorio `src/AppBundle/Command/` y con el siguiente contenido:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
namespace AppBundle\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class EmailOfertaDelDiaCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('app:email:oferta-del-dia')
            ->setDefinition(array())
            ->setDescription('Genera y envía a cada usuario el email con la
                              oferta diaria')
```

```
            ->setHelp('...')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $output->writeln('Generando emails...');

        // ...
    }
}
```

Los comandos de consola de Symfony heredan de la clase base `Command` y deben incluir al menos los métodos `configure()` y `execute()`. El método `configure()` define toda la información básica del comando:

- `setName()`, establece el nombre del comando, que es exactamente lo que hay que escribir para ejecutar el comando. En este caso: `php app/console app:email:oferta-del-dia`.
- `setDefinition()`, define los argumentos y las opciones que permite y/o requiere el comando, tal y como se explica más adelante.
- `setDescription()`, establece el mensaje breve de ayuda que se muestra al lado de cada comando en el listado que aparece al ejecutar `php app/console`.
- `setHelp()`, establece el mensaje completo de ayuda que se muestra cuando se solicita explícitamente la ayuda del comando con `php app/console help nombre-del-comando`

Aunque no es obligatorio hacerlo, resulta habitual dividir el nombre del comando en varias partes separadas por el carácter `:`. La primera parte (ej. `app:`) permite diferenciar fácilmente tus comandos del resto de comandos de Symfony; la segunda parte (ej. `email:`) permite agrupar todos los comandos relacionados (`doctrine:`, `generate:`, etc.); la tercera parte (ej. `oferta-del-dia`) se considera el nombre específico del comando dentro de ese grupo de comandos.

Por su parte, el método `execute()` contiene todo el código que ejecuta el comando. A este método se le pasa un objeto `$input`, a través del cual se obtiene el valor de las opciones y argumentos indicados por el usuario, y un objeto `$output`, a través del cual se muestra información en la consola.

Si ejecutas ahora el comando `php app/console` después de añadir la clase `EmailOfertaDelDiaCommand`, verás en el listado un nuevo comando llamado `app:email:oferta-del-dia` junto al resto de comandos internos de Symfony:

```
$ php app/console

Available commands:
  help                        Displays help for a command
  list                        Lists commands
  ...
app
```

```
  app:email:oferta-del-dia    Genera y envía a cada usuario ...
...
```

18.1.1 Opciones y argumentos

Los comandos pueden variar su comportamiento mediante las opciones y argumentos indicados por el usuario. Si por ejemplo la aplicación *Cupon* tuviese miles de usuarios registrados, sería una locura generar todos los emails a la vez. Una alternativa más lógica sería enviar los emails por ciudades, indicando para ello el *slug* de la ciudad al ejecutar el comando:

```
$ php app/console app:email:oferta-del-dia sevilla
```

Este tipo de parámetros se llaman **argumentos** y pueden ser obligatorios u opcionales. Cada comando puede definir tantos argumentos como desee mediante la clase `InputArgument` dentro del método `setDefinition()`:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php

// ...
use Symfony\Component\Console\Input\InputArgument;

protected function configure()
{
    $this
        ->setName('app:email:oferta-del-dia')
        ->setDefinition(array(
            new InputArgument(
                'ciudad',
                InputArgument::OPTIONAL,
                'El slug de la ciudad para la que se generan los emails'
            ),
        ))
        // ...
}
```

El constructor de la clase `InputArgument` admite los siguientes parámetros con los siguientes valores por defecto:

```
InputArgument($nombre, $tipo = null, $descripcion = '', $por_defecto = null)
```

- `$nombre`, es el nombre del argumento, que debe ser único para un mismo comando. Aunque el usuario no escribe este nombre al ejecutar el comando, se utiliza dentro del código para acceder al valor del argumento.
- `$tipo`, indica el tipo de argumento y su valor sólo puede ser:
    - `InputArgument::REQUIRED`, el argumento es obligatorio y por tanto, siempre hay que indicarlo cuando se ejecuta el comando.

    - `InputArgument::OPTIONAL`, el argumento es opcional y no es necesario añadirlo al ejecutar el comando. Si el comando tiene dos o más argumentos opcionales, para indicar el segundo argumento es obligatorio indicar también el primero. El motivo es que nunca se incluye el nombre de los argumentos, por lo que sólo importa el orden en el que se indican.
    - `InputArgument::IS_ARRAY`, este tipo de argumento prácticamente no se utiliza. Permite capturar todos los argumentos pasados al comando como si fueran un único argumento en forma de array PHP.

- `$descripcion`, es la descripción del argumento que se muestra cuando se consulta la ayuda del comando mediante `php app/console help nombre-del-comando`.
- `$por_defecto`, es el valor por defecto del argumento.

Los comandos también pueden variar su comportamiento mediante las **opciones**. A diferencia de los argumentos, estas siempre se indican con su nombre precedido de dos guiones medios (`--opcion=valor`).

Siguiendo con el mismo comando del ejemplo anterior, cuando la aplicación tenga decenas de miles de usuarios registrados, puede ser interesante dividir el comando en dos partes: generar los emails (personalizados para cada usuario) y enviarlos. Esto se puede conseguir por ejemplo con una opción llamada `accion` que admita los valores `generar` y `enviar`:

```
$ php app/console app:email:oferta-del-dia --accion=generar
```

Las opciones también pueden ser obligatorias u opcionales y se definen mediante la clase `InputOption` dentro del método `setDefinition()` del comando:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php

// ...
use Symfony\Component\Console\Input\InputOption;

protected function configure()
{
    $this
        ->setName('app:email:oferta-del-dia')
        ->setDefinition(array(
            new InputOption(
                'accion',
                null,
                InputOption::VALUE_OPTIONAL,
                'Indica si los emails sólo se generan o también se envían',
                'enviar'
            ),
        ))
        // ...
}
```

El constructor de la clase `InputOption` admite los siguientes argumentos con los siguientes valores por defecto:

```
InputOption($nombre, $atajo = null, $tipo = null, $descripcion = '', $por_defecto = null)
```

- `$nombre`, es el nombre único de la opción dentro del comando. Para añadir una opción al ejecutar el comando, es obligatorio incluir este nombre precedido por dos guiones medios: `--nombre=valor`.
- `$atajo`, es el nombre corto de la opción, que permite ejecutar el comando más ágilmente. Si por ejemplo defines una opción con el nombre `version`, un atajo adecuado sería `v`. Los atajos se indican con un solo guión medio (`-v`) en vez de los dos guiones del nombre (`--version`).
- `$tipo`, indica el tipo de opción y, aunque solamente suelen utilizarse los dos primeros tipos, existen cuatro tipos disponibles:
  - `InputOption::VALUE_REQUIRED`, es obligatorio indicar un valor cuando se utiliza la opción.
  - `InputOption::VALUE_OPTIONAL`, no es obligatorio, pero sí es posible, indicar un valor cuando se utiliza la opción.
  - `InputOption::VALUE_NONE`, la opción no permite indicar ningún valor, sólo su nombre. Un buen ejemplo es la opción `--no-warmup` del comando `cache:clear`.
  - `InputOption::VALUE_IS_ARRAY`, se puede indicar más de un valor repitiendo varias veces la misma opción. Un buen ejemplo es la opción `--fixtures` del comando `doctrine:fixtures:load` que permite indicar varios archivos de datos: `$php app/console doctrine:fixtures:load --fixtures=archivo1 --fixtures=archivo2 --fixtures=archivo3`
- `$descripcion`, es la descripción de la opción que se muestra cuando se consulta la ayuda del comando mediante `php app/console help nombre-del-comando`.
- `$por_defecto`, es el valor por defecto de la opción. Se puede indicar en todos los tipos de opciones salvo en `InputOption::VALUE_NONE`.

Aunque no definas opciones en tus comandos, la aplicación `app/console` de Symfony incluye automáticamente las siguientes opciones a todos tus comandos:

```
Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v Increase verbosity of messages.
  --version        -V Display this program version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
```

```
  --shell          -s Launch the shell.
  --env            -e The Environment name.
  --no-debug          Switches off debug mode.
```

Las tres opciones más interesantes son:

- `--env`, permite ejecutar una tarea en el entorno de ejecución indicado, por lo que es una **opción imprescindible** en aplicaciones web reales. Así, cuando ejecutes un comando en producción, no olvides añadir siempre la opción `--env=prod`
- `--quiet`, indica que el comando no debe mostrar por pantalla ningún mensaje ni ninguna otra información.
- `--no-interaction`, impide que el comando pida al usuario que conteste preguntas, tome decisiones o introduzca valores. Esta opción es ideal para los comandos que se ejecutan mediante tareas programadas del sistema operativo.

Después de añadir el argumento y la opción, el método `setDefinition()` completo es el que se muestra a continuación:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php

// ...
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

protected function configure()
{
  $this
      ->setName('app:email:oferta-del-dia')
      ->setDefinition(array(
          new InputArgument('ciudad', InputArgument::OPTIONAL,
              'El slug de la ciudad para la que se generan los emails'),
          new InputOption('accion', null, InputOption::VALUE_OPTIONAL,
              'Indica si los emails sólo se generan o también se envían', 'envia
r'),
      ))
      // ...
}
```

Como tanto el argumento como la opción son opcionales, puedes combinarlos como quieras al ejecutar el comando:

```
$ php app/console app:email:oferta-del-dia sevilla --accion=generar
$ php app/console app:email:oferta-del-dia --accion=enviar sevilla
$ php app/console app:email:oferta-del-dia --accion=generar
$ php app/console app:email:oferta-del-dia
```

Para obtener el valor de los argumentos y de las opciones dentro del método `execute()` del comando, utiliza los métodos `getOption()` y `getArgument()`:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $ciudad = $input->getArgument('ciudad');
        $accion = $input->getOption('accion');

        // ...
    }
}
```

18.1.2 Interactuando con el usuario

Durante su ejecución, los comandos pueden interactuar con el usuario mostrando mensajes informativos o realizando preguntas para que el usuario tome decisiones. A partir de la versión 2.8, Symfony incluye una serie de métodos que estandarizan la forma en la que se muestra y se pide información. De esta manera, el aspecto de tus comandos será homogeneo y no te supondrá ningún esfuerzo.

Para utilizar estos métodos, debes instanciar la clase `SymfonyStyle` de la siguiente manera (es común llamar `$io` a la variable porque se encarga tanto del *input* como del *output* del comando):

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

class EmailOfertaDelDiaCommand extends Command
{
    protected function configure()
    {
        // ...
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);
        // ...
    }
}
```

18.1.2.1 Mostrando mensajes informativos

Los mensajes informativos se muestran con los siguientes métodos:

- `title(string $texto)`, muestra el título principal del comando
- `section(string $texto)`, muestra un subtítulo, ideal para dividir secciones dentro de un comando largo.
- `text(string|array $texto)`, muestra un texto normal sin ningún formato. Puedes pasar tanto una cadena de texto como un array de cadenas.
- `comment(string|array $texto)`, muestra un texto formateado como menos importante que el texto normal (admite tanto cadenas como arrays de cadenas).
- `listing(array $elementos)`, muestra el array de cadenas de texto que se le pasa en forma de lista de elementos.
- `table(array $cabeceras, array $filas)`, muestra una tabla con la información que se le pasa sobre sus filas y cabeceras.

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Component\Console\Style\SymfonyStyle;

class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        $io->title('Email Oferta del Día');
        $io->text('Comienza el proceso de generación de emails...');

        // ...

        $io->comment(array(
            'Generados 10 emails',
            'Comienza el envío de los mensajes',
            'Conectando con el servidor de correo...'
        ));

        // ...
    }
}
```

El aspecto de los mensajes también se puede modificar para que muestre diferentes colores de letra y de fondo. La forma más sencilla de conseguirlo es mediante los *formateadores* `<info>` y `<comment>` que incluye Symfony:

```
$io->comment(array(
    'Generados <info>10</info> emails',
    '<comment>Comienza el envío</comment> de los mensajes',
    '<info>Conectando</info> con el <comment>servidor de correo</comment>...'
));
```

Symfony muestra cada *formateador* con el siguiente aspecto:

Nombre	Color de letra	Color de fondo
`comment`	amarillo	*(transparente)*
`error`	blanco	rojo
`info`	verde	*(transparente)*
`question`	negro	cyan *(azul muy claro)*

Si ninguno de los *formateadores* predefinidos se ajusta a tus necesidades, puedes crear tus propios estilos fácilmente indicando el color de letra y de fondo para cada mensaje:

```
$io->text('Generados <fg=magenta;bg=yellow>10</> emails');
$io->text('Generados <fg=red;option=underscore>10</> emails');
$io->text('Generados <bg=blue>10</> emails');
```

El color del texto se indica mediante `fg` (del inglés *foreground*) y el color de fondo mediante `bg` (del inglés *background*). Los colores disponibles son `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`. Las opciones que modifican las características del texto se indican mediante `option` y los valores permitidos son `bold` (en negrita), `underscore` (subrayado), `blink` (parpadeante), `reverse` (en negativo, con los colores invertidos) y `conceal` (oculto, no se muestra, sólo sirve para cuando el usuario introduce información que no se quiere mostrar por pantalla).

NOTA Dependiendo de tu sistema operativo y de la aplicación que utilices para la consola de comandos, puede que veas todos, algunos o ninguno de los colores y estilos soportados por Symfony.

No pienses que modificar los colores de los mensajes importantes es una pérdida de tiempo. Si en una consola se muestran decenas de mensajes, resulta esencial resaltar por ejemplo los mensajes de error con un color de fondo rojo, para detectarlos cuanto antes.

18.1.2.2 Mostrando el resultado del comando

Resulta esencial que la ejecución del comando indique claramente al usuario su resultado: éxito o error. Para ello, Symfony proporciona los siguientes métodos:

- `success(string|array $message)`
- `error(string|array $message)`
- `warning(string|array $message)`

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Component\Console\Style\SymfonyStyle;

class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        // ...

        if ($emailsEnviados === true) {
            $io->success(sprintf('%d emails enviados correctamente.', $numEmail
s));
        } else {
            $io->error('Los emails no se han enviado correctamente.');
        }
    }
}
```

18.1.2.3 Preguntando al usuario

Además de los argumentos y las opciones, los comandos pueden solicitar más información a los usuarios durante su ejecución. Esto es útil por ejemplo para confirmar que el usuario quiere realizar una determinada acción.

Para ello, Symfony proporciona los siguientes métodos:

- `ask(string $pregunta, string|null $valorPorDefecto = null, callable|null $validador = null)`, el método más utilizado para pedir información al usuario.
- `askHidden(string $pregunta, callable|null $validador = null)`, similar al anterior, pero la respuesta del usuario no se muestra por pantalla.
- `confirm(string $pregunta, bool $valorPorDefecto = true)`, se emplea para que el usuario confirme (pulsando `y`) o rechace (pulsando `n`) el mensaje mostrado.
- `choice(string $pregunta, array $choices, string|int|null $valorPorDefecto = null)`, permite pedir información al usuario pero restringiendo las respuestas a los valores definidos en este método.

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Component\Console\Style\SymfonyStyle;

class EmailOfertaDelDiaCommand extends Command
{
    // ...
```

```
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);
        $ciudad = $io->ask('¿Para qué ciudad quieres generar los emails?', 'sevi
lla');

        // ...

        $respuesta = $io->confirm('¿Quieres enviar ahora todos los emails?',
'n');

        // ...

        $clave = $io->askHidden('Introduce la clave numérica', function ($valo
r) {
            if (false == is_numeric($valor)) {
                throw new \InvalidArgumentException('El valor introducido no es
un número');
            }

            return $valor;
        });
    }
}
```

Una buena práctica relacionada con las preguntas al usuario consiste en agruparlas todas bajo el método `interact()` del comando, que se ejecuta siempre que el comando no incluya la opción `--no-interaction`. Además, el método `interact()` se ejecuta antes que el método `execute()`, por lo que es ideal para completar o modificar las opciones y argumentos mediante preguntas al usuario:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Component\Console\Style\SymfonyStyle;

class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function interact(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        $io->title('Bienvenido al generador de emails');
        $io->text('Para continuar, debes contestar a varias preguntas...');

        $ciudad = $io->ask('¿Para qué ciudad quieres generar los emails?', 'sevi
lla');
```

```
        $input->setArgument('ciudad', $ciudad);

        $accion = $io->choice('¿Qué quieres hacer con los emails?', array('envia
r', 'generar'), 0);
        $input->setOption('accion', $accion);

        if (!$io->confirm(sprintf('¿Quieres %s ahora los emails de %s?', $accio
n, $ciudad))) {
            // ...
        }
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        // ...
    }
}
```

18.2 Generando la *newsletter* de cada usuario

La primera parte del comando que se está desarrollando consiste en generar la *newsletter* de cada usuario que quiera recibirla. Para ello se obtiene en primer lugar el listado de usuarios y la *oferta del día* de todas las ciudades.

Como se realizan consultas a la base de datos, el comando debe tener acceso al contenedor de inyección de dependencias. Por eso el comando hereda de la clase `ContainerAwareCommand`, que añade un método para poder obtener el contenedor con la instrucción `$this->getContainer();` (tal como se explica en el apéndice B (página 505)):

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;

class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        $em = $this->getContainer()->get('doctrine')->getManager();

        // Obtener el listado de usuarios que permiten el envío de email
        $usuarios = $em->getRepository('AppBundle:Usuario')
                       ->findBy(array('permiteEmail' => true));

        $io->comment(sprintf(
            'Se van a enviar <info>%s</info> emails', count($usuarios)
```

```
        ));

        // Buscar la 'oferta del día' en todas las ciudades de la aplicación
        $ofertas = array();
        $ciudades = $em->getRepository('AppBundle:Ciudad')->findAll();
        foreach ($ciudades as $ciudad) {
            $id   = $ciudad->getId();
            $slug = $ciudad->getSlug();

            $ofertas[$id] = $em->getRepository('AppBundle:Oferta')
                               ->findOfertaDelDiaSiguiente($slug);
        }

        // ...
    }
}
```

Una vez obtenido el *entity manager* a través del contenedor, las consultas a la base de datos son muy sencillas con los métodos `findBy()` y `findAll()`. Para simplificar el código, se define una consulta propia llamada `findOfertaDelDiaSiguiente` en el repositorio de la entidad `Oferta`.

Como el comando se ejecuta todas las noches, no se puede buscar la *oferta del día* de hoy, sino la *oferta del día* que se publicará al día siguiente:

```
// src/AppBundle/Repository/OfertaRepository.php
class OfertaRepository extends EntityRepository
{
    // ...

    public function findOfertaDelDiaSiguiente($ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, c, t
              FROM AppBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t
             WHERE o.revisada = true
               AND o.fechaPublicacion < :fecha
               AND c.slug = :ciudad
          ORDER BY o.fechaPublicacion DESC');
        $consulta->setParameter('fecha', new \DateTime('tomorrow'));
        $consulta->setParameter('ciudad', $ciudad);
        $consulta->setMaxResults(1);

        return $consulta->getSingleResult();
    }
}
```

18.2.1 Renderizando plantillas

El contenido del email que se envía a cada usuario se genera a partir de una plantilla Twig. Para renderizar plantillas dentro de un comando, no puedes utilizar el mismo atajo `render()` que en los controladores, sino que tienes que llamar al método `render()` del servicio `twig` que obtienes a partir del contenedor:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $contenido = $this->getContainer()->get('twig')->render(
            'email/oferta_del_dia.html.twig',
            array('ciudad' => $ciudad, 'oferta' => $oferta, ...)
        );

        // ...
    }
}
```

El email que se envía a cada usuario incluye la información básica de la *oferta del día* de su ciudad, un mensaje con el nombre del usuario instándole a comprar y un enlace informándole sobre cómo darse de baja de la newsletter.

Crea una nueva plantilla llamada `oferta_del_dia.html.twig` en `app/Resources/views/email/` y copia el siguiente código:

```
{# app/Resources/views/email/oferta_del_dia.html.twig #}

<html>
<head></head>
<body>
    <table>
        <tr>
            <td><a href="{{ host ~ path('portada') }}">CUPON</a></td>
            <td>Oferta del día en <span>{{ ciudad.nombre }}</span></td>
        </tr>
        <tr>
            <td>
                <a href="{{ url('oferta_detalle', { slug: oferta.slug }) }}">{{
oferta.nombre }}</a>
                {{ oferta.descripcion | mostrar_como_lista }}
            </td>
            <td>
                <img src="{{ host ~ asset(oferta.rutaFoto, 'ofertas') }}" />
            </td>
```

```
        </tr>
        <tr>
            <td>
                <a href="{{ host ~ path('oferta_detalle', { slug: oferta.slug
}) }}">Ver oferta</a>
            </td>
            <td>
                {{ oferta.precio }} &euro; <strong>{{ descuento(oferta.precio, o
ferta.descuento) }}</strong>
            </td>
        </tr>

        <tr>
            <td colspan="2"><strong> Date prisa {{ usuario.nombre }}!</strong>
             Esta oferta caduca el {{ oferta.fechaExpiracion|date }}</td>
        </tr>

        <tr>
            <td colspan="2">
                &copy; {{ 'now'|date('Y') }} - Has recibido este email porque es
tás suscrito al servicio de envío de <em>"La oferta del día"</em>. Para darte d
e baja de este servicio, accede a <a href="{{ host ~ path('usuario_perfil')
}}">tu perfil</a> y desactiva la opción de envío de emails.
            </td>
        </tr>
</body>
</html>
```

Debido al caótico soporte de los estándares HTML y CSS en los diferentes clientes de correo electrónico, la plantilla se crea con una tabla HTML y cada elemento define sus propios estilos CSS. Por razones de legibilidad y espacio disponible, el código anterior no muestra los estilos CSS necesarios, pero puedes verlos en la plantilla del repositorio: <https://github.com/javiereguiluz /Cupon/blob/2.8/app/Resources/views/email/oferta_del_dia.html.twig>).

Después de definir la plantilla, el comando ya puede crear los emails *renderizando* esta plantilla con las variables adecuadas para cada usuario:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        // ...
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $host = 'dev' == $input->getOption('env') ?
            'http://127.0.0.1:8000' :
```

```
            'http://cupon.com';

        $accion = $input->getOption('accion');

        $em = $this->getContainer()->get('doctrine')->getManager();

        $usuarios = $em->getRepository('AppBundle:Usuario')->findBy(...);

        // Buscar la 'oferta del día' en todas las ciudades de la aplicación
        $ofertas = array();
        $ciudades = $em->getRepository('AppBundle:Ciudad')->findAll();
        foreach ($ciudades as $ciudad) {
            $id   = $ciudad->getId();
            $slug = $ciudad->getSlug();

            $ofertas[$id] = $em->getRepository('AppBundle:Oferta')
                               ->findOfertaDelDiaSiguiente($slug);
        }

        // Generar el email personalizado de cada usuario
        foreach ($usuarios as $usuario) {
            $ciudad = $usuario->getCiudad();
            $oferta = $ofertas[$ciudad->getId()];

            $contenido = $contenedor->get('twig')->render(
                'email/oferta_del_dia.html.twig',
                array('host'    => $host,
                      'ciudad'  => $ciudad,
                      'oferta'  => $oferta,
                      'usuario' => $usuario)
            );

            // Enviar el email ...
        }
    }
}
```

18.3 Enviando la newsletter

SwiftMailer (http://swiftmailer.org/) es una de las mejores y más completas librerías de PHP para enviar emails. Hace unos años el proyecto estuvo a punto de ser abandonado, pero Symfony lo *adoptó* y desde entonces se encarga de mantenerlo. Por ello Symfony dispone de una integración excelente con SwiftMailer.

El envío de emails es un requerimiento tan habitual en las aplicaciones web que SwiftMailer se encuentra activado por defecto en Symfony. Si lo has deshabilitado manualmente, vuelve a activarlo añadiendo la siguiente línea en el archivo de configuración `app/AppKernel.php`:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

Para enviar un email, crea una instancia de la clase `Swift_Message` y utiliza sus métodos para indicar las diferentes partes del mensaje. Una vez creado el objeto que representa el mensaje, envía el email con el método `send()` del servicio `mailer`:

```
// src/AppBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $contenedor = $this->getContainer();

        // ...

        $mensaje = \Swift_Message::newInstance()
            ->setSubject('Oferta del día')
            ->setFrom('mailing@cupon.com')
            ->setTo('usuario1@localhost')
            ->setBody($contenido)
        ;

        $this->contenedor->get('mailer')->send($mensaje);
    }
}
```

El método `setFrom()` con el que se establece la dirección del remitente también permite incluir su nombre y no sólo la dirección de correo electrónico:

```
$mensaje = \Swift_Message::newInstance()
    ->setFrom('mailing@cupon.com')
    ...

$mensaje = \Swift_Message::newInstance()
    ->setFrom(array('mailing@cupon.com' => 'Cupon - Oferta del día'))
    ...
```

Si el contenido del email es de tipo HTML, conviene indicarlo explícitamente:

```
$mensaje = \Swift_Message::newInstance()
    ->setBody($contenido)
    ...

$mensaje = \Swift_Message::newInstance()
    ->setBody($contenido, 'text/html')
    ...
```

Una buena práctica más recomendable consiste en incluir en el mismo mensaje la versión de texto y la versión HTML del contenido:

```
$texto = $contenedor->get('twig')->render(
    'email/oferta_del_dia.txt.twig',
    array( ... )
);

$html  = $contenedor->get('twig')->render(
    'email/oferta_del_dia.html.twig',
    array( ... )
);

$mensaje = \Swift_Message::newInstance()
    ->setBody($texto)
    ->addPart($html, 'text/html')
    ...
```

Por último, también puedes adjuntar fácilmente cualquier tipo de archivo:

```
$documento = $this->getContainer()->getParameter('kernel.root_dir')
             .'/../web/uploads/documentos/promocion.pdf';

$mensaje = \Swift_Message::newInstance()
    ->attach(\Swift_Attachment::fromPath($documento))
    ->...
```

18.3.1 Configurando el envío de emails

Antes de enviar los emails es necesario configurar correctamente el servicio `mailer`. De esta forma la aplicación sabrá por ejemplo qué servidor de correo utilizar para enviar los mensajes. La configuración se realiza en el archivo `app/config/config.yml`:

```
# app/config/config.yml

# ...
swiftmailer:
    transport: %mailer_transport%
    host:      %mailer_host%
    username:  %mailer_user%
    password:  %mailer_password%
```

A su vez, el archivo anterior utiliza los valores establecidos en el archivo `app/config/parameters.yml`:

```
# app/config/parameters.yml

parameters:
    # ...

    mailer_transport:  smtp
    mailer_host:       localhost
    mailer_user:       ~
    mailer_password:   ~
```

La configuración por defecto del `mailer` supone que está disponible un servidor SMTP en el mismo equipo en el que se está ejecutando la aplicación. A continuación se muestra una configuración más avanzada:

```
# app/config/config.yml

# ...
swiftmailer:
    transport:  smtp
    host:       smtp.ejemplo.org
    port:       587
    encryption: ssl
    auth_mode:  login
    username:   tu-login
    password:   tu-contraseña
```

Las opciones disponibles y sus posibles valores son los siguientes:

- `transport`, indica el tipo de transporte utilizado para enviar los mensajes. Los valores permitidos son:
    - `smtp`, se emplea para conectarse a cualquier servidor de correo local o remoto mediante el protocolo SMTP.
    - `mail`, hace uso de la función `mail()` de PHP y por tanto, es el menos fiable de todos.
    - `sendmail`, utiliza el servidor de correo disponible localmente en el servidor en el que se está ejecutando la aplicación Symfony. A pesar de su nombre, funciona con *Sendmail*, *Postfix*, *Exim* y cualquier otro servidor compatible con *Sendmail*.
    - `gmail`, utiliza los servidores de correo electrónico de Google.
- `host`, es el nombre del host o dirección IP del servidor de correo. Por defecto es `localhost`.
- `port`, puerto en el que está escuchando el servidor de correo. Por defecto es `25`.
- `encryption`, indica el tipo de encriptación utilizada para comunicarse con el servidor. Los dos valores permitidos son `ssl` y `tls`. Para que funcionen correctamente, la versión de PHP

debe soportarlos a través de OpenSSL. Esto se puede comprobar fácilmente con una llamada a la función `stream_get_transports()`.

- `auth_mode`, es la forma en la que el mailer se autentica ante el servidor de correo. Los valores permitidos son `plain`, `login` y `cram-md5`. El valor más común es `login` que utiliza un nombre de usuario y contraseña.
- `username`, nombre de usuario utilizado para conectarse al servidor mediante la autenticación de tipo `login`.
- `password`, contraseña utilizada para conectarse al servidor mediante la autenticación de tipo `login`.

18.3.2 Enviando emails en desarrollo

Cuando se está desarrollando la aplicación, enviar emails es una de las tareas más molestas. Por una parte, configurar bien un servidor de correo es algo costoso y crear decenas de cuentas de correo para probar que los mensajes llegan bien también es tedioso. Pero por otra parte, es imprescindible probar que los emails se envían bien y que el destinatario visualiza correctamente su contenido.

Seguramente, la forma más sencilla de probar en desarrollo el envío de emails consiste en utilizar Gmail. Para ello, sólo debes indicar el valor `gmail` como tipo de transporte y añadir tu dirección de correo y contraseña:

```
# app/config/config_dev.yml

# ...
swiftmailer:
    transport: gmail
    username:  tu-direccion@gmail.com
    password:  tu-contraseña
```

Observa que la configuración anterior se ha añadido en el archivo `app/config/config_dev.yml`. De esta forma sólo se tiene en cuenta cuando la aplicación (o el comando) se ejecute en el entorno de desarrollo. Symfony también permite deshabilitar el envío de los mensajes, para poder probar todo el proceso sin tener que enviar realmente los emails. Para ello sólo hay que asignar el valor `true` a la opción `disable_delivery`:

```
# app/config/config_dev.yml

# ...
swiftmailer:
    transport:        gmail
    username:         tu-direccion@gmail.com
    password:         tu-contraseña
    disable_delivery: true
```

Por último, para probar el envío de los emails sin tener que configurar decenas de cuentas de correo, puedes utilizar la opción `delivery_address` indicando una dirección de correo a la que se enviarán todos los emails, independientemente del destinatario real del mensaje:

```
# app/config/config_dev.yml

# ...
swiftmailer:
    transport:        gmail
    username:         tu-direccion@gmail.com
    password:         tu-contraseña
    delivery_address: cuenta-de-prueba@ejemplo.org
```

18.3.3 Enviando emails en producción

Cuando la aplicación se ejecuta en producción el problema de los emails es que cuesta mucho tiempo enviarlos. Si por ejemplo envías un email dentro del controlador, el tiempo de espera hasta que el mensaje se ha enviado puede ser inaceptable desde el punto de vista del usuario.

Por eso en producción es habitual utilizar la técnica del *spooling*, que guarda todos los emails a enviar en un archivo llamado *spool*. Después, otro proceso se encarga de enviar todos los mensajes del *spool*, normalmente por lotes para mejorar el rendimiento. Symfony también soporta esta técnica y por eso sólo es necesario indicarle la ruta del archivo que hará de *spool*:

```
# app/config/config_prod.yml

# ...
swiftmailer:
    transport:  smtp
    spool:
        type:   file
        path:   %kernel.cache_dir%/swiftmailer/spool
```

Observa que la configuración anterior se realiza en el archivo `app/config/config_prod.yml` para que sólo se tenga en cuenta cuando la aplicación o el comando se ejecutan en el entorno de producción. Para indicar la ruta hasta el archivo *spool* es recomendable utilizar rutas relativas mediante los parámetros `%kernel.root_dir%` (directorio raíz de la aplicación no del proyecto, su valor normalmente es `/app`) y `%kernel.cache_dir%` (directorio de la caché de la aplicación, normalmente `app/cache/`).

Esta página se ha dejado vacía a propósito

CAPÍTULO 19

Mejorando el rendimiento

Symfony es un framework suficientemente rápido para la mayoría de aplicaciones web. Además, cuando las aplicaciones son muy complejas, la arquitectura flexible y desacoplada de Symfony permite escalarlo para soportar decenas de millones de usuarios (como por ejemplo en sitios web como `spotify.com`, desarollado con Symfony).

En cualquier caso, existen opciones de configuración, técnicas y trucos que pueden mejorar todavía más el rendimiento de tus aplicaciones Symfony.

19.1 Mejorando el rendimiento de la parte del cliente

De poco sirve dedicar horas de esfuerzo a reducir 5 milisegundos el tiempo de generación de una página en el servidor si después el usuario debe esperar 10 segundos a que se carguen las imágenes, hojas de estilos y archivos JavaScript.

Asegúrate de que has optimizado al máximo la parte del cliente antes de empezar a optimizar la parte del servidor tal y como se explica en las siguientes secciones.

19.2 Mejorando el entorno de ejecución

Antes de empezar a optimizar tu aplicación Symfony, asegúrate de que has optimizado al máximo el servidor donde se va a ejecutar la aplicación en producción:

- Usar PHP 7 en vez de PHP 5.3 hará que tu aplicación se ejecute 4 veces más rápido.
- Activar OPcache (http://php.net/manual/es/book.opcache.php) hará que tu aplicación se ejecute 10 veces más rápido que si no lo activas.
- Activar las cachés y optimizaciones de tu base de datos hará que las consultas se ejecuten órdenes de magnitud más rápido.
- Usar servicios como Redis para cachear información que se accede continuamente reducirá la carga del servidor significativamente.

19.3 Desactivando las funcionalidades que no utilizas

Symfony está formado por tantos componentes, que es muy posible que tu aplicación no utilice varios de ellos. Para mejorar ligeramente el rendimiento, desactiva o elimina todo lo que no utilices:

1. Elimina en la clase `app/AppKernel.php` todos los *bundles* que no utilizas. Asegúrate de que los *bundles* que solo se necesitan al desarrollar la aplicación no se cargan en el entorno de producción.

2. Si tu aplicación no está traducida a varios idiomas, desactiva el servicio de traducción:

```
# app/config/config.yml
framework:
    translator: { enabled: false }
```

3. Si creas las plantillas con Twig, desactiva PHP como motor de plantillas:

```
# app/config/config.yml
framework:
    templating: { engines: ['twig'] }  # antes era -> engines: ['php', 'twig']
```

4. Si no defines la validación de las entidades con anotaciones, desactívalas:

```
# app/config/config.yml
framework:
    validation: { enable_annotations: false }
```

5. Desactiva el soporte de ESI si no lo necesitas para la caché de HTTP:

```
# app/config/config.yml
framework:
    esi: { enabled: false }
```

19.4 Mejorando la carga de las clases

El primer concepto importante relacionado con la carga de clases es el archivo `app/bootstrap.php.cache`. Se trata de un archivo gigantesco en el que se incluye el código de las clases más utilizadas por Symfony.

Así cada vez que se recibe una petición, Symfony abre y carga un único archivo, en vez de tener que localizar, abrir y cargar cientos de pequeños archivos. Si observas el código fuente de los controladores frontales de desarrollo (`web/app_dev.php`) o de producción (`web/app.php`) verás cómo siempre cargan este archivo:

```
// web/app.php
use Symfony\Component\HttpFoundation\Request;

$loader = require __DIR__.'/../app/autoload.php';
include_once __DIR__.'/../app/bootstrap.php.cache';

// ...
```

Cada vez que instalas o actualizas los *vendors* Symfony regenera automáticamente este archivo. Si por cualquier circunstancia quieres regenerar el archivo manualmente, accede al directorio raíz del proyecto y ejecuta el siguiente comando:

```
$ cd proyectos/cupon/
$ php vendor/sensio/distribution-bundle/Sensio/Bundle/DistributionBundle/Resources/bin/build_bootstrap.php
```

Por otra parte, cada vez que importas una clase mediante la instrucción `use`, Symfony debe localizar el archivo PHP correspondiente a partir del *namespace* indicado. Para reducir de forma significativa el tiempo empleado en localizar las clases, haz que Composer optimice la forma en la que asocia *namespaces* y archivos físicos ejecutando el siguiente comando:

```
$ cd proyectos/cupon/
$ composer dump-autoload --optimize
```

Optimizar la información de Composer puede mejorar hasta un 15% el rendimiento de la aplicación, por lo que no olvides utilizar la opción `--optimize` en el servidor de producción.

Además, para reducir todavía más el tiempo empleado en localizar las clases, puedes hacer que el cargador de clases de Symfony utilice la cache APC de PHP. Si tienes instalada y configurada correctamente la extensión APC en tu servidor PHP, descomenta las siguientes líneas que se encuentran en el archivo `web/app.php`:

```
// web/app.php
// ...

$apcLoader = new Symfony\Component\ClassLoader\ApcClassLoader(sha1(__FILE__), $loader);
$loader->unregister();
$apcLoader->register(true);
```

El cargador de clases `ApcClassLoader` guarda en la caché de APC la ruta de todas las clases importadas, por lo que la búsqueda sólo se realiza una vez por cada clase. Esta diferencia tiene un impacto positivo significativo en el rendimiento de la aplicación.

El primer argumento que se pasa a la clase `AppClassLoader` es el prefijo utilizado para guardar la información en la cache de APC. Este prefijo debe ser diferente para cada aplicación que se ejecute en el mismo servidor. Una forma simple de asegurarte de que así sea es usar el hash de la ruta del propio controlador frontal (`sha1(__FILE__)`) ya que será diferente para cada aplicación.

19.5 Mejorando el rendimiento de Doctrine

Doctrine es un proyecto independiente de Symfony, pero su rendimiento influye decisivamente en el rendimiento global de las aplicaciones. Por eso es necesario conocer cómo configurarlo y utilizarlo de la manera más eficiente.

19.5.1 Caché de configuración

La configuración por defecto de Symfony hace que Doctrine no utilice ninguna caché. Esto significa que en cada petición Doctrine carga toda la información de las entidades (llamada *mapping*) y convierte todas las consultas DQL al código SQL que se ejecuta en la base de datos.

¿Qué sentido tiene convertir una y otra vez una consulta DQL que nunca cambia? ¿Para qué se carga en cada petición la información de las entidades si casi nunca cambia? En el entorno de desarrollo este problema no es tan importante, pero **en producción todas las aplicaciones deberían usar la caché de configuración de Doctrine**.

Suponiendo que en el servidor de producción utilices APC (http://www.php.net/manual/es/book.apc.php) , añade lo siguiente en el archivo de configuración del entorno de producción:

```
# app/config/config_prod.yml
doctrine:
    orm:
        metadata_cache_driver: apc
        query_cache_driver:    apc
```

La configuración anterior indica a Doctrine que debe utilizar la caché de APC para guardar toda la información de las entidades (`metadata_cache_driver`) y todas las transformaciones de DQL en SQL (`query_cache_driver`).

Doctrine también soporta otros tipos de caché, como Memcache, Redis y Xcache (ver referencia de configuración de Doctrine (http://symfony.com/doc/current/reference/configuration/doctrine.html)). También dispones de un tipo especial de caché llamada `array` que simplemente guarda la información en arrays de PHP. Como no se trata de una caché de verdad, el rendimiento no mejora, por lo que simplemente se puede emplear para probar el uso de cachés en el entorno de desarrollo.

19.5.2 Caché de consultas

Al igual que sucede con la información de las entidades y la transformación de DQL, el resultado de algunas consultas a la base de datos no cambia en mucho tiempo. En la aplicación *Cupon* por ejemplo, la consulta que obtiene el listado de ciudades no cambia su resultado hasta que no se añade una nueva ciudad en la aplicación, algo que sucede pocas veces al año. ¿Para qué se consulta entonces en cada petición la lista de ciudades de la aplicación?

Doctrine también incluye soporte para guardar el resultado de las consultas en la caché. Los tipos de caché y sus opciones de configuración son los mismos que los explicados anteriormente. Por tanto, modifica el archivo de configuración de producción para añadir una nueva caché llamada `result_cache_driver`:

```
# app/config/config_prod.yml
doctrine:
    orm:
        metadata_cache_driver: apc
        query_cache_driver:    apc

        result_cache_driver:   apc
```

Al contrario de lo que sucede con las otras cachés, activar la caché `result_cache_driver` no implica una mejora inmediata en el rendimiento de la aplicación. El motivo es que después de activarla, debes modificar ligeramente todas y cada una de las consultas en las que quieras utilizar la

caché. El siguiente ejemplo muestra el cambio necesario en la consulta `findListaCiudades` que obtiene el listado de todas las ciudades de la aplicación:

```
// src/AppBundle/Repository/CiudadRepository.php

// Antes
public function findListaCiudades()
{
    $em = $this->getEntityManager();
    $consulta = $em->createQuery('SELECT c FROM CiudadBundle:Ciudad c ORDER BY
c.nombre');

    return $consulta->getArrayResult();
}

// Después
public function findListaCiudades()
{
    $em = $this->getEntityManager();
    $consulta = $em->createQuery('SELECT c FROM CiudadBundle:Ciudad c ORDER BY
c.nombre');
    $consulta->useResultCache(true);

    return $consulta->getArrayResult();
}
```

El método `useResultCache()` indica a Doctrine que el resultado de esta consulta debe buscarse primero en la caché de resultados. Si se encuentra el resultado cacheado, se devuelve inmediatamente sin tener que hacer la consulta en la base de datos. Si no se encuentra, se realiza una consulta normal a la base de datos y el resultado se guarda en la caché para su posterior reutilización.

Por defecto el resultado se guarda en la caché *para siempre*, es decir, hasta que se borre la caché o se reinicie el servicio de caché (APC, Memcache, Redis, etc.) Si quieres limitar el tiempo que un resultado permanece en la caché, el método `useResultCache()` admite un segundo parámetro con el tiempo de vida de la caché indicado en segundos:

```
$consulta = ...
$consulta->useResultCache(true, 3600); // el resultado se guarda 1 hora

$consulta->useResultCache(true, 600);  // el resultado se guarda 10 minutos

$consulta->useResultCache(true, 60);   // el resultado se guarda 1 minuto
}
```

Aunque parezca contradictorio, usar la caché de resultados no siempre es una buena idea. Considera por ejemplo la consulta que realiza la aplicación para encontrar la oferta del día en una determinada ciudad:

```
// src/AppBundle/Repository/OfertaRepository.php
public function findOfertaDelDia($ciudad)
{
    $em = $this->getEntityManager();

    $consulta = $em->createQuery('
        SELECT o, c, t
          FROM AppBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t
         WHERE o.revisada = true
           AND o.fechaPublicacion < :fecha
           AND c.slug = :ciudad
      ORDER BY o.fechaPublicacion DESC');
    $consulta->setParameter('fecha', new \DateTime('now'));
    $consulta->setParameter('ciudad', $ciudad);
    $consulta->setMaxResults(1);

    return $consulta->getSingleResult();
}
```

Como cada día sólo puede haber una *oferta del día* en una determinada ciudad, la consulta busca aquellas ofertas cuya fecha de publicación sea anterior al momento actual y se queda con la más reciente. El momento actual se obtiene mediante `new \DateTime('now')` por lo que siempre es diferente en cada consulta y en la caché se guardarían resultados que nunca se van a poder reutilizar.

Una posible solución a este problema consiste en limitar las posibles horas a las que se puede publicar una oferta. Así, en vez de `new \DateTime('now')` se podría utilizar `new \DateTime('today')`, por lo que la consulta sería la misma durante todo el día.

19.5.3 Mejorando tus consultas

19.5.3.1 Utiliza las consultas JOIN

Cuando se realiza una consulta de Doctrine, la única información que contienen los objetos del resultado es la que se indica en la parte `SELECT` de la consulta DQL. Si tratas de acceder a cualquier otra información relacionada con el objeto, se genera una nueva consulta a la base de datos.

La siguiente consulta obtiene por ejemplo la información sobre la oferta que se llama `Oferta de prueba`:

```
SELECT o FROM AppBundle:Oferta o WHERE o.nombre = "Oferta de prueba"
```

Si utilizas el objeto que devuelve la consulta anterior en una plantilla Twig:

```
<h1>Oferta</h1>

Nombre:      {{ oferta.nombre }}
Descripción: {{ oferta.descripcion }}
Precio:      {{ oferta.precio }}
Ciudad:      {{ oferta.ciudad.nombre }}
```

Las propiedades `nombre`, `descripcion` y `precio` pertenecen a la entidad `oferta` y por tanto, se incluyen dentro del objeto obtenido como resultado de la consulta. Sin embargo, cuando se trata de obtener el nombre de la ciudad asociada a la oferta (`oferta.ciudad.nombre`) esta información no está disponible en el resultado de la consulta. Por ese motivo, para renderizar la plantilla son necesarias dos consultas: la primera es la que realiza el código y la segunda es una consulta que Doctrine realiza automáticamente para obtener la información de la ciudad.

Si se utilizaran consultas SQL normales, este problema se soluciona fácilmente añadiendo un `JOIN` entre la tabla de las ofertas y la de las ciudades. Doctrine también permite realizar `JOIN` entre diferentes entidades. A continuación se muestra la misma consulta anterior a la que se ha añadido un `JOIN` con la entidad ciudad:

```
SELECT o FROM AppBundle:Oferta o JOIN o.ciudad c
 WHERE o.nombre = "Oferta de prueba"
```

Si pruebas esta nueva consulta, verás que el resultado es el mismo que antes, ya que también se necesitan dos consultas a la base de datos. La razón es que Doctrine sólo incluye en los objetos del resultado la información solicitada en el `SELECT`. Como la consulta sólo pide la información de la entidad `o` (que representa a la oferta) el resultado no incluye la información de la entidad `c` (que representa a la ciudad).

¿Para qué sirve entonces un `JOIN` como el anterior? Doctrine los denomina *JOIN normales* y se utilizan para refinar los resultados de búsqueda. La consulta anterior se podría restringir para que la búsqueda se limite a una única ciudad:

```
SELECT o FROM AppBundle:Oferta o JOIN o.ciudad c
 WHERE o.nombre = "Oferta de prueba"
   AND c.nombre = "Valencia"
```

Para obtener en una única consulta tanto la oferta como su ciudad asociada, además del `JOIN` es necesario modificar el `SELECT` de la consulta:

```
SELECT o, c FROM AppBundle:Oferta o JOIN o.ciudad c
            WHERE o.nombre = "Oferta de prueba"
```

La instrucción `SELECT o, c` indica que cada objeto del resultado debe contener tanto la información de la entidad `Oferta` como la información de la entidad `Ciudad`. Así se obtiene toda la información que necesita la plantilla en una única consulta.

Las consultas pueden añadir tantos `JOIN` como necesiten y el `SELECT` puede obtener tantas entidades como sea necesario. El siguiente ejemplo muestra una consulta que obtiene una oferta junto con la información de su tienda y su ciudad asociadas:

```
  SELECT o, c, t
    FROM AppBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t
   WHERE o.revisada = TRUE
     AND o.fechaPublicacion < :fecha
     AND c.slug = :ciudad
ORDER BY o.fechaPublicacion DESC
```

19.5.3.2 Utiliza la mejor *hidratación*

Después de ejecutar una consulta, Doctrine aplica a cada resultado obtenido un proceso conocido como *hidratación*. Este proceso no sólo penaliza seriamente el rendimiento, sino que resulta innecesario en muchos casos. Por tanto, además de optimizar las consultas DQL, es muy importante seleccionar bien el tipo de *hidratación* que se aplica a los resultados.

Los tipos de hidratación soportados por Doctrine son los siguientes:

- `Query::HYDRATE_OBJECT`, es el tipo por defecto. Devuelve como resultado un array en el que cada elemento es un objeto del mismo tipo que la primera entidad incluida en el `SELECT` de la consulta DQL. Este objeto, a su vez, puede incluir los objetos de todas sus entidades asociadas.
- `Query::HYDRATE_ARRAY`, devuelve como resultado un array en el que cada elemento es un array asociativo simple que sólo contiene el valor de las propiedades de las entidades incluidas en la consulta.
- `Query::HYDRATE_SCALAR`, similar al anterior, pero el array resultante es unidimensional, por lo que las entidades relacionadas no se incluyen en forma de array sino que sus propiedades se añaden directamente al único array devuelto.
- `Query::HYDRATE_SINGLE_SCALAR`, sólo se puede utilizar cuando se consulta una única propiedad de una sola entidad. El resultado es directamente el valor solicitado (un número, una cadena de texto, etc.)
- `Query::HYDRATE_SIMPLEOBJECT`, sólo se puede utilizar cuando la consulta devuelve una única entidad, sin importar el número de resultados obtenidos.

A continuación se muestra un ejemplo de todos los tipos de hidratación disponibles:

Hidratación `HYDRATE_OBJECT`

Consulta:

```
$consulta = $em->createQuery('
    SELECT o, c
      FROM AppBundle:Oferta o JOIN o.ciudad c
  ORDER BY o.id ASC
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_OBJECT);
```

Resultado:

```
array(
  0 => object(AppBundle\Entity\Oferta) {
    ["id":protected]=> 1
    ["nombre":protected]=> "Oferta #0-1"
    ["slug":protected]=> "oferta-0-1"
    ["descripcion":protected]=> "Lorem ipsum ..."
    ["condiciones":protected]=> "Lorem ipsum ..."
```

```
    ["rutaFoto":protected]=> "foto4.jpg"
    ["precio":protected]=> "28.13"
    ["descuento":protected]=> "9.85"
    ["fechaPublicacion":protected]=> object(DateTime) { ... }
    ["fechaExpiracion":protected]=>  object(DateTime) { ... }
    ["compras":protected]=> 0
    ["umbral":protected]=> 49
    ["revisada":protected]=> false
    ["ciudad":protected]=> object(AppBundle\Entity\Ciudad) {
      ["id":protected]=> 7
      ["nombre":protected]=> "Barcelona"
      ["slug":protected]=> "barcelona"
    }
    ["tienda":protected]=> object(Proxies\CuponAppBundleEntityTiendaProxy) {
      ["_entityPersister":...:private]=> object(Doctrine\ORM\Persisters\...) {
        ["id":protected]=> 36
        ["nombre":protected]=> "Tienda #36"
        // ... resto de propiedades de la tienda
      }
    }
  }
    // ... miles de líneas más con propiedades de Doctrine ...

  1 => object(AppBundle\Entity\Oferta) {
    // Misma estructura de datos para el segundo resultado y siguientes
  }

  // ...
)
```

El resultado es un array con tantos elementos como resultados produzca la consulta. Cada elemento del array es un objeto del mismo tipo que la primera entidad indicada en el `SELECT` de la consulta DQL. Las entidades relacionadas también se incluyen, pero no todas de la misma forma.

Si la entidad relacionada se ha incluido en un `JOIN`, sus datos se incluyen directamente en el objeto. Si la entidad no se ha incluido en algún `JOIN`, sus datos no están disponibles en el objeto y aparecen en forma de *proxy* (como por ejemplo la entidad asociada `Tienda` en el resultado anterior).

Un objeto de tipo *proxy* indica que aunque los datos no están disponibles, lo estarán si es necesario. Si tratas de acceder a las propiedades de un objeto no cargado, Doctrine hace automáticamente una consulta a la base de datos para obtener sus datos. Aunque este comportamiento puede parecer positivo, el peligro es que puede disparar rápidamente el número de consultas realizadas.

Por último, los objetos del array son tan monstruosamente gigantescos y recursivos, que si haces un `var_dump()` para ver sus contenidos, el navegador deja de responder. Utiliza en su lugar el `var_dump()` especial de Doctrine: `\Doctrine\Common\Util\Debug::dump($resultado);`

Hidratación `HYDRATE_ARRAY`

Consulta:

```
$consulta = $em->createQuery('
    SELECT o, c
      FROM AppBundle:Oferta o JOIN o.ciudad c
  ORDER BY o.id ASC
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_ARRAY);
```

Resultado:

```
array(
  0 => array (
    "id" => 1,
    "nombre" => "Oferta #0-1",
    "slug"   => "oferta-0-1",
    "descripcion" => "Lorem ipsum ...",
    "condiciones" => "Lorem ipsum ...",
    "rutaFoto"   => "foto4.jpg",
    "precio" => "28.13",
    "descuento" => "9.85",
    "fechaPublicacion" => object(DateTime) { ... },
    "fechaExpiracion"  => object(DateTime) { ... },
    "compras" => 0,
    "umbral"  => 49,
    "revisada" => false,
    "ciudad" => array(
      "id" => 7,
      "nombre" => "Barcelona",
      "slug" => "barcelona"
    )

  1 => array(
    // Misma estructura de datos para el segundo resultado y siguientes
  )

  ...
```

El resultado es un array con tantos elementos como resultados produzca la consulta. Cada elemento es un array asociativo que solamente contiene las propiedades de la entidad principal. Sus entidades relacionadas simplemente se incluyen en forma de array asociativo con las propiedades de esa entidad.

Si una entidad relacionada no se ha incluido en un `JOIN` sus datos no sólo no aparecen en el resultado sino que no se pueden obtener de ninguna manera (Doctrine no hace una consulta automática para obtener los datos, como sucedía en el caso `HYDRATE_OBJECT`).

Una diferencia importante respecto a la hidratación anterior es que ahora las propiedades conservan su nombre original (por ejemplo `fechaPublicacion`) y por tanto no se debe utilizar la notación de los *getters* de los objetos (por ejemplo `fechaPublicacion`). Esto significa que si utilizas la hidratación `HYDRATE_OBJECT` y la cambias por `HYDRATE_ARRAY`, es posible que tengas que modificar el nombre de las propiedades en algunas plantillas.

Hidratación `HYDRATE_SCALAR`

Consulta:

```
$consulta = $em->createQuery('
    SELECT o, c
      FROM AppBundle:Oferta o JOIN o.ciudad c
  ORDER BY o.id ASC
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_SCALAR);
```

Resultado:

```
array(
  0 => array(
    "o_id" => 1,
    "o_nombre" => "Oferta #0-1",
    "o_slug" => "oferta-0-1",
    "o_descripcion" => "Lorem ipsum ...",
    "o_condiciones" => "Lorem ipsum ...",
    "o_rutaFoto" => "foto4.jpg",
    "o_precio" => "28.13",
    "o_descuento" => "9.85",
    "o_fechaPublicacion" => object(DateTime) { ... },
    "o_fechaExpiracion"  => object(DateTime) { ... },
    "o_compras" => 0,
    "o_umbral" => 49,
    "o_revisada" => false,
    "c_id" => 7,
    "c_nombre" => "Barcelona",
    "c_slug" => "barcelona"
  )

  1 => array(
    // Misma estructura de datos para el segundo resultado y siguientes
  )
)
```

El resultado es un array con tantos elementos como resultados produzca la consulta. Cada elemento es un array asociativo unidimensional que contiene las propiedades de la primera entidad de la consulta y de todas sus entidades relacionadas. El nombre de la propiedad se prefija con el nombre asignado a cada entidad en la consulta DQL (`o` para ofertas, `c` para ciudades). Al igual que

sucede con HYDRATE_ARRAY, las entidades relacionadas que no se ha incluido en un JOIN, no sólo no aparecen en el array, sino que no se pueden obtener de ninguna manera.

Hidratación HYDRATE_SIMPLEOBJECT

Consulta:

```
$consulta = $em->createQuery('
    SELECT o
      FROM AppBundle:Oferta o JOIN o.ciudad c
     WHERE o.id = 1
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_SIMPLEOBJECT);
```

Resultado:

```
array(
  0 => object(AppBundle\Entity\Oferta) {
    ["id":protected]=> 1
    ["nombre":protected]=> "Oferta #0-1"
    ["slug":protected]=> "oferta-0-1"
    ["descripcion":protected]=> "Lorem ipsum ..."
    ["condiciones":protected]=> "Lorem ipsum ..."
    ["rutaFoto":protected]=> "foto4.jpg"
    ["precio":protected]=> "28.13"
    ["descuento":protected]=> "9.85"
    ["fechaPublicacion":protected]=> object(DateTime) { ... }
    ["fechaExpiracion":protected]=> object(DateTime) { ... }
    ["compras":protected]=> 0
    ["umbral":protected]=> 49
    ["revisada":protected]=> false
    ["ciudad":protected]=> NULL
    ["tienda":protected]=> NULL
)
```

El resultado es un array de un único elemento. Este elemento es un objeto del mismo tipo que la entidad incluida en el SELECT de la consulta DQL.

Hidratación HYDRATE_SINGLE_SCALAR

Consulta:

```
$consulta = $em->createQuery('
    SELECT o.nombre
      FROM AppBundle:Oferta o JOIN o.ciudad c
     WHERE o.id = 1
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_SINGLE_SCALAR);
```

Resultado:

```
"Oferta #0-1"
```

El resultado es directamente el valor de la propiedad consultada, sin arrays ni objetos de ningún tipo. Este tipo de hidratación es ideal cuando sólo quieres obtener una única propiedad de una entidad o un único valor calculado con las funciones `SUM`, `COUNT`, etc.

¿Qué tipo de hidratación se debe elegir? Depende del tipo de consulta, pero en general:

- `HYDRATE_ARRAY`, si el resultado de la consulta simplemente se visualiza en una plantilla y no sufre ningún tipo de modificación. Recuerda seleccionar todas las entidades relacionadas en el `SELECT` de la consulta DQL.
- `HYDRATE_SINGLE_SCALAR`, cuando se obtiene una sola propiedad o un único valor calculado con las funciones `SUM`, `COUNT`, etc.
- `HYDRATE_OBJET`, cuando se van a modificar los objetos del resultado o cuando prefieres cargar los datos de las entidades relacionadas bajo demanda, realizando nuevas consultas a la base de datos cuando sea necesario.
- `HYDRATE_SCALAR` y `HYDRATE_SIMPLEOBJECT` no son necesarios en la mayoría de aplicaciones web *normales*.

Doctrine incluye una serie de atajos para no tener que manejar las constantes `HYDRATE_*`:

- `getArrayResult()` es un atajo de `getResult(Query::HYDRATE_ARRAY)`
- `getScalarResult()` es un atajo de `getResult(Query::HYDRATE_SCALAR)`
- `getSingleScalarResult()` es un atajo de `getResult(Query::HYDRATE_SINGLE_SCALAR)`

Además, también se incluyen dos métodos `get*Result()` especiales, que se pueden combinar con los diferentes tipos de *hidratación*:

- `getOneOrNullResult()`, devuelve un único resultado o el valor `null` cuando la búsqueda no produce resultados. Si la consulta devuelve más de un resultado, se lanza la excepción de tipo `NonUniqueResultException`.
- `getSingleResult()`, devuelve un único objeto. Si no se encuentra ninguno, se lanza la excepción `NoResultException`. Si se encuentra más de uno, se lanza la excepción `NonUniqueResultException`.

19.6 Mejorando el rendimiento de la aplicación con cachés

Las técnicas de las secciones anteriores pueden llegar a mejorar el rendimiento de la aplicación de forma apreciable. Sin embargo, para conseguir un aumento exponencial del rendimiento, la única técnica eficaz es el uso de la caché de HTTP junto con un *proxy* inverso. El siguiente capítulo explica en detalle cómo conseguirlo.

Esta página se ha dejado vacía a propósito

CAPÍTULO 20

Caché

La inmensa mayoría de sitios y aplicaciones web son dinámicos. Esto significa que cuando un usuario solicita una página, el servidor web busca los contenidos (normalmente en una base de datos) y crea en ese momento la página HTML que entrega al usuario.

A pesar de su naturaleza dinámica, la información de los sitios web no suele cambiar a cada instante. Si un usuario solicita la portada del sitio y medio segundo después la solicita otro usuario, es poco probable que los contenidos hayan cambiado en ese lapso de tiempo.

Gracias a ello los sitios web pueden utilizar sistemas de caché para mejorar su rendimiento en varios órdenes de magnitud. Idealmente la caché de un sitio web guarda una copia del código HTML de cada página y lo sirve a los usuarios sin tener que acceder a la aplicación Symfony.

Desafortunadamente, son raras las ocasiones en las que se pueden guardar las páginas enteras. Lo habitual es que algunas partes de la página se actualicen constantemente y no se puedan guardar en la misma caché que el resto de la página. Además, otras partes de la página pueden depender del usuario conectado en la aplicación, por lo que tampoco se pueden guardar en la caché común.

Como es habitual, Symfony no implementa su propio mecanismo de caché, sino que hace uso de la caché del estándar HTTP tal y como se explica en la parte 6 del estándar HTTP bis (http://datatracker.ietf.org/doc/draft-ietf-httpbis-p6-cache/) .

20.1 La caché del estándar HTTP

El siguiente esquema muestra gráficamente los diferentes tipos de caché que existen entre tu aplicación Symfony y el navegador del usuario:

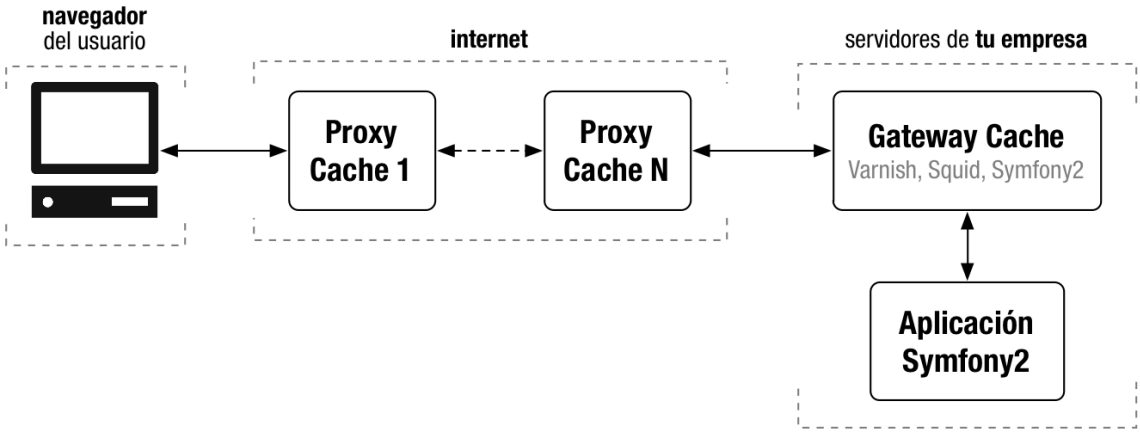

Figura 20.1 Cachés que pueden existir entre tu aplicación Symfony y el navegador del usuario

Los diferentes tipos de caché existentes son los siguientes:

1. **Navegador**: la caché del *navegador* del usuario.
2. **Proxy cache**: la caché del *proxy* por el que atraviesan las comunicaciones del usuario con Internet. Pueden existir varios *proxy caches* entre el usuario y la aplicación. Muchas empresas los utilizan para reducir el tráfico con Internet y para hacer cumplir la normativa interna de la empresa. Las operadoras que ofrecen la conexión a Internet también suelen instalar este tipo de *proxy caches* por los mismos motivos, aunque no suelen informar al usuario de ello.
3. **Gateway cache**: la caché instalada por los administradores de sistemas para reducir la carga del servidor web y aumentar así el rendimiento de la aplicación. Normalmente se instalan en uno o más servidores físicos diferentes al servidor web, pero también se pueden hacer *gateway caches* con software. También se conocen como *proxy inverso* o *reverse proxy cache*.

La *gateway cache* es la única sobre la que tienes un control absoluto. El resto de cachés están fuera de tu control, pero puedes manipularlas con las cabeceras definidas en el estándar HTTP.

La caché del navegador es **privada** (`private` en terminología HTTP) lo que significa que sus contenidos no se comparten entre diferentes usuarios. El resto de cachés son esencialmente **públicas** (`shared` en terminología HTTP), ya que su objetivo es servir los mismos contenidos cacheados a muchos usuarios diferentes. Sin embargo, cuando se accede a páginas seguras (`https://`) o cuando lo indique la aplicación, estas cachés también se pueden convertir en privadas.

Para evitar efectos indeseados, por defecto Symfony devuelve todas las respuestas como privadas. Puedes modificar este comportamiento gracias a los métodos `setPrivate()` y `setPublic()` del objeto `Response`. El siguiente ejemplo muestra cómo hacer que la respuesta devuelta por la acción de la portada sea pública, de forma que se pueda cachear y reutilizar entre varios usuarios:

```
class DefaultController extends Controller
{
    // ...

    public function portadaAction($ciudad)
    {
        // ...

        // Antes: (la respuesta es privada)
        // return $this->render('portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora: (la respuesta es pública)
        $respuesta = $this->render('portada.html.twig', array(
            'oferta' => $oferta
        ));
        $respuesta->setPublic();
        return $respuesta;
```

```
    }
}
```

Una forma más rápida de configurar la caché de las acciones sin tener que modificar su código es usar la anotación `@Cache` de la siguiente manera:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(public=true)
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

Por último, ten en cuenta que sólo se pueden cachear las peticiones realizadas con métodos seguros e *idempotentes* (que siempre obtienen el mismo resultado sin importar las veces que se repita la petición). En el estándar HTTP, los únicos métodos de este tipo son `GET`, `HEAD`, `OPTIONS` y `TRACE`. Así que no es posible cachear las peticiones de tipo `POST`, `PUT` y `DELETE`.

20.2 Estrategias de caché

El estándar HTTP define dos estrategias de caché claramente diferenciadas: expiración y validación. A su vez, estas estrategias utilizan cuatro cabeceras HTTP: `Cache-Control`, `Expires`, `ETag`, `Last-Modified`. Symfony soporta las dos estrategias e incluye métodos para manipular fácilmente las cuatro cabeceras relacionadas.

Para simplificar las explicaciones de las próximas secciones, en todos los ejemplos se supone que el usuario ha solicitado una página al servidor a las 15:00 horas GMT del 8 de diciembre de 2013, lo que en el formato de las fechas de HTTP se indica como `Sun, 08 Dec 2013 15:00:00 GMT`.

20.2.1 La estrategia de expiración

La **caché por expiración** es la estrategia recomendada siempre que sea posible. Sus resultados son los mejores porque reduce tanto el ancho de banda como la cantidad de CPU utilizada. Su comportamiento se basa en indicar cuándo caduca un contenido. Mientras no caduque, las cachés no vuelven a solicitarlo al servidor sino que sirven directamente el contenido cacheado. La fecha de caducidad de los contenidos se indica con la cabecera `Expires` o con la cabecera `Cache-Control`.

20.2.2 La cabecera Expires

La cabecera `Expires` indica en su valor la fecha y hora a partir de la cual se considera que el contenido está caducado y se debe volver a pedir al servidor. En Symfony su valor se establece con el método `setExpires()` del objeto `Response`. El siguiente código hace que la portada del sitio caduque cinco minutos después de crearla:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(expires="now + 5 minutes")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

Si cargas la página en el navegador, no notarás ningún cambio aparente. Sin embargo, si observas las cabeceras HTTP incluidas en la respuesta del servidor, verás lo siguiente:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
Content-Type: text/html; charset=UTF-8
Cache-Control: private, must-revalidate
Expires: Sun, 08 Dec 2013 15:05:00 GMT

200 OK
```

La opción `expires` hace que la respuesta del servidor incluya la cabecera `Expires` con el valor de la fecha de caducidad de la página (las `15:05:00`, cinco minutos después de crear la portada).

> **NOTA** Para eliminar la cabecera `Expires` de la respuesta, establece el valor de `expires` a `null`.

Observa cómo la respuesta también incluye la cabecera `Cache-Control` con el valor `private`. Esta cabecera la añade automáticamente Symfony para indicar que la respuesta es privada y no se debe entregar a ningún otro usuario salvo al que realizó la solicitud.

¿Cómo se comporta la aplicación después de añadir la cabecera `Expires`?

- Si es la primera vez que el usuario solicita la página, se genera en ese momento y se entregan sus contenidos.
- Si el mismo usuario, utilizando el mismo navegador, vuelve a solicitar la página sin que hayan pasado cinco minutos desde su petición anterior, el navegador muestra al instante la página guardada en su caché. Ni siquiera se realiza una petición al servidor. Este comportamiento se mantiene aunque cierres el navegador completamente y lo vuelvas a abrir para solicitar la página (siempre que no hayan pasado los cinco minutos establecidos como caducidad).
- Si el mismo usuario solicita la página con otro navegador o han pasado más de cinco minutos desde que solicitó la página o si pulsa el icono de *Recargar página*, el navegador borra la página que tiene en su caché y la vuelve a solicitar al servidor, mostrando al usuario los nuevos contenidos.
- Si otro usuario solicita la página, la aplicación vuelve a generarla y se la entrega al nuevo usuario.

20.2.3 La cabecera Cache-Control

Aunque la cabecera `Expires` es muy sencilla, la recomendación es utilizar siempre que sea posible la cabecera `Cache-Control`, ya que dispone de muchas más opciones relacionadas con el comportamiento de la caché. Además, la cabecera `Cache-Control` puede establecer simultáneamente varias opciones de la caché, separándolas entre sí por una coma. Según el estándar HTTP, las opciones disponibles son las siguientes:

```
Cache-Control: public, private, no-cache, no-store, no-transform, must-revalidat
e, proxy-revalidate, max-age=NN, s-maxage=NN, cache-extension
```

Las opciones de la caché se indican simplemente añadiendo el nombre de la opción, salvo las opciones `max-age` y `s-maxage` que requieren indicar también un número. A continuación se explica el significado de cada opción:

- `public`, indica que la caché es pública, por lo que es seguro guardar el contenido en la caché y servirlo a cualquier usuario que lo solicite.
- `private`, indica que la caché es privada, por lo que el contenido sólo se puede guardar en la caché del usuario que realizó la petición.
- `no-cache`, impide que el contenido solicitado por el usuario se sirva desde la caché, por lo que obliga a pedir los contenidos al servidor.
- `no-store`, sus efectos son similares a la opción anterior, ya que impide que el contenido entregado por el servidor se guarde en la caché, sin importar si es pública o privada.
- `must-revalidate`, indica que cuando el contenido caduca ya no se puede seguir mostrando al usuario, por lo que es necesario realizar una petición al servidor. Esto se debe cumplir aun cuando el contenido (caducado) se encuentre en la caché y no se pueda contactar con el servidor (porque está caído, por problemas de red, etc.)
- `proxy-revalidate`, tiene el mismo significado que `must-revalidate`, pero no se aplica a las cachés privadas.

- `max-age=NN`, el contenido debe considerarse caducado después de que transcurran el número de segundos indicado desde que se creó el contenido. En otras palabras, se trata del tiempo de vida en segundos del contenido.
- `s-maxage=NN`, tiene el mismo significado que `max-age` pero se aplica sobre las cachés públicas, donde además tiene prioridad sobre las cabeceras `max-age` y `Expires`.
- `no-transform`, indica que ningún intermediario (*proxy*, caché, etc.) debe cambiar ni el contenido de la respuesta ni el valor de las cabeceras `Content-Encoding`, `Content-Range` y `Content-Type`.

El siguiente código hace que la caché de la portada sea privada y tenga un tiempo de vida de 5 minutos (`300` segundos):

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(maxage="300")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

Modificar el objeto de la respuesta con la opción `maxage` tampoco tiene ningún efecto aparente al recargar la página en el navegador. No obstante, si vuelves a observar las cabeceras HTTP de la respuesta del servidor:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
Content-Type: text/html; charset=UTF-8
Cache-Control: max-age=300, private

200 OK
```

La opción `maxage` añade la cabecera `Cache-Control` en la respuesta para definir con `max-age` el número de segundos establecido en el controlador. Además, como no se indica el tipo de caché, Symfony añade la opción `private` para hacerla privada.

Después de utilizar la opción `maxage` la aplicación se comporta exactamente igual que tras añadir la cabecera `Expires`:

- Si es la primera vez que el usuario solicita la página, se genera en ese momento y se entregan sus contenidos.
- Si el mismo usuario, utilizando el mismo navegador, vuelve a solicitar la página sin que hayan pasado cinco minutos desde su petición anterior, el navegador muestra al instante la página guardada en su caché. Ni siquiera se realiza una petición al servidor. Este comportamiento se mantiene aunque cierres el navegador completamente y lo vuelvas a abrir para solicitar la página (siempre que no hayan pasado los cinco minutos establecidos como caducidad).
- Si el mismo usuario solicita la página con otro navegador o han pasado más de cinco minutos desde que solicitó la página o si pulsa el icono de *Recargar página*, el navegador borra la página que tiene en su caché y la vuelve a solicitar al servidor, mostrando al usuario los nuevos contenidos.
- Si otro usuario solicita la página, la aplicación vuelve a generarla y se la entrega al nuevo usuario.

NOTA Resulta habitual utilizar la cabecera `Cache-Control: max-age=0` cuando el servidor no quiere que el navegador guarde la página en su caché o cuando el navegador quiere que el servidor le entregue una página nueva en vez de la de la caché.

Por otra parte, en vez de `maxage` también puedes establecer la opción `smaxage`:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(smaxage="300")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }
}
```

La opción `smaxage` no solo añade la opción `s-maxage` en la cabecera `Cache-Control`, sino que también establece automáticamente la opción `public`:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
Content-Type: text/html; charset=UTF-8
Cache-Control: public, s-maxage=300

200 OK
```

El motivo por el que se añade la opción `public` es que la opción `s-maxage` establece el tiempo de vida del contenido dentro de una caché pública, por lo que implícitamente está indicando que la caché debe ser pública.

El efecto que produce la opción `s-maxage` en el rendimiento de la aplicación no se puede explicar en detalle hasta que no se configure más adelante un *proxy inverso* para atender las peticiones de la aplicación.

Si estableces tanto la cabecera `Expires` como las opciones `maxage` o `smaxage` en una misma respuesta, la opción `max-age` tiene preferencia sobre `Expires` en las cachés privadas y la opción `s-maxage` tiene preferencia sobre `Expires` en las cachés públicas.

Por último, el objeto `Response` de Symfony incluye el método `expire()` para marcar la respuesta como caducada. Internamente este método establece la *edad* de la página (cabecera `Age`) al valor `max-age` establecido en la cabecera `Cache-Control`.

20.2.4 La estrategia de validación

La **caché por validación** es la estrategia que debes utilizar siempre que no puedas hacer uso de la **caché por expiración**. Este es el caso de las páginas que se deben actualizar tan pronto como varíe su información y en las que el ritmo de actualización es impredecible, por lo que no tiene sentido utilizar ni `Expires` ni `Cache-Control`.

El rendimiento de la aplicación con este modelo de caché no mejora tanto como con el anterior, ya que sólo se ahorra ancho de banda, pero el consumo de CPU se mantiene. Su comportamiento se basa en preguntar al servidor si la página que se encuentra en la caché sigue siendo válida o no. Para ello se añade en cada página un identificador único que cambia cuando se modifiquen los contenidos (etiqueta `ETag`) o se añade la fecha en la que los contenidos de la página se modificaron por última vez (etiqueta `Last-Modified`).

Cuando el usuario solicita una página que el navegador tiene en la caché, el navegador envía al servidor el valor de la etiqueta `ETag` o de `Last-Modified` y pregunta si la página sigue siendo válida. Si lo es, el servidor responde con un código de estado `304` (`Not modified`) y el navegador muestra la página cacheada. Si no es válida, el servidor genera de nuevo la página y la entrega al navegador.

20.2.5 La cabecera ETag

Según el estándar HTTP, el valor de la cabecera `ETag` (del inglés *entity-tag*) es *"una cadena de texto que identifica de forma única las diferentes representaciones de un mismo recurso"*. En otras palabras, el

valor `ETag` no sólo es único para cada página del sitio sino que varía cada vez que cambia algún contenido de la página.

El contenido de `ETag` o la forma de calcularlo es responsabilidad de la aplicación, ya que ni el estándar HTTP ni Symfony proporcionan ninguna recomendación ni utilidad para calcularlo. El motivo es que sólo el autor de la aplicación sabe cuándo una página ha variado sus contenidos y por tanto, cuándo se debe modificar el `ETag` de la página.

El objeto `Response` de Symfony incluye un método llamado `setEtag()` para añadir la cabecera `ETag` en la respuesta. En el siguiente ejemplo, el valor del `ETag` se calcula mediante el valor del `md5()` del contenido completo de la página, de forma que no varíe a menos que se modifiquen los contenidos de la página:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(maxage="300")
     */
    public function portadaAction($ciudad)
    {
        // ...

        $respuesta = $this->render('portada.html.twig', array(
            'oferta' => $oferta
        ));

        $etag = md5($respuesta);
        $respuesta->setEtag($etag);

        return $respuesta;
    }
}
```

El código anterior hace que la respuesta del servidor incluya las siguientes cabeceras:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
ETag: "5391e925cae5dc96784db0f1cd1890e7"
Content-Type: text/html; charset=UTF-8
Cache-Control: private, must-revalidate
```

```
200 OK
```

Después de añadir la cabecera `ETag`, la aplicación se comporta de la siguiente manera:

- Si es la primera vez que el usuario solicita la página, se genera en ese momento y se entregan sus contenidos junto con la etiqueta `ETag`.
- Si el mismo usuario utiliza el mismo navegador para volver a solicitar la página, el navegador pregunta al servidor si la página que tiene en la caché sigue siendo válida. Para ello añade en la petición la cabecera `If-None-Match` y el valor de la etiqueta `ETag`. El servidor vuelve a calcular la `ETag` del contenido y:
  - Si el valor de las dos `ETag` coincide, el servidor devuelve como respuesta un código de estado `304` (*Not modified*) y el navegador muestra al usuario la página que tiene en su caché.
  - Si no coinciden, el servidor vuelve a generar la página y la entrega al usuario junto con la nueva etiqueta `ETag`.
- Si otro usuario solicita la página, la aplicación vuelve a generarla y se la entrega al nuevo usuario junto con la etiqueta `ETag`.

El problema del código anterior es que añade la etiqueta `ETag` en la respuesta pero no comprueba si su valor coincide con el que envía el navegador, así que la aplicación nunca devuelve una respuesta `304` y es como si no existiera la caché.

Solucionar este problema es muy sencillo gracias al método `isModified()` del objeto `Response`. Este método compara el valor de la etiqueta `ETag` de la respuesta con el de la etiqueta `ETag` de la petición que se pasa como argumento. Si coinciden, envía una respuesta con código de estado `304` (*Not modified*). Si no coinciden, envía la respuesta completa normal. El siguiente ejemplo muestra el código completo necesario cuando se utilizan etiquetas `ETag`:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/{ciudad}", name="portada")
     * @Cache(maxage="300")
     */
    public function portadaAction($ciudad)
    {
        $respuesta = $this->render('portada.html.twig', array(
            'oferta' => $oferta
```

```
        ));

        $etag = md5($respuesta);
        $respuesta->setEtag($etag);

        $respuesta->isNotModified($this->getRequest());

        return $respuesta;
    }
}
```

Utilizar el resumen MD5 del contenido como `ETag` de una página es la solución más sencilla, pero también la más ineficiente. El motivo es que hay que generar la página completa para calcular el `ETag`.

Como el valor de la etiqueta `ETag` es una cadena de texto sin ningún formato preestablecido, puedes utilizar cualquier valor que indique si el contenido de la página ha variado. Así por ejemplo, en un sitio web que muestra noticias podrías utilizar la concatenación del número de versión de la noticia y del número total de comentarios:

```
$etag = $noticia->getVersion().'-'.count($noticia->getComentarios());
$respuesta->setEtag($etag);
// Resultado: ETag: "3-145"
```

Este valor de `ETag` también se puede definir mediante las anotaciones:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Cache(ETag="noticia.getVersion() ~ noticia.getComentarios()")
     */
    public function portadaAction()
    {
        // ...
    }
}
```

Cada vez que un editor revise la noticia o cada vez que un usuario añada un comentario, el valor de la etiqueta `ETag` variará. Obviamente esta etiqueta no es tan precisa como utilizar el MD5 de todo el contenido de la página, pero el esfuerzo requerido para calcularla es mucho menor.

El estándar HTTP denomina *etiquetas débiles* a estas etiquetas `ETag` muy sencillas de calcular pero que no tienen una gran precisión. De hecho, estas etiquetas pueden no cambiar para cada posible

variación de los contenidos, bien porque no se pueda calcular un nuevo valor o bien porque no sea eficiente recalcularlo con cada cambio.

Si utilizas etiquetas débiles en tu aplicación, pasa el valor `true` como segundo argumento del método `ETag`. Symfony prefija el valor de estas etiquetas con los caracteres `W/` (del inglés *weak*), tal y como dicta el estándar HTTP:

```
// Etiqueta ETag fuerte
$etag = md5($respuesta);
$respuesta->setEtag($etag);
// Resultado: ETag: "5391e925cae5dc96784db0f1cd1890e7"

// Etiqueta ETag débil
$etag = $noticia->getVersion().'-'.count($noticia->getComentarios());
$respuesta->setEtag($etag, true);
// Resultado: ETag: W/"3-145"
```

20.2.6 La cabecera Last–Modified

La cabecera `Last-Modified` indica la fecha y hora a la que el servidor web cree que fueron modificados por última vez los contenidos de la página. Una vez más, el estándar HTTP no proporciona ninguna indicación ni herramienta para determinar el valor de esta cabecera.

Normalmente las páginas web complejas determinan la fecha de última modificación de cada contenido de la página y se quedan con el más reciente. Por ello, si sabes que tu aplicación utilizará este tipo de caché, es una buena idea añadir la propiedad `updated_at` en todas las entidades de la aplicación, de manera que puedas determinar fácilmente cuándo se modificaron por última vez.

El valor de esta cabecera en Symfony se establece con ela opción `lastModified`:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Cache(lastModified="oferta.getUpdatedAt()")
     */
    public function portadaAction()
    {
        // ...
    }
}
```

El código anterior hace que la respuesta del servidor incluya las siguientes cabeceras:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
Last-Modified: Wed, 06 Dec 2013 13:00:00 GMT
Content-Type: text/html; charset=UTF-8
Cache-Control: private, must-revalidate

200 OK
```

Después de añadir la cabecera `Last-Modified`, la aplicación se comporta de la siguiente manera:

- Si es la primera vez que el usuario solicita la página, se genera en ese momento y se entregan sus contenidos junto con la etiqueta `Last-Modified`.
- Si el mismo usuario utiliza el mismo navegador para volver a solicitar la página, el navegador pregunta al servidor si la página que tiene en la caché sigue siendo válida. Para ello añade en la petición la cabecera `If-Modified-Since` y el valor de la etiqueta `Last-Modified` recibida anteriormente.
    - Si la fecha indicada en la etiqueta `Last-Modified` del navegador es igual o más reciente que la calculada por el servidor, se devuelve como respuesta un código de estado `304` (*Not modified*) y el navegador muestra al usuario la página que tiene en su caché.
    - Si la fecha enviada por el navegador es anterior a la fecha calculada por el servidor, se vuelve a generar la página y se entrega al usuario junto con la nueva fecha en la cabecera `Last-Modified`.
- Si otro usuario solicita la página, la aplicación vuelve a generarla y se la entrega al nuevo usuario junto con la etiqueta `Last-Modified`.

Al igual que sucedía en el caso de la etiqueta `ETag` el código mostrado anteriormente no cumple con el comportamiento deseado para la etiqueta `Last-Modified`. Sólo se establece su valor, pero no se compara con el valor enviado por el navegador, por lo que nunca se devuelve una respuesta `304` y la caché no está funcionando como se espera.

La solución consiste en utilizar el mismo método `isModified()` utilizado anteriormente, ya que sirve tanto para `ETag` como para `Last-Modified`. Simplemente pasa como argumento el objeto que representa a la petición del usuario y el método se encarga de realizar la comparación y de generar la respuesta `304` cuando sea necesario:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
```

```
     * @Route("/{ciudad}", name="portada")
     */
    public function portadaAction()
    {
        // ...

        $respuesta = $this->render('portada.html.twig', array(
            'oferta' => $oferta
        ));

        $fecha = $oferta->getUpdatedAt();
        $respuesta->setLastModified($fecha);

        $respuesta->isNotModified($this->getRequest());

        return $respuesta;
    }
}
```

Además del método `isNotModified()`, el objeto `Response` dispone del método `setNotModified()` para crear rápidamente una respuesta de tipo `304` (*Not modified*). Además de establecer el código de estado, este método elimina cualquier contenido de la respuesta.

20.2.7 La estrategia por defecto de Symfony

Symfony soporta todas las estrategias de caché del estándar HTTP a través de las cuatro cabeceras `Expires`, `Cache-Control`, `ETag` y `Last-Modified`. Como estas cabeceras no son mutuamente excluyentes, Symfony incluye algunas cabeceras y opciones por defecto:

- Si no incluyes ni `Cache-Control` ni `Expires` ni `ETag` ni `Last-Modified`, se añade automáticamente la cabecera `Cache-Control=no-cache` para indicar que esta página no utiliza ninguna caché.
- Si incluyes `Expires` o `ETag` o `Last-Modified`, se añade automáticamente la cabecera `Cache-Control=private, must-revalidate` para indicar que esta página utiliza una caché privada.
- Si incluyes `Cache-Control` pero no defines ni la propiedad `public` ni `private`, se añade automáticamente el valor `private` para hacer que la página se guarde en una caché privada.

20.3 Cacheando con *reverse proxies*

Las estrategias y cabeceras explicadas en las secciones anteriores se aplican en todos los tipos de caché que existen entre el usuario y la aplicación: el navegador, uno o más *proxy caches* y uno o más *reverse proxies*.

No obstante, la única manera de aumentar exponencialmente el rendimiento de la aplicación consiste en utilizar un *reverse proxy*. Esta es la única caché sobre la que tienes un control absoluto, por lo que puedes crear una caché pública y ya no dependes del comportamiento (en ocasiones aleatorio) de los navegadores de los usuarios.

Los *reverse proxies* más conocidos son [Varnish](https://www.varnish- cache.org/) y Squid (http://www.squid-cache.org/) . Los dos son proyectos de software libre y relativamente fáciles de instalar, configurar y utilizar. En cualquier caso, como utilizar un *reverse proxy* es tan importante en las aplicaciones web profesionales, Symfony ya incluye un *reverse proxy*. Está programado en PHP, por lo que no es tan rápido como los anteriores, pero tiene la ventaja de que no hace falta instalar nada.

20.3.1 El *reverse proxy* de Symfony

La gran ventaja del *reverse proxy* de Symfony es que modificando solamente dos líneas de código en tu aplicación, puedes multiplicar su rendimiento. Observa el código del controlador frontal que se utiliza en producción:

```
// web/app.php

// ...

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
//$kernel = new AppCache($kernel);

//Request::enableHttpMethodParameterOverride();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

Su funcionamiento es realmente sencillo: se crea el núcleo o *kernel* de la aplicación para el entorno de producción, se carga una caché con clases importantes de Symfony y se despacha la petición del usuario.

Para activar el *reverse proxy* de Symfony, sólo tienes que descomentar las dos líneas que aparecen comentadas en el código anterior:

```
// web/app.php

// ...

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel = new AppCache($kernel);

Request::enableHttpMethodParameterOverride();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

El código anterior simplemente *encierra* el *kernel* de la aplicación en otra clase de tipo caché. Esta clase llamada `AppCache` dispone inicialmente del siguiente contenido:

```
<?php
// app/AppCache.php

require_once __DIR__.'/AppKernel.php';
use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
{
}
```

Antes de probar los cambios realizados es importante asegurar que la caché está bien configurada. Las opciones de configuración se indican añadiendo un método llamado `getOptions()` en la clase anterior. A continuación se muestran todas las opciones disponibles y los valores que Symfony les asigna por defecto:

```
// app/AppCache.php
require_once __DIR__.'/AppKernel.php';
use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends Cache
{
    protected function getOptions()
    {
        return array(
            'debug'                  => false,
            'default_ttl'            => 0,
            'private_headers'        => array('Authorization', 'Cookie'),
            'allow_reload'           => false,
            'allow_revalidate'       => false,
            'stale_while_revalidate' => 2,
            'stale_if_error'         => 60,
        );
    }
}
```

Este es el propósito de cada opción:

- `debug`: permite añadir información de depuración en la respuesta generada por la caché de Symfony. Si no se indica explícitamente un valor, se le asigna el mismo valor que el de la opción `debug` del kernel. Cuando vale `true`, la respuesta incluye una cabecera especial llamada `X-Symfony-Cache` que indica si la página se ha obtenido de la caché (`fresh`) o si se ha generado dinámicamente (`miss`).
- `default_ttl`: el tiempo en segundos durante el que se guarda la página en la caché. Esta opción no se tiene en cuenta en las respuestas que definen sus propias cabeceras `Cache-Control` o `Expires`.

- `private_headers`: si la petición del usuario contiene alguna de las cabeceras incluidas en este array, la caché se hace privada automáticamente (con `Cache-Control: private`) a menos que la respuesta indique explícitamente si la respuesta es pública o privada mediante la cabecera `Cache-Control`. El valor por defecto es `array('Authorization', 'Cookie')`, por lo que las páginas que requieren usuario + contraseña y las que incluyen *cookies* no se guardan en la caché pública.
- `allow_reload`: indica si el navegador del usuario puede forzar la recarga de la caché incluyendo la cabecera `Cache-Control: "no-cache"` en la petición. Su valor por defecto es `false`, aunque para cumplir con la especificación RFC 2616 del estándar HTTP debería ser `true`.
- `allow_revalidate`: indica si el navegador del usuario puede forzar la revalidación de la caché incluyendo la cabecera `Cache-Control: "max-age=0"` en la petición. Su valor por defecto es `false`, aunque para cumplir con la especificación RFC 2616 del estándar HTTP debería ser `true`.
- `stale_while_revalidate`: indica el número de segundos durante los cuales la caché puede seguir enviando una página caducada mientras revalida su contenido en segundo plano. Su valor por defecto es `2`, pero no se tiene en cuenta si se emplea la extensión `stale-while-revalidate` de `Cache-Control`.
- `stale_if_error`: indica el número de segundos durante los cuales la caché puede seguir enviando una página caducada cuando se produce un error. Su valor por defecto es `60`, pero no se tiene en cuenta si se emplea la extensión `stale-if-error` de `Cache-Control`.

20.3.2 Aumentando exponencialmente el rendimiento de la aplicación

Una vez activado y configurado el *reverse proxy* de Symfony, ya sólo es necesario refactorizar ligeramente el código del controlador de aquellas páginas que se pueden guardar en una caché pública.

20.3.2.1 La portada

La primera página que deberías modificar es la **portada**, ya que sus contenidos se pueden guardar en una caché pública y como seguramente será la página más visitada del sitio web, la mejora en el rendimiento será muy notable. Añade el siguiente código para que la portada permanezca `1` minuto en la caché:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", defaults={ "ciudad" = "%app.ciudad_por_defecto%" }, n
ame="portada")
     * @Cache(smaxage="60")
     */
```

```
    public function portadaAction($ciudad)
    {
        // ...
    }

    // ...
}
```

Después de estos cambios, la respuesta que envía el servidor incluye las siguientes cabeceras:

```
Date: Sun, 08 Dec 2013 15:00:00 GMT
Content-Length: 1024
Server: Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8r
Content-Type: text/html; charset=UTF-8
Age: 0
Cache-Control: public, s-maxage=60
X-Symfony-Cache: GET /: fresh

200 OK
```

En primer lugar se incluye la cabecera `Cache-Control` que indica que la página se guarda en una caché pública y que su tiempo de vida son 60 segundos. Después se incluye la cabecera `Age` que indica el número de segundos que han transcurrido desde que se generó la página y se guardó en la caché.

Este ejemplo supone que la petición del usuario es la primera que se realiza, por lo que el valor de `Age` es `0`. En las siguientes peticiones este valor aumenta hasta que llegue a `60`, que es cuando la página se borra de la caché y se vuelve a generar.

Por último, si has modificado la opción `debug` de la caché a `true` en el archivo `app/AppCache.php`, también se incluye la cabecera `X-Symfony-Cache`. Su valor indica que la página obtenida mediante la petición `GET /` se ha podido obtener de la caché (`fresh`) en vez de haber sido generada (`miss`).

En la máquina utilizada para las pruebas de rendimiento, este simple cambio multiplicó por cuatro el rendimiento de la portada.

20.3.2.2 El resto de páginas de la aplicación

La página de **detalle de una oferta** es muy similar a la portada del sitio, por lo que también se puede guardar en la caché pública, por ejemplo durante 10 minutos:

```
// src/AppBundle/Controller/OfertaController.php

/**
 * @Route("/{ciudad}/ofertas/{slug}", name="oferta")
 * @Cache(smaxage="60")
 */
public function ofertaAction($ciudad, $slug)
{
```

```
    // ...
}
```

La página de **ofertas recientes en una ciudad** es susceptible de ser cacheada por un largo período de tiempo, como por ejemplo una hora:

```
// src/AppBundle/Controller/CiudadController.php

/**
 * @Route("/{ciudad}/recientes", name="ciudad_recientes")
 * @Cache(smaxage="3600")
 */
public function recientesAction(Request $request, $ciudad)
{
    // ...
}
```

La página de **ofertas recientes de una tienda** también se puede cachear durante una hora por ejemplo:

```
// src/AppBundle/Controller/TiendaController.php
//
/**
 * @Route("/{ciudad}/tiendas/{tienda}", requirements={ "ciudad" = ".+" }, nam
e="tienda_portada")
 * @Cache(smaxage="3600")
 */
public function portadaAction(Request $request, $ciudad, $tienda)
{
    // ...
}
```

20.3.2.3 Las páginas privadas

El resto de páginas del *frontend* son privadas, ya que requieren que el usuario introduzca sus credenciales. Recuerda que la configuración por defecto de la caché de Symfony impide cachear páginas cuya petición incluya las cabeceras `Authorization` o `Cookie`.

Por otra parte, en la mayoría de aplicaciones web no existen páginas totalmente públicas o totalmente privadas, sino que las páginas incluyen partes públicas y privadas.

Si accedes por ejemplo a la portada del sitio (`http://127.0.0.1:8000/app.php`) verás que no es cierto que sea una página completamente pública. En la zona lateral se incluye una caja de *login*. Cuando el usuario está logueado, el contenido de esa caja muestra el nombre del usuario y varios enlaces para ver su perfil y para desconectarse de la aplicación.

Así que si has utilizado el código mostrado en las secciones anteriores, cuando un usuario se conecte a la aplicación, sus datos se guardarán en la caché y se mostrarán a cualquier otro usuario que solicite la portada. Para evitarlo, debes hacer uso de la tecnología ESI.

20.4 ESI

En las aplicaciones web reales, es casi imposible que puedas guardar páginas enteras en la caché. Normalmente, algunos *trozos* de las páginas contienen información privada del usuario (como su nombre o algún dato de su perfil). Por tanto, esas partes no se pueden guardar en la caché pública, sino en una caché privada.

Otro caso muy habitual es que algunas partes de la página se puedan guardar mucho tiempo en la caché (porque varían muy poco) pero otras partes tengan un tiempo de vida mucho menos, quizás de tan solo unos pocos segundos.

La tecnología que permite solucionar estos escenarios y muchos otros se denomina ESI, del inglés *Edge Side Includes*. Gracias a ESI puedes establecer diferentes estrategias de caché para diferentes partes de una misma página.

Técnicamente, ESI es un lenguaje de marcado muy sencillo basado en XML y que se utiliza para construir los contenidos de los sitios web muy dinámicos y así mejorar su escalabilidad. La especificación oficial del lenguaje ESI (http://www.w3.org/TR/esi-lang) ha sido creada por empresas como Akamai, Oracle, BEA e Interwoven y está publicada en el W3C.

ESI permite definir diferentes estrategias de caché para diferentes fragmentos de una misma página. Su funcionamiento se basa en construir dinámicamente el contenido completo de la página mediante las diferentes etiquetas `<esi>` incluidas en su código.

Siguiendo con el ejemplo de la portada del sitio web, todos sus contenidos son cacheables públicamente salvo la caja de login del lateral. Si el contenido HTML original de la página es el siguiente:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, cada día ofertas increíbles ...</title>
    <link href="/css/frontend.css" rel="stylesheet" type="text/css" />
    <link rel="shortcut icon" href="/favicon.png" />
</head>

<body id="portada">
    <!-- ... -->

    <section id="login">
        <p>Conectado como <strong>Nombre Apellido1 Apellido2</strong></p>

        <a href="/app.php/es/usuario/perfil">Ver mi perfil</a>
        <a href="/app.php/es/usuario/logout">Cerrar sesión</a>
    </section>

    <!-- ... -->
</body>
</html>
```

Si utilizas ESI, el código HTML anterior se convierte en lo siguiente:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, cada día ofertas increíbles ...</title>
    <link href="/css/frontend.css" rel="stylesheet" type="text/css" />
    <link rel="shortcut icon" href="/favicon.png" />
</head>

<body id="portada">
    <!-- ... -->

    <section id="login">
        <esi:include src="/app.php/AppBundle/Default/cajaLogin.php" />
    </section>

    <!-- ... -->
</body>
</html>
```

Los *reverse proxies* compatibles con ESI procesan todas las etiquetas de tipo `<esi>` antes de servir la página cacheada. Concretamente, la etiqueta `<esi:include>` le indica al *proxy* que debe obtener el recurso indicado en su atributo `src` y el resultado obtenido debe insertarlo en ese mismo lugar antes de servir la página al usuario.

Gracias a ESI puedes aprovechar la naturaleza estática de la mayor parte de los contenidos de las páginas, pero manteniendo la naturaleza dinámica de alguna de sus partes.

NOTA Las URL mostradas en estos primeros ejemplos no son reales, sino que se han preparado con una finalidad didáctica. Más adelante se muestra un ejemplo de URL real utilizada por ESI en Symfony.

Esta etiqueta permite incluso definir un contenido alternativo (atributo `alt`) por si el contenido de `src` no está disponible. También permite indicar (atributo `onerror`) qué hacer cuando se produce un error al obtener cualquiera de esos contenidos:

```
<!-- ... -->

<section id="login">
    <esi:include src="/app.php/AppBundle/Default/cajaLogin.php"
                 alt="/app.php/OfertaBundle/Estaticas/anonimo.html"
                 onerror="continue" />
</section>

<!-- ... -->
```

Symfony simplifica al máximo el uso de ESI en sus aplicaciones ya que ni siquiera hace falta escribir ninguna etiqueta `<esi>`. Con unos pequeños cambios en las plantillas de la aplicación puedes hacer que tu aplicación sea compatible con ESI. Además, cuentas con la ventaja de que si dejas de utilizar un *reverse proxy*, Symfony se encarga de modificar automáticamente las plantillas para servir los contenidos sin etiquetas `<esi>`.

20.4.1 Configurando ESI

El soporte de ESI no se encuentra activado por defecto en Symfony. Así que el primer paso consiste en abrir el archivo de configuración `config.yml` de la aplicación y establecer a `true` el parámetro `enabled` de la opción `esi`:

```
# app/config/config.yml
# ...
framework:
    esi: { enabled: true }
```

Si recuerdas el ejemplo anterior de la etiqueta `<esi:include>`, los *trozos* especiales de la página se sirven mediante llamadas a la aplicación, por lo que Symfony ya incluye la ruta y el controlador necesarios para procesar las páginas con ESI. Así que simplemente cambiando esta opción de configuración, ya puedes utilizar ESI en tu aplicación Symfony.

20.4.2 Refactorizando las plantillas

Todas las plantillas del *frontend* incluyen en su lateral la caja de login mediante la plantilla `frontend.html.twig` de la que heredan. Así que para añadir el soporte de ESI en la aplicación solo es necesario modificar esa plantilla base.

La caja de login se muestra mediante la función `render()` de Twig, que incluye en la página el resultado de la ejecución de una determinada acción:

```
{# app/Resources/views/frontend.html.twig #}
{% extends 'base.html.twig' %}

{# ... #}

{% block body %}
<header>
    {# ... #}
</header>

<article>
    {% block article %}{% endblock %}
</article>

<aside>
    {% block aside %}
    <section id="login">
    {{ render(controller('usuario/_caja_login.html.twig', { id: block('id') }))
}}
```

```
    </section>
    {% endblock %}
</aside>
{% endblock %}
```

Para permitir que la caja de login pueda definir su propia estrategia de caché independiente de la página, reemplaza la función `render()` de Twig por `render_esi()`:

```
{# app/Resources/views/frontend.html.twig #}

{# ... #}

{% block aside %}
<section id="login">
{{ render_esi(controller('usuario/_caja_login.html.twig')) }}
</section>
{% endblock %}
```

NOTA La función `render_esi()` en realidad es un atajo de la función `render()`. Si no quieres modificar las funciones `render()` de las plantillas, utiliza en su lugar la opción `strategy`:

```
{# atajo para mostrar plantillas con ESI #}
{{ render_esi(controller('...', { ... })) }}

{# mostrando plantillas con ESI y la función render() #}
{{ render(
  controller('...', { ... }), { strategy: 'esi' }
) }}
```

Aunque utilices la función `render_esi()` en tus plantillas, si Symfony detecta que no hay un *reverse proxy* o que no es compatible con ESI, incluye directamente en la plantilla el contenido devuelto por la acción y no se produce ningún error.

Si sientes curiosidad, puedes acceder al directorio `http_cache/` dentro de la caché de un entorno de ejecución (`app/cache/dev/` o `app/cache/prod/`) para ver el código fuente de las páginas de la caché. Así comprobarás cómo realiza Symfony las llamadas internas para procesar los elementos ESI:

```
<!-- app/cache/prod/http_cache/../../../../9f2a..35ec -->
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, cada día ofertas increíbles en tu ciudad ...</title>
```

```
<!-- ... -->
<section id="login">
<?php echo $this->esi->handle($this, '/app.php/_internal/AppBundle%3AUsuario%3Ac
ajaLogin/id%3Dportada.html', '', true) ?>
</section>
<!-- ... -->
```

Por último, indica en la acción `cajaLogin()` del *bundle* `AppBundle` que se trata de un contenido privado que no debe guardarse en la caché pública. Puedes añadir también la cabecera `Cache-Control` con la opción `max-age` para que los contenidos de la caja de login se guarden en la caché durante unos segundos:

```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/{ciudad}", defaults={ "ciudad" = "%app.ciudad_por_defecto%" }, n
ame="portada")
     * @Cache(smaxage="60")
     */
    public function portadaAction($ciudad)
    {
        // ...
    }

    // ...
}
```

```
// src/AppBundle/Controller/UsuarioController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class UsuarioController extends Controller
{
    /**
     * @Cache(maxage="30")
     */
    public function cajaLoginAction()
    {
        // ...
    }

    // ...
}
```

Este es el último cambio necesario para añadir soporte de ESI en la aplicación. Ahora la portada se sirve desde la caché pública, pero la caja de login se obtiene para cada usuario. Así la aplicación sigue manteniendo un gran rendimiento, pero sigue siendo dinámica y segura, ya que los datos privados de los usuarios ya no se muestran a cualquier usuario que acceda al sitio web.

TRUCO Una forma sencilla de comprobar si ESI está funcionando tal como se desea es añadir el código `'now'|date('H:i:s')` en diferentes partes de la plantilla. Si no utilizas cachés, cada vez que accedes a la página cambia la hora de todos los relojes. Si utilizas ESI, cada reloj se actualizará con un ritmo diferente y permanecerá fijo tanto tiempo como permanezca la página en la caché.

Symfony también incluye soporte de las opciones de ESI que permiten mejorar la experiencia de usuario cuando se produce un error. La opción `alt` por ejemplo indica el controlador alternativo que se ejecuta cuando el controlador indicado en la función `render()` no está disponible o produce algún error:

```
<section id="login">
{{ render_esi(
    controller('AppBundle:Usuario:estoNoExiste'),
    { alt: 'AppBundle:Usuario:anonimo' }
) }}
</section>
```

En el código anterior, el controlador de la función `render()` hace referencia a una acción que no existe. Sin embargo, como se ha definido la opción `alt`, la aplicación no sólo no muestra un error sino que ejecuta la acción `anonimoAction()` del controlador `UsuarioController`.

Symfony permite ir un paso más allá en el tratamiento de los errores. Si el controlador alternativo no existe o produce algún error, la aplicación mostrará ese error. Para evitarlo, añade la opción `ignore_errors: true` y Symfony ignorará los errores de forma silencionsa, no mostrando nada en el lugar donde se encuentra la llamada a la función `render()`:

```
<section id="login">
{{ render_esi(
    controller('AppBundle:Usuario:estoNoExiste'),
    {
        alt: 'AppBundle:Usuario:anonimo',
        ignore_errors: true
    }
) }}
</section>
```

20.4.3 Variando las respuestas

La enorme variedad y disparidad de navegadores disponibles en el mercado hace que no sea suficiente con todo lo explicado en las secciones anteriores. En la caché se guarda una página por cada URL de la aplicación.

El problema se puede producir si un usuario solicita una página y su navegador, como la mayoría, soporta las respuestas comprimidas (indicado por ejemplo con la cabecera `Accept-Encoding: gzip, deflate`). Symfony genera la página y el *reverse proxy* la entrega y guarda comprimida. Si a continuación un usuario solicita la misma página y su navegador no soporta la compresión utilizada por el *proxy*, se producirá un error porque el *proxy* sólo sabe que a una determinada URL le corresponde una determinada página en la caché.

La solución consiste en guardar diferentes versiones de una misma página cacheada, cada una de ellas adaptada a una característica (o carencia) de los navegadores. Para ello se utiliza la cabecera `Vary` de HTTP, que indica de qué cabeceras de la petición del navegador depende la respuesta del servidor.

Si sólo quieres incluir soporte para las diferentes compresiones disponibles, añade el siguiente método `setVary()` en el objeto `Response`:

```
public function portadaAction($ciudad)
{
    // ...

    $respuesta = $this->render( ... );
    $respuesta->setSharedMaxAge(60);
    $respuesta->setVary('Accept-Encoding');

    return $respuesta;
}
```

El método `setVary()` también acepta como argumento un array para indicar más de una cabecera:

```
public function portadaAction($ciudad)
{
    // ...

    $respuesta = $this->render( ... );
    $respuesta->setSharedMaxAge(60);
    $respuesta->setVary(array('Accept-Encoding', 'Host'));
    // $respuesta->setVary(array('Accept-Encoding', 'User-Agent'));
    // $respuesta->setVary(array('Accept-Encoding', 'User-Agent', 'Host'));

    return $respuesta;
}
```

Sección 5

Apéndices

Esta página se ha dejado vacía a propósito

APÉNDICE A

El motor de plantillas Twig

Twig es un motor y lenguaje de plantillas para PHP que es rápido, eficiente y seguro. Symfony recomienda utilizar Twig para crear todas las plantillas, aunque si lo prefieres también puedes crearlas con PHP.

La sintaxis de Twig se ha diseñado para que las plantillas sean concisas y muy fáciles de leer y de escribir. Observa por ejemplo el siguiente código de una plantilla Twig (aunque nunca hayas utilizado Twig, es muy posible que entiendas perfectamente su funcionamiento):

```
{% if usuario is defined %}
    Hola {{ usuario.nombre }}
    hoy es {{ 'today'|date('d/m/Y') }}
{% endif %}
```

Observa ahora el código PHP equivalente al código Twig anterior:

```
<?php if (isset($usuario)): ?>
    Hola <?php echo htmlspecialchars($usuario->getNombre(), ENT_QUOTES, 'UT
F-8') ?>
    hoy es <?php $hoy = new \DateTime(); echo $hoy->format('d/m/Y');  ?>
<?php endif; ?>
```

¿Entiendes ahora por qué la mayoría de programadores que conocen Twig ya no vuelven a utilizar PHP para crear sus plantillas? Además de ser mucho más *limpias* y concisas, las plantillas de Twig son seguras por defecto, por lo que no debes aplicar el mecanismo de escape al valor de las variables (función `htmlspecialchars()`). Además, al ejecutar la aplicación, las plantillas de Twig se *compilan* a código PHP nativo, por lo que el rendimiento y el consumo de memoria es similar al de las plantillas PHP.

La mejor referencia para aprender Twig es su documentación oficial, que puedes encontrar en http://twig.sensiolabs.org/documentation. Los contenidos de este apéndice resumen las partes esenciales de esa documentación.

A.1 Sintaxis básica

Las plantillas de las aplicaciones web suelen utilizar un lenguaje para crear los contenidos (HTML, XML, JavaScript) y otro lenguaje para añadir la lógica dentro de las plantillas (Twig, PHP).

Para separar uno de otro, los lenguajes de programación definen etiquetas especiales. PHP por ejemplo define las etiquetas `<?php` y `?>` para delimitar su código dentro de una plantilla. Igualmente, Twig define tres etiquetas para distinguir el código Twig del resto de contenidos:

- `{{` y `}}` para mostrar el valor de una variable.
- `{%` y `%}` para añadir lógica en la plantilla.
- `{#` y `#}` para incluir un comentario.

A.2 Twig para diseñadores

El objetivo de Twig es conseguir que los maquetadores y diseñadores sin conocimientos de programación sean capaces de crear todas las plantillas de la aplicación de forma autónoma, sin ayuda de los programadores. De esta forma se acelera el desarrollo de las aplicaciones y se mejora la productividad.

Por eso Twig ha sido ideado para que sea realmente fácil de aprender, leer y escribir por parte de profesionales sin un perfil técnico avanzado. Esta primera sección explica todos los conocimientos básicos imprescindibles para los maquetadores. La siguiente sección, ideada para programadores, explica las características más avanzadas de Twig.

A.2.1 Mostrar información

Las páginas HTML que se envían a los usuarios normalmente se generan de forma dinámica a partir de plantillas. Para rellenar de contenido las páginas, las plantillas obtienen la información a través de las variables.

Para mostrar el contenido de una variable en la plantilla, escribe su nombre encerrado entre dos pares de llaves: `{{ nombre-de-la-variable }}`. El siguiente código indica cómo mostrar el valor de tres variables:

```
<p>Hola {{ nombre }}. Tienes {{ edad }} años
y vives en {{ ciudad }}</p>
```

Una misma variable puede contener muchas propiedades diferentes. En ese caso, puedes mostrar cada propiedad con la notación: `{{ variable.propiedad }}`. Imagina que en el ejemplo anterior todos los datos del usuario se guardan en una variable llamada `usuario`. Para mostrar la información, deberías modificar el código por lo siguiente:

```
<p>Hola {{ usuario.nombre }}. Tienes {{ usuario.edad }} años
y vives en {{ usuario.ciudad }}</p>
```

Utilizar una u otra forma de mostrar información es indiferente para Twig, pero la segunda suele producir plantillas más legibles. En cualquier caso, los programadores con los que trabajes te informarán sobre la forma de obtener la información de la aplicación.

A.2.2 Modificar información

Modificar la información antes de mostrarla es muy común en las plantillas de las aplicaciones. Imagina que quieres mostrar la descripción de un producto en el sitio web de una tienda de comercio electrónico. Lo más fácil sería escribir simplemente `{{ producto.descripcion }}`.

Sin embargo, si la descripción contiene etiquetas HTML, podría interferir con el propio diseño de la página. Así que para evitar estos problemas, lo mejor es eliminar todas las etiquetas HTML que pueda contener la descripción. En Twig la información se modifica mediante filtros, utilizando la siguiente sintaxis:

```
{{ producto.descripcion|striptags }}
```

La palabra `striptags` es el nombre del filtro que se aplica al contenido de la variable antes de mostrarla. El filtro `striptags` elimina cualquier etiqueta HTML que contenga la variable y es uno de los muchos filtros que ya incluye Twig, tal y como se explicará más adelante.

Los filtros siempre se escriben detrás del nombre de la variable y separados por el carácter `|` (que es la barra vertical que se obtiene al pulsar la tecla `Alt.` junto con la tecla del número `1`). Normalmente no se dejan espacios en blanco entre la variable y el filtro, pero puedes añadir tantos como quieras porque Twig los ignora todos:

```
{# las siguientes instrucciones son equivalentes #}
{{ producto.descripcion|striptags }}
{{ producto.descripcion | striptags }}
{{ producto.descripcion   |striptags }}
{{ producto.descripcion|   striptags }}
{{ producto.descripcion   |   striptags }}
```

El siguiente ejemplo utiliza el filtro `upper` (del inglés, *uppercase*) para mostrar el contenido de una variable en letras mayúsculas:

```
{{ articulo.titular|upper }}
```

Todos los filtros de Symfony se pueden encadenar para aplicarlos en cascada. El siguiente ejemplo elimina todas las posibles etiquetas HTML del titular de un artículo y después convierte su contenido a mayúsculas:

```
{{ articulo.titular|striptags|upper }}
```

El orden en el que escribes los filtros es muy importante, ya que Twig los aplica siempre ordenadamente empezando desde la izquierda.

Algunos filtros permiten modificar su comportamiento pasándoles información adicional entre paréntesis. El filtro `join` se emplea para unir los elementos de una lista:

```
{{ producto.etiquetas|join }}
```

Sin embargo, por defecto `join` une todos los elementos sin dejar ningún espacio en blanco entre ellos. Para añadir ese espacio en blanco, indícalo entre paréntesis al añadir el filtro:

```
{{ producto.etiquetas|join(' ') }}
```

De todos los filtros que incluye Twig, a continuación se explican los más útiles para los maquetadores:

`date`, muestra una fecha con el formato indicado. Las variables utilizadas para indicar el formato son las mismas que las de la función date() de PHP.

```
{# Si hoy fuese 21 de julio de 2016, mostraría '21/7/2016' #}
{{ 'today'|date('d/m/Y') }}

{# Si además fuesen las 18:30:22, mostraría '21/7/2016 18:30:22' #}
{{ 'now'|date('d/m/Y H:i:s') }}

{# También se puede aplicar sobre variables #}
{# Si no se indica el formato, se muestra como 'July 21, 2016 18:30' #}
{{ oferta.fechaExpiracion|date }}
```

`striptags`, elimina todas las etiquetas HTML y XML del contenido de la variable. También reemplaza dos o más espacios en blanco por uno solo.

```
{{ '<strong>Lorem ipsum</strong> dolor sit amet'|striptags }}
{# Muestra 'Lorem ipsum dolor sit amet' #}
```

`default`, permite asignar un valor a las variables que no existen o están vacías.

```
{{ descripcion|default('Este producto todavía no tiene una descripción') }}
```

Si la descripción existe y no está vacía, se muestra su contenido. Si no, se muestra el mensaje *"Este producto todavía no tiene una descripción"*

`nl2br`, transforma los saltos de línea en elementos `<br/>`.

```
{# 'descripcion' es igual a:
     Esta es la descripción
     corta del producto en
     varias líneas.
#}

{{ descripcion|nl2br }}

{# Muestra:
     Esta es la descripción <br/>
     corta del producto en <br/>
     varias líneas.
#}
```

`upper`, transforma el contenido de la variable a mayúsculas.

```
{# Muestra 'INFORMACIÓN DE CONTACTO' #}
{{ 'Información de Contacto'|upper }}
```

`lower`, transforma el contenido de la variable a minúsculas.

```
{# Muestra 'menú' #}
{{ 'Menú' | lower }}
```

`capitalize`, transforma la primera letra del texto a mayúsculas y el resto de letras a minúsculas.

```
{# Muestra 'Los precios no incluyen iva' #}
{{ 'Los precios NO incluyen IVA' | capitalize }}
```

`title`, transforma la primera letra de cada palabra a mayúsculas y el resto de letras a minúsculas.

```
{# Muestra 'Los Precios No Incluyen Iva' #}
{{ 'Los precios NO incluyen IVA' | title }}
```

`trim`, elimina los espacios en blanco del principio y del final.

```
{# Muestra 'Descripción del producto escrita por el usuario.' #}
{{ '  Descripción del producto escrita por el usuario.   '|trim }}
```

Este filtro también permite indicar entre paréntesis el carácter o caracteres que quieres eliminar. Esta característica te puede servir por ejemplo para eliminar el punto del final en las frases para las que no quieres mostrarlo:

```
{# Muestra 'Descripción del producto escrita por el usuario.' #}
{{ '  Descripción del producto escrita por el usuario.  '|trim }}

{# Muestra '  Descripción del producto escrita por el usuario.  ' #}
{{ '  Descripción del producto escrita por el usuario.  '|trim('.') }}

{# Muestra 'Descripción del producto escrita por el usuario' #}
{{ '  Descripción del producto escrita por el usuario.  '|trim('. ') }}
```

`number_format`, modifica la forma en la que se muestran los números con decimales:

```
{# si precio = 19,95 se muestra 19.95 #}
{{ precio }}

{# si precio = 19,9 se muestra 19.9 #}
{{ precio }}

{# si precio = 19,95 se muestra 19,950 #}
{{ precio | number_format(3, ',', '.') }}

{# si precio = 19,9 se muestra 19.90 #}
{{ precio | number_format(2, '.', ',') }}
```

`join`, crea una cadena de texto concatenando todos los valores de la colección de elementos sobre la que se aplica el filtro.

```
{# La variable meses contiene los valores ['Enero', 'Febrero', 'Marzo'] #}
{{ meses|join }}
```

```
{# en la página se muestra: 'EneroFebreroMarzo' #}
```

Como casi siempre es necesario separar los elementos que se unen, el filtro `join` permite indicar entre paréntesis el carácter o caracteres que se utilizan para unir los elementos:

```
{# Muestra 'Enero Febrero Marzo' #}
{{ meses|join(' ') }}

{# Muestra 'Enero, Febrero, Marzo' #}
{{ meses|join(', ') }}

{# Muestra 'Enero - Febrero - Marzo' #}
{{ meses|join(' - ') }}
```

La sección *Twig para programadores* (página 457) muestra otros filtros más avanzados que también incluye Twig.

A.2.3 Mecanismo de escape

Si intentas mostrar en una plantilla el contenido de una variable que incluye etiquetas HTML, puede que el resultado obtenido no sea lo que esperabas. Imagina que un producto dispone de la siguiente descripción:

```
<strong>Lorem ipsum</strong> dolor site <em>amet</em>.
```

Si ahora incluyes en una plantilla el código `{{ producto.descripcion }}` para mostrar por pantalla la descripción, Twig mostrará lo siguiente:

```
&lt;strong&gt;Lorem ipsum&lt;/strong&gt; dolor site &lt;em&gt;amet&lt;/em&gt;.
```

Para evitar que el contenido mal formado de una variable pueda *romper* la página y para evitar potenciales problemas de seguridad, Twig por defecto no permite mostrar etiquetas HTML y por eso modifica el contenido de todas las variables aplicando lo que se denomina el *mecanismo de escape*.

Aunque puede resultarte extraño, este comportamiento por defecto de Twig es el más correcto y te evitará muchos problemas en tus plantillas. Para no aplicar el mecanismo de escape en una determinada variable, utiliza el filtro `raw`:

```
{{ producto.descripcion|raw }}
```

El filtro `raw` ordena a Twig que muestre el contenido original de la variable, contenga lo que contenga, sin realizar ninguna modificación. Por tanto, el resultado del código anterior será que la plantilla muestra el contenido `<strong>Lorem ipsum</strong> dolor site <em>amet</em>.` original.

A.2.4 Espacios en blanco

Cuando Twig crea una página a partir de una plantilla, mantiene todos los espacios en blanco (tabuladores, nuevas líneas, espacios) que contenga la plantilla. Este comportamiento de Twig es el más apropiado en la mayoría de los casos, pero se puede modificar.

Imagina que has escrito el siguiente código HTML lleno de espacios para mejorar su legibilidad:

```
<ul>
  <li>
    <a ...>XXX</a>
  </li>

  <li>
    ...
```

Twig dispone de una etiqueta especial llamada `{% spaceless %}` que elimina todos los espacios en blanco del código que encierra. Si modificas el ejemplo anterior por lo siguiente:

```
{% spaceless %}
<ul>
  <li>
    <a ...>XXX</a>
  </li>

  <li>
    ...
{% endspaceless %}
```

Al generar una página a partir de la plantilla anterior, Twig incluye el siguiente código, sin ningún espacio en blanco:

```
<ul><li><a ...>XXX</a></li><li>...
```

A.3 Twig para programadores

A.3.1 Variables

Mostrar el valor de una variable en una plantilla Twig es tan sencillo como encerrar su nombre entre dos pares de llaves: `{{ nombre-de-la-variable }}`. No obstante, en las aplicaciones web reales suele ser habitual utilizar la notación `{{ variable.propiedad }}`.

Twig es tan flexible que esta última notación funciona tanto si tus variables son objetos como si son arrays y tanto si sus propiedades son públicas o si se acceden mediante *getters*. En concreto, la expresión `{{ variable.propiedad }}` hace que Twig busque el valor de la propiedad utilizando las siguientes instrucciones y en el siguiente orden:

1. `$variable["propiedad"]`
2. `$variable->propiedad`
3. `$variable->propiedad()`
4. `$variable->getPropiedad()`
5. `$variable->isPropiedad()`
6. Si ninguna de las instrucciones anteriores funciona, se devuelve `null`

Como en las aplicaciones Symfony es habitual trabajar con objetos que representan entidades de Doctrine, los objetos están llenos de *getters* y *setters*. Así que Twig casi siempre encuentra el valor de las propiedades con `$variable->getPropiedad()`.

Además de la notación `{{ variable.propiedad }}`, puedes utilizar la notación alternativa `{{ variable["propiedad"] }}`. En este último caso, Twig sólo comprueba si existe un array llamado `variable` con una clave llamada `propiedad`. Si no existe, devuelve el valor `null`.

NOTA La lógica que utiliza internamente Twig para determinar el valor de la expresión `{{ variable.propiedad }}` es el principal *cuello de botella* de su rendimiento. Como no es posible mejorarlo con código PHP, **a partir de la versión 1.4** Twig incluye una extensión de PHP programada en C para mejorar muy significativamente el rendimiento de esta parte.

Además de las variables que se pasan a la plantilla, puedes crear nuevas variables con la etiqueta `set`

```
{% set variable = valor %}
```

Las variables de Twig pueden ser de tipo numérico, *booleano*, array y cadena de texto:

```
{# Cadena de texto #}
{% set nombre = 'José García' %}

{# Valores numéricos #}
{% set edad = 27 %}
{% set precio = 104.83 %}

{# Valores booleanos #}
{% set conectado = false %}

{# Arrays normales #}
{% set tasaImpuestos = [4, 8, 18] %}

{# Arrays asociativos #}
{% set direcciones = { publica: 'http://...', privada: 'http://...' } %}

{# Array asociativo que combina todos los valores anteriores #}
{% set perfil = {
    nombre:    'José García',
    perfiles: ['usuario', 'administrador'],
    edad:      27,
    validado: true
} %}
```

Las cadenas de texto se encierran entre comillas simples o dobles. Los números y los valores *booleanos* se indican directamente. Los arrays *normales* se encierran entre corchetes (`[` y `]`) y los arrays asociativos o *hashes* entre llaves (`{` y `}`).

Twig también permite crear e inicializar más de una variable a la vez. Para ello, escribe varias variables separadas por comas y define el mismo número de valores después del símbolo `=`:

```
{% set variable1, variable2 = valor1, valor2 %}

{% set nombre, edad, activado = 'José García', 27, true %}
```

Para concatenar variables entre sí o con otros valores, utiliza el operador `~`:

```
{% set nombreCompleto = nombre ~ ' ' ~ apellidos %}
{% set experiencia = edad ~ ' años' %}
```

Si necesitas definir una variable muy compleja concatenando muchos valores diferentes, es más conveniente utilizar la etiqueta `set` de la siguiente manera:

```
{% set perfil %}
  {{ usuario.apellidos }}, {{ usuario.nombre }}
  {{ usuario.edad }} años
  Página: {{ usuario.url }}
{% endset %}
```

El problema de la notación `{{ variable.propiedad }}` utilizada por Twig es que el nombre de la propiedad no puede ser variable. Por eso, Twig también incluye la función `attribute()` para obtener el valor de propiedades cuyo nombre es variable:

```
{# Los dos siguientes ejemplos son equivalentes #}
{{ oferta.descripcion }}

{% set propiedad = 'descripcion' %}
{{ attribute(oferta, propiedad) }}
```

El siguiente ejemplo almacena la forma de contacto preferida del usuario en una variable. Así se obtiene el contacto de cualquier usuario con una sola instrucción gracias a la función `attribute()`:

```
{% set usuario1 = { email: '...', movil: '...', contacto: 'email' } %}
{% set usuario2 = { email: '...', movil: '...', contacto: 'movil' } %}

{# Se muestra el email del usuario1 y el móvil del usuario2 #}
Forma de contacto de usuario 1 {{ attribute(usuario1, usuario1.contacto) }}
Forma de contacto de usuario 2 {{ attribute(usuario2, usuario2.contacto) }}
```

El segundo parámetro de la función `attribute()` también puede ser el nombre del método de un objeto. En este caso, también se puede utilizar un tercer parámetro para indicar el valor de los argumentos que se pasan al método.

A.3.2 Espacios en blanco

Además de la etiqueta `{% spaceless %}`, es posible controlar el tratamiento de los espacios en blanco a nivel de cada variable. Para ello se añade el operador `-` (guión medio) en el lado por el que quieres eliminar los espacios en blanco:

Operador de Twig	Equivalente PHP	Resultado
`{{- variable }}`	`ltrim(variable)`	Elimina los espacios del lado izquierdo
`{{ variable -}}`	`rtrim(variable)`	Elimina los espacios del lado derecho
`{{- variable -}}`	`trim(variable)`	Elimina todos los espacios que rodean al valor de la variable

A.3.3 Filtros

La forma estándar de indicar los filtros (`{{ variable|striptags|upper }}`) no es cómoda cuando se quieren aplicar los mismos filtros al contenido de muchas variables. En este caso, es mejor hacer uso de la etiqueta `filter`:

```
{% filter title|nl2br %}
  <h1>{{ oferta.titulo }}</h1>
  <p>{{ oferta.descripcion }}</p>
  <a href="...">comprar</a>
{% endfilter %}
```

Los filtros indicados en la etiqueta `{% filter %}` se aplican a todos los contenidos de su interior, no sólo a las variables o expresiones de Twig. Por tanto, en el ejemplo anterior el texto *comprar* se muestra como *Comprar*.

Twig ya incluye los filtros más comúnmente utilizados al crear las plantillas, aunque también puedes crear cualquier otro filtro que necesites. Además de los filtros básicos explicados en las secciones anteriores, Twig incluye los siguientes filtros avanzados.

`format()`, similar a la función printf() de PHP, ya que formatea una cadena de texto sustituyendo sus variables por los valores indicados:

```
{# Muestra: "Hola José, tienes 56 puntos" #}
{{ "Hola %s, tienes %d puntos"|format('José', 56) }}

{# También se pueden utilizar variables en el filtro #}
{% set puntos = 56 %}
{{ "Hola %s, tienes %d puntos"|format('José', puntos) }}
```

`replace()`, muy similar al filtro format() pero el formato de las variables de la cadena de texto se puede elegir libremente:

```
{{ "Hola #nombre#, tienes #puntuacion# puntos"|replace({
    '#nombre#': 'José',
    '#puntuacion#': '56'
}) }}
```

`reverse`, invierte el orden de los elementos de un array o de un objeto que implemente la interfaz Iterator:

```
{% set clasificacion = { 'Equipo2': 35, 'Equipo4': 32, 'Equipo1': 28 } %}
{% set losPeores = clasificacion|reverse %}
```

```
{# losPeores = { 'Equipo1': 28, 'Equipo4': 32, 'Equipo2': 35 } #}
```

length, devuelve el número de elementos de un array, colección o secuencia. Si es una cadena de texto, devuelve el número de letras:

```
{# 'ofertas' es una variable que se pasa a la plantilla #}
Se han encontrado {{ ofertas|length }} ofertas

{% set abecedario = 'a'..'z' %}
El abecedario en inglés tiene {{ abecedario|length }} letras

{% set longitud = 'anticonstitucionalmente'|length %}
La palabra más larga en español tiene {{ longitud }} letras
```

slice, extrae *un trozo* de una colección o de una cadena de texto.

```
{% set clasificacion = { 'Equipo1', 'Equipo5', 'Equipo2', 'Equipo4', 'Equipo3'
} %}

{# si se pasan dos parámetros:
     * el primero indica la posición donde empieza el trozo
     * el segundo indica el número de elementos que se cogen
#}

{# se queda sólo con el primer elemento #}
{% set ganador = clasificacion|slice(1, 1) %}

{# se queda con los tres primeros elementos #}
{% set podio   = clasificacion|slice(1, 3) %}

{# se queda sólo con el elemento que se encuentra en la
   quinta posición #}
{% set ultimo  = clasificacion|slice(5, 1) %}

{# si se pasa un parámetro:
     * si es positivo, el trozo empieza en esa posición
       y llega hasta el final
     * si es negativo, el trozo empieza en esa posición
       contada desde el final de la colección
#}

{# se queda con todos los elementos a partir de la segunda posición #}
{% set perdedores = clasificacion|slice(2) %}

{# sólo se queda con el último elemento de la colección #}
{% set ultimo = clasificacion|slice(-1) %}
```

Internamente este filtro funciona sobre los arrays y colecciones como la función `array_slice` de PHP y sobre las cadenas de texto como la función `substr` de PHP.

`sort`, ordena los elementos de un array aplicando la función `asort()` de PHP, por lo que se mantienen los índices en los arrays asociativos:

```
{% set contactos = [
    { 'nombre': 'María',   'apellidos' : '...' },
    { 'nombre': 'Alberto', 'apellidos' : '...' },
    { 'nombre': 'José',    'apellidos' : '...' },
] %}

{% for contacto in contactos|sort %}
    {{ contacto.nombre }}
{% endfor %}
{# Se muestran en este orden: Alberto, José, María #}

{% set ciudades = ['Paris', 'Londres', 'Tokio', 'Nueva York'] %}
{% set ordenadas = ciudades|sort %}
{# ordenadas = ['Londres', 'Nueva York', 'Paris', 'Tokio'] #}
```

`merge`, combina el array que se indica como parámetro con el array sobre el que se aplica el filtro:

```
{% set documentos = ['DOC', 'PDF'] %}
{% set imagenes = ['PNG', 'JPG', 'GIF'] %}
{% set videos = ['AVI', 'FLV', 'MOV'] %}

{% set formatos = documentos|merge(imagenes)|merge(videos)  %}
{# formatos = ['DOC', 'PDF', 'PNG', 'JPG', 'GIF', 'AVI', 'FLV', 'MOV'] #}
```

`json_encode`, codifica el contenido de la variable según la notación JSON. Internamente utiliza la función `json_encode()` de PHP, por lo que es ideal en las plantillas de las aplicaciones AJAX y JavaScript.

```
{% set perfil = {
    nombre: 'José García',
    edad:   27,
    emails: ['email1@localhost', 'email2@localhost']
} %}

{{ perfil|json_encode }}
{# Muestra: {"nombre":"José García", "edad":27, "emails":["email1@localhost", "e
mail2@localhost"]} #}
```

`url_encode`, codifica el contenido de la variable para poder incluirlo de forma segura como parte de una URL. Internamente utiliza la función `urlencode()` de PHP.

```
{% set consulta = 'ciudad=paris&orden=ascendente&límite=10' %}
{{ consulta|url_encode }}
{# Muestra: ciudad%3Dparis%26orden%3Dascendente%26l%C3%ADmite%3D10 #}
```

`convert_encoding`, transforma una cadena de texto a la codificación indicada. Este filtro requiere que esté activada o la extensión `iconv` o la extensión `mbstring` de PHP:

```
{{ descripcion|convert_encoding('UTF-8', 'iso-8859-1') }}
```

El primer parámetro es la codificación a la que se convierte la cadena y el segundo parámetro indica su codificación original.

`date_modify`, modifica una fecha sumando o restando una cantidad de tiempo.

```
Tu cuenta de prueba caduca el día:
{{ usuario.fechaAlta|date_modify('+1 week')|date }}
```

El parámetro que se pasa al filtro `date_modify` es cualquier cadena de texto que entienda la función `strtotime` de PHP, por lo que sus posibilidades son casi ilimitadas.

A.3.4 Mecanismo de escape

Como se explicó en la sección *Twig para maquetadores*, Twig aplica por defecto un mecanismo de escape al contenido de todas las variables. Para evitarlo en una variable específica, aplícale el filtro `raw`:

```
{{ variable|raw }}
```

Si utilizas Symfony, puedes controlar el escapado automático de variables con la opción `autoescape` del servicio `twig` en el archivo de configuración `app/config/config.yml`:

```
# app/config/config.yml
twig:
    autoescape: true
```

El valor `true` es su valor por defecto y hace que todas las variables de la plantilla se escapen. Para no aplicar el mecanismo de escape a ninguna variable, utiliza el valor `false`.

Aunque deshabilites el escapado automático de variables, puedes escapar cada variable individualmente mediante el filtro `escape` o `e`:

```
{# Escapando el contenido de una variable #}
{{ variable|escape }}

{# Equivalente al anterior, pero más conciso #}
{{ variable|e }}
```

Dependiendo del contenido que estés generando con la plantilla Twig (una página HTML, un archivo CSS, un archivo JavaScript, etc.) el mecanismo de escape debe ser diferente. Por eso el filtro `escape` permite indicar el tipo de escape aplicado:

```
{{ variable|e('html') }}
{{ variable|e('html_attr') }}
{{ variable|e('js') }}
{{ variable|e('css') }}
{{ variable|e('url') }}
```

A.3.5 Estructura de control for

La estructura de control `for` es un buen ejemplo de cómo Twig combina la facilidad de uso con otras opciones mucho más avanzadas. El uso básico del `for` consiste en iterar sobre todos los elementos de una colección de variables:

```
{% for articulo in articulos %}
    {# ... #}
{% endfor %}
```

Para que el código anterior funcione correctamente, no es obligatorio que la variable `articulos` sea un array. Basta con que la variable sobre la que se itera implemente la interfaz `Traversable` o `Countable`. Si programas aplicaciones con Symfony y Doctrine, las colecciones de objetos que devuelven las búsquedas de Doctrine ya implementan esa interfaz.

Twig también permite iterar sobre rangos definidos dentro del propio bucle gracias al operador `in`:

```
{% for i in [3, 6, 9] %}
    {# ... #}
{% endfor %}
```

Los valores sobre los que itera `in` también se pueden definir mediante secuencias de valores gracias al operador `..`, cuyo funcionamiento es idéntico al de la función `range()` de PHP:

```
{# el bucle itera 11 veces y en cada iteración
   la variable 'i' vale 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 #}
{% for i in 0..10 %}
    {# ... #}
{% endfor %}

{# el bucle itera 26 veces y en cada iteración
   la variable 'i' toma el valor de una letra del alfabeto #}
{% for i in 'a'..'z' %}
    {# ... #}
{% endfor %}
```

Además de la estructura `for` habitual, Twig ha ideado una variante llamada `for ... else`, similar al `if ... else`, y que puede resultar muy útil:

```
{% for articulo in articulos %}
    {# ... #}
{% else %}
    No existen artículos
{% endfor %}
```

Si en el código anterior la variable `articulos` no contiene ningún elemento, en vez de iterarse sobre sus contenidos, se ejecuta directamente el código encerrado por `else`. De esta forma, si una consulta a la base de datos devuelve varios registros, se muestran en forma de listado; pero si la consulta devuelve un resultado vacío, se muestra el mensaje *"No existen artículos"*.

La estructura `for ... else` es un buen ejemplo de las utilidades que incluye Twig para hacer las plantillas más concisas y fáciles de leer y que no están disponibles en PHP.

La estructura de control `for` crea en su interior una variable especial llamada `loop` con la que se puede obtener información sobre cada iteración:

```
{% for articulo in articulos %}
  Artículo número {{ loop.index }}
  Todavía faltan  {{ loop.revindex }} artículos
{% endfor %}
```

Las propiedades disponibles en la variable `loop` son las siguientes:

Propiedad	Contenido
`loop.index`	Número de iteración, siendo `1` la primera (`1`, `2`, `3`, ... `N`)
`loop.index0`	Número de iteración, siendo `0` la primera (`0`, `1`, `2`, ... `N-1`)
`loop.revindex`	Número de iteraciones que faltan, siendo `1` la primera (`N`, `N-1`, `N-2`, ... `1`)
`loop.revindex0`	Número de iteraciones que faltan, siendo `0` la primera (`N-1`, `N-2`, `N-3`, ... `0`)
`loop.first`	`true` si es la primera iteración, `false` en cualquier otro caso
`loop.last`	`true` si es la última iteración, `false` en cualquier otro caso
`loop.length`	Número total de iteraciones

Empleando la variable especial `loop` resulta muy sencillo crear por ejemplo un paginador:

```
{% for pagina in paginas %}
  {% if not loop.first %} <a href="...">Anterior</a> {% endif %}

  {# ... #}

  {% if not loop.last %} <a href="...">Siguiente</a> {% endif %}
{% endfor %}
```

Los bucles `for` también se pueden anidar. En este caso, puedes acceder a la variable `loop` del bucle padre a través de la propiedad `parent`:

```
{% for seccion in secciones %}
  {% for categoria in categorias %}
    Sección número {{ loop.parent.loop.index }}
    Categoría número {{ loop.index }}
  {% endfor %}
{% endfor %}
```

Si en vez de iterar por los elementos de una variable quieres hacerlo por sus claves, utiliza el filtro `keys`:

```
{% for codigo in productos | keys %}
    {# ... #}
{% endfor %}
```

También puedes utilizar el formato alternativo del bucle `for`:

```
{% for codigo, producto in productos %}
    {# ... #}
{% endfor %}
```

La desventaja de los bucles `for` de Twig respecto a los de PHP es que no existen mecanismos para el control de las iteraciones, como `break` (para detener el bucle) o `continue` (para saltar una o más iteraciones).

No obstante, Twig permite filtrar la secuencia sobre la que itera el bucle `for`:

```
{# Iterar sólo sobre las ofertas baratas #}
{% for oferta in ofertas if oferta.precio < 10 %}
    {# ... #}
{% endfor %}

{# Iterar sólo sobre los números impares #}
{% for numero in 1..100 if numero is odd %}
    {# ... #}
{% endfor %}

{# Sólo itera sobre los usuarios que sean amigos #}
{% set usuarios = 1..30 %}
{% set amigos = [12, 29, 34, 55, 67] %}
{% for usuario in usuarios if usuario in amigos %}
    {# ... sólo itera sobre 12 y 29 ... #}
{% endfor %}
```

Twig también incluye dos funciones muy útiles para los bucles `for`: `range()` y `cycle()`. La función `range()`, que internamente utiliza la función `range()` de PHP, es similar al operador `..` que crea secuencias, pero añade un tercer parámetro opcional para controlar el salto entre dos valores consecutivos:

```
{# Itera sobre todas las letras del alfabeto inglés #}
{% for letra in range('a', 'z') %}
    {# a, b, c, ..., x, y, z #}
{% endfor %}

{# Mismo resultado que el código anterior #}
{% for letra in 'a'..'z' %}
    {# a, b, c, ..., x, y, z #}
```

```
{% endfor %}

{# Itera sobre una de cada tres letras del alfabeto inglés #}
{% for letra in range('a', 'z', 3) %}
    {# a, d, g, j, m, p, s, v, y #}
{% endfor %}

{# Itera sólo sobre los números pares #}
{% for numero in range(0, 50, 2) %}
    {# 0, 2, 4, ..., 46, 48, 50 #}
{% endfor %}
```

La función `cycle()` recorre secuencialmente los elementos de un array. Cuando llega al último elemento, vuelve al primero, por lo que el array se puede recorrer infinitamente.

```
{# Añadir 'par' o 'impar' a cada fila de la tabla #}
<table>
    {% for oferta in ofertas %}
    <tr class="{{ cycle(['par', 'impar'], loop.index) }}">
        {# ... #}
    </tr>
    {% endfor %}
</table>
```

Esta plantilla de Twig genera el siguiente código HTML (el número de filas depende del número de elementos de la variable `ofertas`):

```
<table>
    <tr class="impar">
        ...
    </tr>

    <tr class="par">
        ...
    </tr>

    <tr class="impar">
        ...
    </tr>

    <!-- ... -->

    <tr class="par">
        ...
    </tr>
</table>
```

El primer parámetro de la función `cycle()` es el array con los elementos que se recorren cíclicamente. El segundo parámetro indica el número de elemento seleccionado (si es mayor que el

número de elementos, se vuelve a empezar por el primer elemento). Si quieres utilizar el número de iteración para seleccionar el elemento, recuerda que dispones de las variables especiales `loop.index` y `loop.index0`.

A.3.6 Estructura de control if

El uso básico de la estructura de control `if` es similar al de cualquier otro lenguaje de programación:

```
{% if usuario.conectado %}
    {# ... #}
{% endif %}
```

Twig también soporta los modificadores `elseif` y `else`:

```
{% if usuario.conectado %}
    {# ... #}
{% elseif usuario.registrado %}
    {# ... #}
{% else %}
    {# ... #}
{% endif %}
```

Normalmente la estructura `if` se combina con los operadores `is` e `is not` y alguno de los tests definidos por Twig:

```
{% if participantes is divisible by(5) %}
    {# ... #}
{% endif %}

{% if descripcion is not empty %}
    {# ... #}
{% endif %}
```

La siguiente tabla muestra todos los tests que incluye Twig por defecto:

Test	Explicación	Código PHP equivalente
`constant(valor)`	Comprueba si la variable contiene un valor igual a la constante indicada	`constant($valor) === $variable`
`defined`	Comprueba que la variable haya sido definida	`isset($variable)`
`divisible by(numero)`	Comprueba si la variable es divisible por el valor indicado	`0 == $variable % $numero`
`empty`	Comprueba si la variable está vacía	`false === $variable \|\| (empty($variable) && '0' != $variable)`

Test	Explicación	Código PHP equivalente
`even`	Comprueba si la variable es un número par	`0 == $variable % 2`
`odd`	Comprueba si la variable es un número impar	`1 == $variable % 2`
`iterable`	Comprueba si la variable es una colección de valores sobre la que se puede iterar	`$variable instanceof Traversable \|\| is_array($variable)`
`none`	Es un alias del test `null`	
`null`	Comprueba si la variable es `null`	`null === $variable`
`same as(valor)`	Comprueba si una variable es idéntica a otra	`$variable === $valor`

Dentro de la estructura de control `if` también resultan muy útiles los operadores para construir expresiones complejas o para combinar varias expresiones entre sí.

Operadores lógicos

Operador	Explicación
`and`	Devuelve `true` solamente si los dos operandos de la expresión son `true`
	Ejemplo: `{% if usuario.registrado and usuario.edad > 18 %}`
`&&`	Notación alternativa del operador `and`
`or`	Devuelve `true` si alguno de los dos operandos de la expresión son `true`
	Ejemplo: `{% if promocionGlobal or producto.enPromocion %}`
`||`	Notación alternativa del operador `or`
`not`	Devuelve el valor contrario de la expresión evaluada | `{% if not ultimoElemento %}`

Los paréntesis permiten agrupar expresiones complejas:

```
{% if (usuario.registrado and pagina.activa)
      or (usuario.registrado and usuario.primeraVisita)
      or usuario.administrador %}
   ...
```

Operadores de comparación

Operador	Explicación
`==`	Devuelve `true` si los dos operandos son iguales
	Ejemplo: `{% if pedido.tipo == 'urgente' %}`
`!=`	Devuelve `true` si los dos operandos son diferentes
	Ejemplo: `{% if pedido.tipo != 'urgente' %}`

Operador	Explicación
`>`	Devuelve `true` si el primer operando es mayor que el segundo
	Ejemplo: `{% if usuario.edad > 18 %}`
`<`	Devuelve `true` si el primer operando es menor que el segundo
	Ejemplo: `{% if producto.stock < 10 %}`
`>=`	Devuelve `true` si el primer operando es mayor o igual que el segundo
	Ejemplo: `{% if credito >= limite %}`
`<=`	Devuelve `true` si el primer operando es menor o igual que el segundo
	Ejemplo: `{% if producto.stock <= umbral %}`

Operador contenedor

Twig define un operador adicional llamado `in` que comprueba si un valor se encuentra dentro de la colección indicada:

```
{# Devuelve true porque el número 3 se encuentra dentro de la colección #}
{{ 3 in [1, 1, 2, 3, 5, 8, 13] }}

{# Devuelve true si la letra es una vocal #}
{{ letra in 'aeiou' }}

{# Comprueba si una cadena se encuentra en otra #}
{% if password in login %}
    ERROR: la contraseña no puede ser una parte del login
{% endif %}
```

Combinando el operador `in` con el test `not` y la estructura `if` se pueden evaluar con facilidad expresiones complejas:

```
{% if usuario.nivel in ['superior', 'avanzado', 'experto'] %}
    {# ... #}
{% endif %}

{% if numero not in 0..20 %}
    {# ... #}
{% endif %}
```

Operadores matemáticos

Operador	Explicación
`+`	Suma dos números o el valor de dos variables: `{{ 3 + 2 }}` = 5
	Ejemplo de uso: `Total: {{ precio + impuestos }}`
`-`	Resta dos números o el valor de dos variables: `{{ 1 - 5 }}` = -4

Operador	Explicación
	Ejemplo de uso: `Total: {{ precio - descuento }}`
`*`	Multiplica dos números o el valor de dos variables: `{{ 2 * 7 }}` = `14`
	Ejemplo de uso: `Subtotal: {{ precioUnitario * unidades }}`
`/`	Divide dos números o el valor de dos variables: `{{ 12 / 5 }}` = `2.4`
	Ejemplo de uso: `Precio medio: {{ total / numeroArticulos }}`
`//`	División entera de dos números o del valor de dos variables: `{{ 12 // 5 }}` = `2`
	Ejemplo de uso: `Jugadas restantes: {{ dinero // precioPorApuesta }}`
`%`	Módulo o resto de la división entera de dos números o del valor de dos variables: `{{ 7 % 2 }}` = `1`
`**`	Eleva el primer número a la potencia del segundo número: `{{ 5 ** 2 }}` = `25`, `{{ 7 ** 3 }}` = `343`

Por último, Twig también soporta el operador ternario de PHP `?`, lo que permite crear plantillas todavía más concisas:

```
{{ usuario.registrado ? 'Ver perfil' : 'Regístrate' }}

{{ producto.stock < 10
    ? 'Sólo quedan ' ~ producto.stock ~ ' unidades'
    : producto.stock ~ ' unidades'
}}
```

Además, a diferencia de PHP, el operador ternario permite omitir el segundo valor de la condición y añadir solamente el valor que se utiliza cuando la condición se cumple. Imagina que quieres añadir la clase CSS `seleccionado` solo cuando se cumpla una determinada condición:

```
{# operador ternario tradicional #}
<li class="{{ condicion ? 'seleccionado' : '' }}"> ... </li>

{# operador ternario simplificado #}
<li class="{{ condicion ? 'seleccionado' }}"> ... </li>
```

A.3.7 Otras funciones y etiquetas

`date()`, además del filtro date, Twig incluye una función llamada `date()`. Esta función convierte el argumento que se le pasa en una fecha válida, ideal para hacer comparaciones:

```
{% if date(usuario.fechaNacimiento) < date('now - 18 years') %}
    {# El usuario es mayor de edad #}
{% endif %}
```

La función `date()` admite como argumento cualquier elemento que entienda la función `date()` de PHP, desde objetos de tipo `DateTime` hasta cadenas de texto describiendo un momento de tiempo.

Opcionalmente puedes pasarle un segundo parámetro indicando en una cadena de texto la zona horaria de la fecha.

`random()`, selecciona aleatoriamente un número, una letra o un elemento de una colección. Esta función de Twig unifica varias funciones de PHP de manera consistente y predecible:

```
{# muestra un número aleatorio entre 0 y 2.147.483.647 #}
{{ random() }}

{# muestra un número aleatorio entre 0 y 10 #}
{{ random(10) }}

{# selecciona una letra aleatoriamente #}
{{ random('Frase de prueba') }}

{# selecciona un elemento de la colección aleatoriamente #}
{% set usuarios = ['...', '...', '...', '...'] %}
{{ random(usuarios) }}
{{ random('a'..'z') }}
```

`flush`, hace que se envíen los contenidos de la plantilla al navegador del usuario, aunque todavía no se haya terminado de crear la plantilla completa. Su uso puede ser necesario para aplicaciones con plantillas extremadamente complejas y largas que quieren mejorar el rendimiento aparente de la aplicación:

```
{# ... código Twig ... #}

{# se envía al usuario todos los contenidos anteriores #}
{% flush %}

{# ... más código Twig ... #}

{# se envía al usuario el resto de contenidos #}
{% flush %}
```

A.3.8 Herencia de plantillas

Según la documentación oficial de Twig, la herencia de plantillas es su característica más destacada. Como sucede con la herencia de clases, el objetivo es la reutilización de código.

Imagina que una aplicación web dispone de dos páginas llamadas `portada.html.twig` y `contacto.html.twig` con el siguiente contenido:

```
{# portada.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, las mejores ofertas y los mejores precios</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
```

```
</head>
<body>
    <h1>La oferta del día</h1>
    {# ... #}
</body>
</html>

{# contacto.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Contacto</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <h1>Contacto</h1>
    {# ... #}
</body>
</html>
```

Las dos páginas comparten la misma estructura y muchas etiquetas HTML. La herencia de plantillas de Twig propone crear una nueva plantilla base que incluya todos los elementos comunes y después hacer que cada plantilla individual herede de la nueva plantilla base.

Para ello, crea en primer lugar una plantilla llamada `base.html.twig`:

```
{# base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>## EL TÍTULO ##</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
    ## EL CONTENIDO ##
</body>
</html>
```

En el código anterior, se han marcado como `## EL TÍTULO ##` y `## EL CONTENIDO##` las partes que cambian en cada plantilla. En Twig estas partes que tienen que rellenar cada plantilla se denominan bloques y se definen con la etiqueta `block`:

```
{# base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

```
    <title>{% block titulo %}{% endblock %}</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
    {% block contenido %}{% endblock %}
</body>
</html>
```

Utilizando esta plantilla base, puedes rehacer la `portada.html.twig` de la siguiente manera:

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

{% block titulo %}Cupon,las mejores ofertas y los mejores precios{% endblock %}

{% block contenido %}
    <h1>La oferta del día</h1>
    {# ... #}
{% endblock %}
```

Cuando una plantilla hereda de otra, su primera etiqueta debe ser `{% extends %}`, que indica la ruta de la plantilla de la que hereda. Una vez añadida la etiqueta `{% extends %}`, esta plantilla ya sólo puede rellenar los bloques de contenido definidos en la plantilla base. Si tratas de añadir nuevos bloques o contenidos HTML, Twig muestra un mensaje de error.

Siguiendo el mismo ejemplo, como `portada.html.twig` hereda de la plantilla `base.html.twig`, sólo puede crear contenidos dentro de dos bloques llamados `titulo` y `contenido`.

El mecanismo de herencia en Twig es bastante flexible, ya que por ejemplo las plantillas que heredan no tienen la obligación de rellenar con contenidos todos los bloques de la plantilla base, sólo aquellos que necesiten. Además, pueden crear nuevos bloques, siempre que estos se definan dentro de algún bloque de la plantilla base.

Aplicando la herencia sobre la plantilla `contacto.html.twig` el resultado es:

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

{% block titulo %}Contacto{% endblock %}

{% block contenido %}
    <h1>Contacto</h1>
    {# ... #}
{% endblock %}
```

Cuando un bloque tiene muy pocos contenidos, como por ejemplo el bloque `titulo` de la plantilla anterior, puedes utilizar una notación más concisa:

```
{# La dos instrucciones siguientes son equivalentes #}
{% block titulo %}Contacto{% endblock %}

{% block titulo 'Contacto' %}

{# También se pueden utilizar variables #}
{% block titulo %}Oferta del día: {{ oferta.titulo }}{% endblock %}

{% block titulo 'Oferta del día: ' ~ oferta.titulo %}
```

A.3.8.1 Definiendo el contenido inicial de los bloques

Los bloques de la plantilla base también pueden incluir contenidos, que se mostrarán siempre que la plantilla hija no defina el bloque. Imagina que la plantilla base define un título por defecto para todas las páginas:

```
{# base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block titulo %}Cupon, las mejores ofertas y los mejores precios{%
endblock %}</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
    {% block contenido %}{% endblock %}
</body>
</html>
```

Si ahora una plantilla hereda de `base.html.twig` y no define el contenido del bloque `titulo`, se utilizará el contenido definido en la plantilla `base.html.twig`. Así que puedes simplificar la plantilla `portada.html.twig`, porque el título que define es el mismo que el de la plantilla base:

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

{% block contenido %}
    <h1>La oferta del día</h1>
    {# ... #}
{% endblock %}
```

Como la plantilla `contacto.html.twig` si que incluye un bloque llamado `titulo` con su propio contenido, se sigue utilizando este título en vez del que define la plantilla base.

Si quieres utilizar el contenido del bloque definido en la plantilla base pero también añadir más contenidos, puedes hacer uso de la función `parent()`. Dentro de un bloque, esta función obtiene el contenido de ese mismo bloque en la plantilla base:

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

{% block titulo %}
    Contacto - {{ parent() }}
{% endblock %}

{% block contenido %}
    <h1>Contacto</h1>
    {# ... #}
{% endblock %}
```

Ahora el título que muestra la página creada con la plantilla `contacto.html.twig` es `Contacto -` seguido del contenido del bloque `titulo` en la plantilla base. Por tanto, el título será *"Contacto - Cupon, las mejores ofertas y los mejores precios"*.

La función `parent()` es ideal por ejemplo para las zonas laterales de las páginas, ya que la plantilla base puede definir los contenidos comunes de esa zona y el resto de plantillas reemplazarlos o ampliarlos con nuevos contenidos.

A.3.8.2 Reutilizando el contenido de los bloques

Una misma plantilla no puede contener dos bloques con el mismo nombre. Si quieres mostrar el contenido de un bloque varias veces, utiliza la función `block()` pasando como parámetro el nombre del bloque.

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

{% block titulo %}Contacto{% endblock %}

{% block contenido %}
    <h1>{{ block('titulo') }}</h1>
    {# ... #}
{% endblock %}
```

En la plantilla anterior, el título que se muestra en la ventana del navegador (etiqueta `<title>`) coincide con el título que se muestra como parte de los contenidos (etiqueta `<h1>`). Como los dos contenidos son iguales, define el valor del bloque `titulo` y utilízalo después dentro de los contenidos gracias a la función `block()`.

A.3.8.3 Anidando bloques

Twig permite anidar bloques dentro de otros bloques, sin importar su número ni el nivel de profundidad del anidamiento. La plantilla base anterior define un solo bloque `contenido` para todos los contenidos de las páginas. Sin embargo, en una aplicación web real puede ser más interesante añadir más bloques dentro del bloque principal de contenidos:

```
{# base.html.twig #}
<!DOCTYPE html>
```

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block titulo %}Cupon, las mejores ofertas y los mejores precios{%
endblock %}</title>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
    {% block contenido %}
        <article>
            {% block principal %}
                <section></section>
            {% endblock %}

            {% block secundario %}
                <aside></aside>
            {% endblock %}
        </article>
    {% endblock %}
</body>
</html>
```

Los bloques anidados se definen de la misma forma que los bloques normales. Si la plantilla tiene mucho código, puedes añadir el nombre del bloque junto a la etiqueta `{% endblock %}`, para localizar más fácilmente el final de cada bloque:

```
{# base.html.twig #}

{# ... #}

<body>
    {% block contenido %}
        <article>
            {% block principal %}
                <section></section>
            {% endblock principal %}

            {% block secundario %}
                <aside></aside>
            {% endblock secundario %}
        </article>
    {% endblock contenido %}
</body>
</html>
```

Esta nueva plantilla define una estructura HTML básica en la que las páginas contienen una zona principal de contenidos y otra zona secundaria. De esta forma, si la plantilla `contacto.html.twig` hereda de la nueva plantilla base, puede utilizar los nuevos bloques en vez del bloque `contenido`:

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

{% block titulo %}Contacto{% endblock %}

{% block principal %}
    <h1>{{ block('titulo') }}</h1>
    {# ... #}
{% endblock %}

{% block secundario %}
    <h3>Quiénes somos</h3>
    {# ... #}
{% endblock %}
```

Sin embargo, la aplicación web puede tener otras páginas especiales que no utilizan la misma estructura a dos columnas propuesta por la plantilla base. Este puede ser el caso por ejemplo de la portada, normalmente la página más especial del sitio web.

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

{% block contenido %}
    <h1>La oferta del día</h1>
    <div>
        {# ... #}
    </div>
{% endblock %}
```

Para definir su propia estructura de contenidos, la plantilla `portada.html.twig` opta por establecer directamente el valor del bloque `contenido`, no haciendo uso de los bloques interiores `principal` y `secundario` de la plantilla base.

A.3.8.4 Herencia dinámica

La etiqueta `{% extends %}` admite cualquier expresión válida de Twig como nombre de la plantilla base. Esto permite por ejemplo utilizar el operador ternario para elegir la plantilla con la que mostrar un listado de elementos:

```
{% extends opciones.compacta ? 'listado.html.twig' : 'tabla.html.twig' %}
```

Si el valor de la opción `compacta` es `true`, los elementos se muestran con una plantilla que hereda de la plantilla `listado.html.twig`. Si la opción vale `false` se utiliza la plantilla `tabla.html.twig`.

También se puede utilizar directamente el valor de una variable para indicar el nombre de la plantilla base:

```
{# se hereda de la plantilla administrador.html.twig #}
{% set usuario = { tipo: 'administrador' } %}
{% extends usuario.tipo ~ '.html.twig' %}
```

```
{# se hereda de la plantilla tienda.html.twig #}
{% set usuario = { tipo: 'tienda' } %}
{% extends usuario.tipo ~ '.html.twig' %}
```

Además de indicar el nombre de una plantilla, también puedes indicar una colección de plantillas. Twig utiliza la primera plantilla que exista empezando por la primera:

```
{% extends ['primera.html.twig', 'segunda.html.twig', 'tercera.html.twig'] %}
```

Gracias a esta herencia dinámica selectiva, el sitio web de un periódico podría por ejemplo utilizar la siguiente estrategia para la plantilla que muestra las noticias:

```
{% extends [
    'categoria_' ~ noticia.categoria ~ '.html.twig',
    'seccion_' ~ noticia.seccion ~ '.html.twig',
    'noticia.html.twig'
] %}
```

Imagina que la noticia que se muestra pertenece a la sección `internacional` y a la categoría `america`. Si existe una plantilla llamada `categoria_america.html.twig`, Twig la utiliza como base de la plantilla que muestra la noticia. Si no existe, Twig busca y tratará de utilizar la plantilla `seccion_internacional.html.twig`. Si tampoco existe esa plantilla, se utiliza la plantilla genérica `noticia.html.twig`.

A.3.9 Reutilización horizontal

La herencia de plantillas permite reutilizar grandes cantidades de código entre plantillas similares. La reutilización horizontal permite extraer aquellas partes de código que se repiten en varias plantillas, sin obligar a que unas hereden de otras.

Imagina que la portada y varias páginas interiores de un sitio web muestran un listado de elementos:

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

{% block contenido %}
    <h1>Ofertas destacadas</h1>
    {% for oferta in destacadas %}
        <h2>{{ oferta.titulo }}</h2>
        <p>{{ oferta.descripcion }}</p>
        {# ... #}
    {% endfor %}
{% endblock %}

{# recientes.html.twig #}
{% extends 'base.html.twig' %}

{% block contenido %}
```

```
    <h1>Ofertas recientes</h1>
    {% for oferta in recientes %}
        <h2>{{ oferta.titulo }}</h2>
        <p>{{ oferta.descripcion }}</p>
        {# ... #}
    {% endfor %}
{% endblock %}

{# ciudad.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Las ofertas de la ciudad Lorem Ipsum</title>
    <link href="estilos_avanzados.css" rel="stylesheet" type="text/css" />
</head>
<body>
    {# ... #}

    <aside>
        <h3>Ofertas cercanas</h3>
        {% for oferta in cercanas %}
            <h2>{{ oferta.titulo }}</h2>
            <p>{{ oferta.descripcion }}</p>
            {# ... #}
        {% endfor %}
    </aside>
</body>
</html>
```

Aparentemente, las tres plantillas anteriores son muy diferentes: `portada` y `recientes` heredan de la plantilla base, pero `ciudad` no utiliza la herencia de plantillas. Además, la plantilla `portada` muestra las ofertas destacadas, la plantilla `recientes` muestra las ofertas recientes y la plantilla `ciudad` muestra las ofertas cercanas.

A pesar de sus diferencias, el código HTML + Twig del listado de ofertas es idéntico en las tres plantillas. Esta es la clave de la reutilización horizontal: localizar *trozos* de código muy similares en diferentes plantillas.

Una vez localizado, extrae el código común y crea una nueva plantilla sólo con ese código (en este ejemplo, la nueva plantilla se llama `listado.html.twig`):

```
{# listado.html.twig #}
{% for oferta in ofertas %}
    <h2>{{ oferta.titulo }}</h2>
    <p>{{ oferta.descripcion }}</p>
    {# ... #}
{% endfor %}
```

Observa que en la plantilla anterior se utiliza la variable `ofertas` como nombre de la colección de ofertas que recorre el bucle. El resto de código es idéntico al de las plantillas anteriores.

A continuación, refactoriza las plantillas originales añadiendo la función `{{ include() }}`:

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

{% block contenido %}
    <h1>Ofertas destacadas</h1>
    {{ include('listado.html.twig') }}
{% endblock %}

{# recientes.html.twig #}
{% extends 'base.html.twig' %}

{% block contenido %}
    <h1>Ofertas recientes</h1>
    {{ include('listado.html.twig') }}
{% endblock %}

{# ciudad.html.twig #}
<!DOCTYPE html>
<html>
    {# ... #}

    <aside>
        <h3>Ofertas cercanas</h3>
        {{ include('listado.html.twig') }}
    </aside>
</body>
</html>
```

La función `{{ include() }}` incluye dentro de una plantilla el código de cualquier otra plantilla indicada como parámetro. Así, en el mismo punto en el que escribas `{{ include('listado.html.twig') }}` se incluirá todo el código de la plantilla `listado`.

La plantilla incluida tiene acceso a todas las variables de la plantilla en la que se incluye. Así por ejemplo, cuando la plantilla `listado` se incluye dentro de `portada`, tiene acceso a cualquier variable de la plantilla `portada`.

Twig permite controlar mediante las opción `with_context` a qué variables pueden acceder las plantillas incluidas. Si no quieres que accedan a ninguna variable:

```
{# ... #}
<h1>Ofertas recientes</h1>
{{ include('listado.html.twig', with_context = false) }}
```

Como ahora la plantilla `listado` no tiene acceso a ninguna variable de la plantilla principal, su código no funciona porque no existe una colección llamada `ofertas` sobre la que pueda iterar.

Si en vez de restringir el acceso a todas las variables prefieres seleccionar a qué variables se puede acceder, pasa las variables permitidas en un array asociativo:

```
{# ... #}
<h1>Ofertas recientes</h1>
{{ include('listado.html.twig', { ofertas: ofertas }, with_context = false) }}
```

Si quieres pasar muchas variables, puede resultar interesante crear el array asociativo primero e indicar después simplemente su nombre:

```
{# ... #}
{% set datos = { ofertas: ofertas, titulo: '...' } %}
<h1>Ofertas recientes</h1>
{{ include('listado.html.twig', datos, with_context = false) }}
```

Pasar variables explícitamente en la función `{{ include() }}` es muy habitual en las aplicaciones web reales. El principal motivo es que permite renombrar variables. Si observas las plantillas originales, verás que cada una llama de forma diferente a su colección de ofertas: en `portada` se llama `destacadas`, en `recientes` se llama `recientes` y en `ciudad` se llama `cercanas`. Como la plantilla `listado.html.twig` espera que la colección se llame `ofertas`, no va a funcionar bien dentro de ninguna plantilla.

La solución consiste en pasar cada plantilla la variable de las ofertas y renombrarla a `ofertas`, que es lo que espera la plantilla `listado.html.twig`:

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

{% block contenido %}
    <h1>Ofertas destacadas</h1>
    {{ include('listado.html.twig', { ofertas: destacadas }) }}
{% endblock %}

{# recientes.html.twig #}
{% extends 'base.html.twig' %}

{% block contenido %}
    <h1>Ofertas recientes</h1>
    {{ include('listado.html.twig', { ofertas: recientes }) }}
{% endblock %}

{# ciudad.html.twig #}
<!DOCTYPE html>
<html>
    {# ... #}
```

```
    <aside>
        <h3>Ofertas cercanas</h3>
        {{ include('listado.html.twig', { ofertas: cercanas }) }}
    </aside>
</body>
</html>
```

A.3.9.1 Reutilización dinámica

Como sucede con la herencia de plantillas, la etiqueta `{{ include() }}` también permite el uso de cualquier expresión válida de Twig como nombre de la plantilla incluida.

```
{% for oferta in ofertas %}
    {{ include(oferta.tipo == 'destacada' ?
      'destacada.html.twig' : 'oferta.html.twig'
    ) }}
{% endfor %}
```

El código anterior incluye la plantilla `destacada.html.twig` para las ofertas destacadas y la plantilla `oferta.html.twig` para cualquier otro tipo de oferta. Las dos plantillas podrán acceder a los datos de la oferta mediante la variable `oferta`.

La función `{{ include() }}` también permite controlar su comportamiento cuando la plantilla incluida no existe. En primer lugar, la opción `ignore_missing` indica que si la plantilla incluida no existe, se ignore completamente la función `{{ include() }}`:

```
{% set seccion = ... %}
{{ include('lateral_' ~ seccion ~ '.html.twig', ignore_missing = true ) }}
```

Si la variable `seccion` fuese `economia`, Twig busca la plantilla llamada `lateral_economia.html.twig`. Si la encuentra, la incluye; si no la encuentra, se ignora esta función `{{ include() }}` y no se produce ningún error.

Igualmente, también puedes indicar varias plantillas para que Twig las vaya probando secuencialmente. La primera plantilla que exista se incluye y el resto se ignoran:

```
{% set seccion   = ... %}
{% set categoria = ... %}

{{ include([
    'lateral_' ~ categoria ~ '.html.twig',
    'lateral_' ~ seccion ~ '.html.twig',
    'lateral.html.twig'
]) }}
```

Si en el código anterior la sección es `economia` y la categoría es `bolsa`, Twig trata de encontrar la plantilla `lateral_bolsa.html.twig`. Si existe esa plantilla, se incluye y se ignora el resto. Si no existe, se repite el proceso para la plantilla `lateral_economia.html.twig`. Si tampoco existe, se incluye la plantilla `lateral.html.twig`. Si tampoco existiera esta plantilla, Twig mostraría un error.

Para evitar este último error, combina el `include` múltiple con la opción `ignore_missing`:

```
{% set seccion   = ... %}
{% set categoria = ... %}

{{ include([
    'lateral_' ~ categoria ~ '.html.twig',
    'lateral_' ~ seccion ~ '.html.twig',
    'lateral.html.twig'
], ignore_missing = true) }}
```

A.3.9.2 Herencia adaptable

Twig incluye una etiqueta llamada `{% embed %}` que combina lo mejor de la herencia (`{% extends %}`) con lo mejor de la reutilización horizontal (`{{ include() }}`). Imagina que en tu aplicación utilizas un sistema de plantillas similar al de Twitter Bootstrap:

```
<!-- ... -->

<!-- grid/rejilla a 2 columnas -->
<div class="row">
    <div class="span9"> Contenido principal </div>

    <div class="span3"> Zona lateral </div>
</div>

<!-- ... -->
```

Cuando utilizas un sistema de plantillas como el anterior, es muy común repetir una y otra vez el código que define los *grids* o rejillas. ¿Cómo se puede reutilizar el código en Twig para escribir cada grid o rejilla una sola vez?

La función `{{ include() }}` no se puede utilizar en este caso, ya que sólo incluye los contenidos que le indicas y no puedes modificarlos (no podrías rellenar el grid/rejilla con contenidos). Utilizar la etiqueta `{% extends %}` sería posible, pero tendrías que crear una plantilla base para cada posible rejilla que se de en tu aplicación.

Imagina esta página compleja que usa una rejilla a tres columnas seguida de otra rejilla a dos columnas iguales y termina con la misma rejilla a tres columnas inicial:

```
<!-- ... -->

<!-- grid/rejilla a 3 columnas -->
<div class="row">
    <div class="span6"> Contenido principal </div>
    <div class="span3"> Zona lateral #1 </div>
    <div class="span3"> Zona lateral #2 </div>
</div>

<!-- grid/rejilla a 2 columnas -->
```

```
<div class="row">
    <div class="span6"> Zona de contenidos #1 </div>
    <div class="span6"> Zona de contenidos #2 </div>
</div>

<!-- grid/rejilla a 3 columnas -->
<div class="row">
    <div class="span6"> Contenido principal </div>
    <div class="span3"> Zona lateral #1 </div>
    <div class="span3"> Zona lateral #2 </div>
</div>

<!-- ... -->
```

La única solución técnicamente viable para crear la estructura anterior consiste en utilizar la etiqueta `{% embed %}`, que se comporta como una función `{{ include() }}` en la que puedes modificar sus contenidos antes de incluirlos.

En primer lugar, define dos plantillas Twig nuevas con el código de cada rejilla:

```
{# rejilla_3_columnas.twig #}
<div class="row">
    <div class="span6">
        {% block contenido %}{% endblock %}
    </div>
    <div class="span3">
        {% block lateral1 %}{% endblock %}
    </div>
    <div class="span3">
        {% block lateral2 %}{% endblock %}
    </div>
</div>

{# rejilla_2_columnas.twig #}
<div class="row">
    <div class="span6">
        {% block contenido1 %}{% endblock %}
    </div>
    <div class="span6">
        {% block contenido2 %}{% endblock %}
    </div>
</div>
```

Ahora ya puedes mostrar esas rejillas en cualquier parte de cualquier otra plantilla Twig:

```
{# página con una rejilla a dos columnas #}
{# ... #}
{% embed 'rejilla_3_columnas.twig' %}
    {% block contenido %} ... {% endblock %}
```

```
    {% block lateral1  %} ... {% endblock %}
    {% block lateral2  %} ... {% endblock %}
{% endembed %}

{# página con dos rejillas a 2 columnas #}
{# ... #}

{% embed 'rejilla_2_columnas.twig' %}
    {% block contenido1 %} ... {% endblock %}
    {% block contenido2 %} ... {% endblock %}
{% endembed %}

{# ... #}

{% embed 'rejilla_2_columnas.twig' %}
    {% block contenido1 %} ... {% endblock %}
    {% block contenido2 %} ... {% endblock %}
{% endembed %}
```

La etiqueta `{% embed %}` admite las mismas opciones que la función `{{ include() }}`, por lo que puedes pasarle variables, limitar el acceso a las variables de la plantilla principal e incluso no mostrar ningún error cuando no exista la plantilla que quieres embeber.

A.4 Extensiones

Twig incluye decenas de filtros, funciones, etiquetas y operadores. No obstante, si desarrollas una aplicación compleja, seguramente tendrás que crear tus propias extensiones. Se define como extensión cualquier elemento que amplíe las características o mejore el funcionamiento de Twig.

Las extensiones más comunes en las aplicaciones web son los macros, las variables globales, las funciones y los filtros.

A.4.1 Creando extensiones propias de Twig

Independientemente del tipo o cantidad de extensiones que definas, todas ellas se definen en clases que heredan de `Twig_Extension` y por convención su nombre acaba en `Extension` y se crean en el directorio `Twig/Extension/` del *bundle*.

Así que si quieres definir por ejemplo una extensión propia llamada `Utilidades` en el *bundle* `AppBundle`, debes crear la siguiente clase:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getName()
    {
        return 'utilidades';
```

```
    }
}
```

En el interior de la clase `UtilidadesExtension` se definen todos los filtros y funciones propios, como se explicará más adelante. Por el momento, el único método obligatorio es `getName()` que devuelve el nombre de la extensión (que debe ser único en la aplicación).

Por último, antes de poder utilizar esta extensión en tus plantillas Twig, es necesario activarla. Para ello, utiliza la siguiente configuración, explicada en la sección *Definiendo servicios especiales* (página 522) del apéndice B:

```
# app/config/config.yml
services:
    app.twig.utilidades:
        class: AppBundle\Twig\Extension\UtilidadesExtension
        tags:
            - { name: twig.extension }
```

A.4.2 Variables globales

Las variables globales son aquellas que están siempre disponibles en todas las plantillas de la aplicación. Aunque su uso resulta muy cómodo, si abusas de las variables globales puedes llegar a penalizar el rendimiento de la aplicación.

Las variables globales se definen bajo la clave `globals` del servicio `twig`:

```
# app/config/config.yml
twig:
    globals:
        impuestos: 18
        categoria_por_defecto: 'novedades'
```

Una vez definidas, ya puedes utilizar estas variables globales directamente en cualquier plantilla de la aplicación como si fuesen variables normales:

```
{% set oferta = ... %}
{# impuestos es una variable global #}
Impuestos: {{ oferta.precio * impuestos / 100 }}

{% for categoria in categorias %}
    {# categoria_por_defecto es una variable global #}
    {% if categoria == categoria_por_defecto %}
        {# ... #}
    {% else %}
        {# ... #}
    {% endif %}
{% endfor %}
```

Como las variables globales se tratan igual que el resto de variables, debes ser cuidadoso al elegir su nombre, para que no se produzcan colisiones con las variables de la plantilla. Una buena práctica recomendada consiste en definir todas las variables globales bajo un prefijo común:

```
# app/config/config.yml
twig:
    globals:
        global:
            impuestos: 18
            categoria_por_defecto: 'novedades'
```

Ahora las variables globales están disponibles en la plantilla a través del prefijo `global`:

```
{% set oferta = ... %}
Impuestos: {{ oferta.precio * global.impuestos / 100 }}

{% for categoria in categorias %}
    {% if categoria == global.categoria_por_defecto %}
        {# ... #}
    {% else %}
        {# ... #}
    {% endif %}
{% endfor %}
```

A.4.3 Macros

Según la documentación oficial de Twig, las macros se emplean para generar *trozos* de código HTML que se repiten una y otra vez en las plantillas. El ejemplo más común es el de los campos de un formulario:

```
<input type="..." name="..." id="..." value="..." required="..." />
```

Si tu plantilla contiene decenas de campos de formulario, define una macro que se encargue de generar su código HTML. Para definir una macro, utiliza la etiqueta `{% macro %}` dentro de la propia plantilla donde se van a utilizar. Cada macro debe tener un nombre único y, opcionalmente, una lista de argumentos:

```
{% macro campo(nombre, requerido, valor, tipo, id) %}
    {# ... #}
{% endmacro %}
```

El interior de la macro puede contener tanto código HTML y código de Twig como necesite. Normalmente su código es muy conciso, como demuestra el siguiente ejemplo de la macro que genera el código HTML de los campos de formulario:

```
{% macro campo(nombre, requerido, valor, tipo = 'text', id = 'nombre') %}
    <input type="{{ tipo }}" name="{{ nombre }}" id="{{ id }}"
           value="{{ valor }}" {{ requerido ? 'required' }} />
{% endmacro %}
```

Los argumentos de la macro siempre son opcionales, por lo que si no indicas su valor no se muestra ningún mensaje de error. Por defecto una macro no tiene acceso a las variables de la plantilla. Si las necesitas, pasa como argumento a la macro una variable especial llamada `_context` (con el guión bajo por delante).

Una vez creada, la macro se puede utilizar en la misma plantilla importándola de la siguiente manera (el nombre `utilidades` se puede elegir libremente):

```
{% import _self as utilidades %}
```

Ahora la plantilla ya puede hacer uso de la función `{{ utilidades.campo(...) }}`, tal y como muestra el siguiente ejemplo:

```
{% macro campo(nombre, requerido, valor, tipo = 'text', id = 'nombre') %}
    <input type="{{ tipo }}" name="{{ nombre }}" id="{{ id }}"
           value="{{ valor }}" {{ requerido ? 'required' }} />
{% endmacro %}

{% import _self as utilidades %}
Nombre:    {{ utilidades.campo('nombre', true, 'José') }}
Apellidos: {{ utilidades.campo('apellidos', true, 'García Pérez') }}
Teléfono:  {{ utilidades.campo('telefono') }}
```

A continuación se muestra el código HTML generado por esta plantilla Twig:

```
Nombre: <input type="text" name="nombre" id="nombre" value="José" required />
Apellidos: <input type="text" name="apellidos" id="apellidos" value="García Pére
z" required />
Teléfono: <input type="text" name="telefono" id="telefono" value=""  />
```

Si quieres reutilizar las mismas macros en diferentes plantillas, primero crea una plantilla dedicada exclusivamente a contener todas las macros. Imagina que esta nueva plantilla se llama `utilidades.html.twig`:

```
{# utilidades.html.twig #}
{% macro campo(nombre, requerido, valor, tipo = 'text', id = 'nombre') %}
    <input type="{{ tipo }}" name="{{ nombre }}" id="{{ id }}"
           value="{{ valor }}" {{ requerido ? 'required' }} />
{% endmacro %}
```

Para utilizar ahora la macro `campo()` dentro de una plantilla llamada `contacto.html.twig`, importa primero la plantilla `utilidades.html.twig` mediante la etiqueta `{% import %}`:

```
{# contacto.html.twig #}
{% import 'utilidades.html.twig' as utilidades %}

Nombre:    {{ utilidades.campo('nombre', true, 'José') }}
Apellidos: {{ utilidades.campo('apellidos', true, 'García Pérez') }}
Teléfono:  {{ utilidades.campo('telefono') }}
```

La palabra reservada `as` indica el nombre de la variable bajo la que se importan las macros. No es obligatorio que el nombre de esta variable coincida con el de la plantilla:

```
{# contacto.html.twig #}
{% import 'utilidades.html.twig' as formulario %}

Nombre:    {{ formulario.campo('nombre', true, 'José') }}
Apellidos: {{ formulario.campo('apellidos', true, 'García Pérez') }}
Teléfono:  {{ formulario.campo('telefono') }}
```

Si en la plantilla `utilidades.html.twig` incluyes muchas macros, no es necesario que las importes todas cuando sólo vas a necesitar unas pocas. Para importar macros individualmente, utiliza la etiqueta `{% from %}`:

```
{# contacto.html.twig #}
{% from 'utilidades.html.twig' import campo %}

Nombre:    {{ campo('nombre', true, 'José') }}
Apellidos: {{ campo('apellidos', true, 'García Pérez') }}
Teléfono:  {{ campo('telefono') }}
```

Observa cómo ahora la macro se importa directamente en la plantilla, por lo que puedes utilizar `campo()` en vez de `utilidades.campo()` o `formulario.campo()`. Si necesitas importar varias macros, indica todos sus nombres separándolos con comas:

```
{% from 'utilidades.html.twig' import campo, boton, texto %}
```

Cuando se importa una macro individual también se puede renombrar mediante la palabra reservada `as`:

```
{# contacto.html.twig #}
{% from 'utilidades.html.twig' import 'campo' as field %}

Nombre:    {{ field('nombre', true, 'José') }}
Apellidos: {{ field('apellidos', true, 'García Pérez') }}
Teléfono:  {{ field('telefono') }}
```

Utilizando la notación `_self.` las macros de una misma plantilla pueden llamarse entre sí. El siguiente ejemplo muestra cómo mejorar la macro `campo()` para poder crear formularios estructurados con tablas HTML, listas de elementos o etiquetas `<div>`:

```
{# utilidades.html.twig #}
{% macro campo(nombre, requerido, valor, tipo = 'text', id = 'nombre') %}
    <input type="{{ tipo }}" name="{{ nombre }}" id="{{ id }}"
           value="{{ valor }}" {{ requerido ? 'required' }} />
{% endmacro %}

{% macro fila(nombre, requerido, valor, tipo, id) %}
    <tr>
```

```
        <td>{{ nombre|capitalize }}</td>
        <td>{{ _self.campo(nombre, requerido, valor, tipo, id) }}</td>
    </tr>
{% endmacro %}

{% macro div(nombre, requerido, valor, tipo, id) %}
    <div>
        <strong>{{ nombre|capitalize }}</strong>
        {{ _self.campo(nombre, requerido, valor, tipo, id) }}
    </div>
{% endmacro %}

{% macro item(nombre, requerido, valor, tipo, id) %}
    <li>
        {{ _self.div(nombre, requerido, valor, tipo, id) }}
    </li>
{% endmacro %}
```

Ahora puedes crear fácilmente formularios con diferentes estructuras internas (tablas, listas):

```
{# contacto.html.twig #}
{% import 'utilidades.html.twig' as formulario %}

<table>
    {{ formulario.fila('nombre', true, 'José') }}
    {{ formulario.fila('apellidos', true, 'García Pérez') }}
    {{ formulario.fila('telefono') }}
</table>

<ul>
    {{ formulario.item('nombre', true, 'José') }}
    {{ formulario.item('apellidos', true, 'García Pérez') }}
    {{ formulario.item('telefono') }}
</ul>
```

De hecho, gracias a la palabra reservada `as`, puedes cambiar la estructura de los formularios sin modificar el código de la plantilla. El truco consiste en cambiar el nombre de la macro al importarla y elegir siempre el mismo nombre:

```
{# contacto.html.twig #}
{% from 'utilidades.html.twig' import fila as campo %}

<table>
    {{ formulario.campo('nombre', true, 'José') }}
    {{ formulario.campo('apellidos', true, 'García Pérez') }}
    {{ formulario.campo('telefono') }}
</table>

{# ... #}
```

```
{% from 'utilidades.html.twig' import item as campo %}
<ul>
    {{ formulario.campo('nombre', true, 'José') }}
    {{ formulario.campo('apellidos', true, 'García Pérez') }}
    {{ formulario.campo('telefono') }}
</ul>
```

A pesar de que son muy útiles, las macros no suelen utilizarse más que para generar *trozos* comunes de código HTML. Cuando la lógica aumenta, se utilizan funciones de Twig o *trozos* de plantilla incluidos con la función `{{ include() }}`.

A.4.4 Filtros

Los filtros son con mucha diferencia las extensiones más utilizadas en las plantillas Twig. Los filtros se pueden aplicar sobre cualquier expresión válida de Twig, normalmente variables. El nombre del filtro siempre se escribe detrás de la expresión, separándolo con una barra vertical `|` y también pueden incluir argumentos:

```
{# filtro sin argumentos #}
{{ variable|filtro }}

{# filtro con argumentos #}
{{ variable|filtro(argumento1, argumento2) }}
```

Técnicamente, un filtro de Twig no es más que una función de PHP a la que se pasa como primer argumento la expresión sobre la que se aplica el filtro:

```
// {{ variable|filtro }} es equivalente a:
echo filtro(variable);

// {{ variable|filtro(argumento1, argumento2) }} es equivalente a:
echo filtro(variable, argumento1, argumento2);
```

Los filtros en Symfony siempre se definen dentro de alguna extensión propia. Siguiendo con el mismo ejemplo de las secciones anteriores, imagina que dispones de la siguiente extensión vacía llamada `Utilidades`:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getName()
    {
        return 'utilidades';
    }
}
```

A continuación se muestra cómo definir un nuevo filtro llamado `longitud` que calcula la longitud de una cadena de texto. En primer lugar añade el método `getFilters()` en la clase de la extensión y declara el nuevo filtro:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter('longitud', array($this, 'longitud')),
        );
    }

    // ...
}
```

El método `getFilters()` devuelve un array con todos los filtros definidos por la extensión. Cada filtro se declara con la clase `Twig_SimpleFilter`. Su primer argumento es el nombre del filtro (lo que escribes en la plantilla para utilizarlo) y el segundo argumento es la función o método PHP que se ejecuta al utilizar el filtro en la plantilla. Normalmente este método se define en la propia clase de la extensión, así que es común usar `array($this, 'nombre_metodo')`:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter('longitud', array($this, 'longitud')),
        );
    }

    public function longitud($valor)
    {
        return strlen($valor);
    }

    // ...
}
```

El primer argumento del método del filtro siempre es el valor (expresión o variable) sobre la que se aplica el filtro en la plantilla. Si el filtro también utiliza parámetros, estos se pasan después del valor:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    // ...

    public function longitud($valor, $parametro1, $parametro2, ...)
    {
        return strlen($valor);
    }
}
```

Una vez implementada la lógica del filtro, y si la extensión `Utilidades` está activada en la aplicación, ya puedes utilizar el nuevo filtro en cualquier plantilla de la siguiente manera:

```
{{ variable|longitud }}
```

A.4.5 Generando código HTML

Twig aplica el mecanismo de escape no sólo a las variables, sino también al resultado de todos los filtros. Por tanto, si tus filtros generan como respuesta código HTML, tendrás que aplicar también el filtro `raw` para evitar problemas:

```
{{ variable|mi_filtro|raw }}
```

Añadir el filtro `raw` siempre que utilices tu filtro es algo tedioso. Por eso Twig permite indicar que la respuesta generada por un filtro es segura y por tanto, que debe mostrarse tal cual en la plantilla. Para ello, añade la opción `is_safe` al definir el filtro:

```
// En Symfony
public function getFilters()
{
    return array(
        new \Twig_SimpleFilter('longitud', array($this, 'longitud'), array(
            'is_safe' => array('html')
        )),
    );
}
```

Por otra parte, si quieres que Twig aplique el mecanismo de escape al valor que pasa al filtro, añade la opción `pre_escape`:

```
// En Symfony
public function getFilters()
{
    return array(
        new \Twig_SimpleFilter('longitud', array($this, 'longitud'), array(
            'pre_escape' => array('html')
        )),
```

```
        );
    }
```

A.4.6 Obteniendo información sobre el entorno de ejecución

El filtro `longitud` definido anteriormente es demasiado simple para utilizarlo en una aplicación web real. El motivo es que en vez de la función `strlen()`, debería hacer uso de la función `mb_strlen()`, que funciona bien con todos los idiomas.

Para un mejor funcionamiento, la función `mb_strlen()` espera como segundo argumento la codificación de caracteres utilizada en la cadena de texto que se le pasa. ¿Cómo se puede determinar la codificación de caracteres dentro de una plantilla de Twig? La respuesta es muy simple, ya que cuando se configura el entorno de ejecución de Twig, una de sus opciones es precisamente el *charset* o codificación de caracteres.

Así que para que los filtros puedan obtener esta información, sólo es necesario que accedan a la configuración del entorno de ejecución de Twig. Para ello, añade la opción `needs_environment` al definir el filtro:

```
// En Symfony
public function getFilters()
{
    return array(
        new \Twig_SimpleFilter('longitud', array($this, 'longitud'), array(
            'needs_environment' => true
        )),
    );
}
```

Después, modifica el código del filtro, ya que ahora Symfony le pasa el entorno de ejecución como primer parámetro:

```
function longitud(\Twig_Environment $entorno, $valor)
{
    $codificacion = $entorno->getCharset();
    return mb_strlen($valor, $codificacion);
}
```

A través de la variable `$entorno` puedes acceder a información como la versión de Twig (`$entorno::VERSION`), la codificación de caracteres utilizada (`$entorno->getCharset()`), o si Twig se está ejecutando en modo *debug* (`$entorno->isDebug()`).

A.4.7 Funciones

Las funciones de Twig son similares a los filtros, pero su finalidad es diferente. El objetivo de los filtros es manipular el contenido de las variables, mientras que las funciones se utilizan para generar contenidos. Su notación también es diferente, ya que las funciones nunca se aplican sobre variables ni expresiones y su nombre siempre va seguido de dos paréntesis:

```
{# función sin argumentos #}
{{ mi_funcion() }}

{# función con argumentos #}
{{ mi_funcion(argumento1, argumento2) }}
```

Definir una función de Twig en Symfony es muy similar a definir un filtro. La única diferencia es que ahora la función se define en el método `getFunctions()` en vez de `getFilters()` y que la función se declara con la clase `Twig_SimpleFunction` en vez de `Twig_SimpleFilter`. El siguiente código muestra cómo definir una función llamada `mi_funcion()`:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getFunctions()
    {
        return array(
            new \Twig_SimpleFunction('mi_funcion', array($this, 'miFuncion')),
        );
    }

    public function miFuncion()
    {
        // ...

        return $respuesta;
    }

    // ...
}
```

Si la función admite parámetros, Symfony los pasa automáticamente al método de la función en el mismo orden en el que se escriben en la plantilla:

```
// src/AppBundle/Twig/Extension/UtilidadesExtension.php
namespace AppBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    // ...

    public function miFuncion($parametro1, $parametro2, ...)
    {
        // ...

        return $respuesta;
```

```
    }
}
```

Una vez implementada la lógica de la función, y si la extensión `Utilidades` está activada en la aplicación, ya puedes utilizarla en cualquier plantilla de la siguiente manera:

```
{{ mi_funcion() }}
```

A.5 Usando Twig en Symfony

Utilizar Twig dentro de una aplicación Symfony es todavía mejor y más fácil que usarlo de forma independiente: las plantillas se cargan automáticamente, la caché y otras opciones importantes ya están preconfiguradas y dispones de funciones y filtros exclusivos como `path()`, `url()` y `trans()`.

A.5.1 Configuración

El comportamiento de Twig se ajusta mediante las opciones de configuración del servicio `twig`. A continuación se muestran las opciones disponibles y sus valores por defecto:

```
# app/config/config.yml
twig:
    auto_reload:          ~
    autoescape:           ~
    base_template_class:  ~
    cache:                %kernel.cache_dir%/twig
    charset:              %kernel.charset%
    debug:                %kernel.debug%
    optimizations:        ~
    strict_variables:     ~
```

- `auto_reload`: indica si se deben volver a compilar las plantillas de Twig cada vez que cambia su código fuente. Establece el valor `true` cuando estés desarrollando la aplicación, para ver instantáneamente los cambios sin tener que vaciar la caché. Si no estableces ningún valor, se le asigna el valor de la opción `debug`.
- `autoescape`: indica si se aplica automáticamente el mecanismo de escape en todas las plantillas. Su valor por defecto es `true`.
- `base_template_class`: establece la clase de la que heredan las plantillas compiladas. Su valor por defecto es `Twig_Template`.
- `cache`: indica la ruta absoluta del directorio caché donde se guardan las plantillas compiladas.
- `charset`: la codificación de caracteres que utilizan las plantillas. Su valor por defecto es `utf-8`.
- `debug`: indica si las plantillas compiladas deben incluir información de depuración. Concretamente, si vale `true` se añade un método `__toString()` a las clases de las plantillas para mostrar los nodos generados.
- `optimizations`: indica qué optimizaciones se utilizan al compilar las plantillas. Su valor por defecto es `-1`, que activa todas las optimizaciones. También puedes utilizar los valores `2`

(optimiza los bucles `for` no creando la variable `loop` a menos que sea necesario) y `4` (optimiza el filtro `raw`). Para desactivar todas las optimizaciones, utiliza el valor `0`.

- `strict_variables`: establece el comportamiento de Twig cuando una variable no existe. Si vale `false`, Twig ignora estos errores y convierte la variable inexistente al valor `null`. Si vale `true`, Twig muestra una excepción cada vez que encuentra una variable que no existe. Su valor por defecto es `false`.

A.5.2 Depuración

Twig incluye la función `dump()` como equivalente de la función `var_dump()` de PHP. Así puedes ver el contenido de cualquier variable, lo que facilita mucho la depuración de las aplicaciones:

```
{{ dump(nombre-de-variable) }}
```

A.5.3 Variables globales

Symfony crea automáticamente varias variables globales para que todas las plantillas Twig tengan acceso directo a los objetos más importantes de la aplicación. A todas ellas se puede acceder mediante la variable especial `app`.

- `app.environment`, devuelve el nombre del entorno bajo el que se está ejecutando la plantilla (`dev`, `prod`, etc.)
- `app.debug`, devuelve el valor de la opción `debug` en la aplicación (por defecto vale `true` en el entorno `dev` y `false` en el entorno `prod`)
- `app.user`, devuelve el objeto que representa al usuario actualmente logueado en la aplicación o `null` si nadie está conectado. Si tus usuarios se crean con una entidad de Doctrine, puedes utilizar este objeto para acceder a cualquiera de sus propiedades: `{{ app.user.nombre }}`, `{{ app.user.email }}`, etc.
- `app.session`, devuelve un objeto representando a la sesión del usuario. Así puedes consultar la información de la sesión (`{{ app.session.get('nombre-opcion') }}`) y también manipularla (`{{ app.session.set('nombre-opcion', 'valor') }}`).
- `app.request`, devuelve el objeto que representa a la petición del usuario. Gracias a este objeto puedes acceder a cualquier información sobre la petición: `{{ app.request.server.get('HTTP_HOST') }}`, `{{ app.request.headers.get('user-agent') }}`, `{{ app.request.cookies.get('nombre-cookie') }}`, etc.

A.5.4 Servicios como variables globales

Además de valores simples, las variables globales de Twig también pueden guardar servicios de Symfony. Imagina que dispones de una clase llamada `Slugger` que se encuentra en el *bundle* `AppBundle` e incluye un método `getSlug()`:

```
namespace AppBundle\Util;

class Slugger
{
```

```
    static public function getSlug($cadena)
    {
        // ...
    }
}
```

La clave para disponer de la clase `Slugger` y sus métodos en cualquier plantilla de Twig es convertir la clase en un servicio. Para ello, abre el archivo de configuración `app/config/services.yml` y añade lo siguiente:

```
# app/config/services.yml
services:
    # ...
    app.slugger:
        class: AppBundle\Util\Slugger
```

A continuación, define una variable global de Twig utilizando este nuevo servicio:

```
# app/config/config.yml
twig:
    globals:
        slugger: '@app.slugger'
```

Para asignar un servicio a una variable global, simplemente indica el nombre del servicio con el prefijo `@`. Con esta configuración, cualquier plantilla Twig de la aplicación puede hacer uso de la variable `slugger` para acceder a los métodos de la clase `Slugger`:

```
{{ slugger.getSlug('Lorem ipsum dolor sit amet') }}
{# el resultado es: lorem-ipsum-dolor-sit-amet #}
```

A.5.5 Internacionalización

Cuando se utiliza dentro de Symfony, Twig incluye una etiqueta y un filtro llamados `trans` para traducir los contenidos de la plantilla. Lo más sencillo es el filtro, sobre todo si quieres traducir el contenido de variables:

```
{{ " Regístrate como usuario!"|trans }}
{{ categoria.nombre|trans }}
```

La etiqueta es más cómoda para traducir contenidos muy largos:

```
{% trans %} Regístrate como usuario!{% endtrans %}

{% trans %}Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eius
mod tempor incididunt ut [...] laboris nisi ut aliquip ex ea commodo consequa
t.{% endtrans %}
```

Utilizar la etiqueta o el filtro es simplemente una cuestión de conveniencia, ya que el funcionamiento interno de Twig para localizar la traducción es idéntico en ambos casos. La única diferencia es cómo se aplica el mecanismo de escape:

```
{# NO se aplica el mecanismo de escape #}
{% trans %}
   <strong>Regístrate</strong> como usuario!
{% endtrans %}

{# SI se aplica el mecanismo de escape #}
{% set mensaje = ' <strong>Regístrate</strong> como usuario!' %}
{{ mensaje|trans }}

{# Evitar que se aplique el mecanismo de escape #}
{{ mensaje|trans | raw }}

{# NO se aplica el mecanismo de escape #}
{{ ' <strong>Regístrate</strong> como usuario!'|trans }}
```

El capítulo 10 (página 265) dedicado a internacionalizar el sitio web explica en detalle todas las opciones del componente de traducción de Symfony y cómo utilizarlo en las plantillas Twig. Entre otros se explica cómo crear catálogos de traducción, los diferentes formatos disponibles, traducciones con variables, traducciones especiales para plurales, traducciones de páginas estáticas, etc.

A.5.6 Enrutamiento

La versión estándar de Twig no incluye ninguna etiqueta para generar enlaces, ya que el enrutamiento depende de la aplicación en la que se utilice Twig. Symfony si que define dos funciones para generar enlaces, llamadas `path()` y `url()`. Su única diferencia es que la primera genera URL relativas y la segunda URL absolutas.

El primer argumento de las dos funciones es obligatorio e indica el nombre de la ruta con la que se genera la URL:

```
<a href="{{ path('portada') }}">Volver a la portada</a>
```

Si la ruta requiere parámetros, se indican como segundo argumento de la función:

```
<a href="{{ path('detalle_producto', { id: 3 }) }}">
    Información sobre el producto
</a>

<a href="{{ path('portada', { seccion: 51, pagina: 3 }) }}">
    Volver al listado de productos
</a>
```

Los ejemplos anteriores emplean la función `path()` para generar los enlaces, por lo que sus URL son relativas. Si necesitas generar URL absolutas, por ejemplo para incluir los enlaces en un archivo RSS o en un email, simplemente cambia `path()` por `url()`.

A.5.7 Personalizando los formularios

El código HTML de los formularios creados por Symfony se estructura con elementos `<div>`. Concretamente, la plantilla utilizada para crear los formularios con Twig se denomina

`form_div_layout.html.twig` y se encuentra en el directorio `src/Symfony/Bridge/Twig/Resources/views/Form/`. En cualquier caso, para modificar el aspecto de los formularios en tus plantillas no es necesario que modifiques las plantillas internas de Symfony.

Imagina que en un formulario quieres mostrar un campo de tipo URL. Lo más sencillo sería mostrarlo con la función `form_row()` de Twig:

```
{{ form_row(url) }}
```

El código HTML generado por la función anterior es:

```
<div><label for="form_url" class=" required">Url</label><input type="url" id="fo
rm_url" name="form[url]" required value=""></div>
```

Si abres la plantilla `form_div_layout.html.twig` comentada anteriormente, verás que los campos de tipo URL se generan de la siguiente manera:

```
{% block url_widget %}
{% spaceless %}
    {% set type = type|default('url') %}
    {{ block('field_widget') }}
{% endspaceless %}
{% endblock url_widget %}
```

Para modificar el aspecto de este tipo de campos en una única plantilla, añade la etiqueta `{% form_theme %}` tal como se muestra a continuación y después define el nuevo aspecto del campo añadiendo un bloque llamado `url_widget` (el mismo nombre que el de la plantilla interna de Twig):

```
{{ form_row(url) }}

{# ... #}

{% form_theme form _self %}

{% block url_widget %}
    {% set type = 'url' %}
    <em>http://</em> {{ block('field_widget') }}
{% endblock url_widget %}
```

La instrucción `{% form_theme form _self %}` indica que en la propia plantilla (`_self`) se define parte o todo el aspecto de los formularios. Twig buscará los bloques que necesita primero en la plantilla y después, si no los encuentra, en la plantilla original de Symfony.

De esta forma, el ejemplo anterior hace que todos los campos de tipo URL muestren el texto `<em>http://</em>` por delante del cuadro de texto donde se escribe la URL. El resto de campos de formulario mantienen su aspecto original.

El problema del código anterior es que no se puede reutilizar en los formularios del resto de plantillas de la aplicación. Para modificar los formularios de varias plantillas, crea primero una plantilla para definir el nuevo aspecto de los campos de formulario:

```
{# app/Resources/views/form/form.html.twig #}
{% block url_widget %}
    {% set type = 'url' %}
    <em>http://</em> {{ block('field_widget') }}
{% endblock url_widget %}
```

Después, en todas las plantillas en las que quieras modificar el aspecto de los formularios, debes añadir la misma etiqueta `{% form_theme %}` de antes, pero ahora no escribas `_self` sino la ruta de esta plantilla:

```
{% form_theme form 'form/form.html.twig' %}

{# ...#}

{{ form_row(url) }}
```

Por último, si quieres modificar el aspecto de todos los formularios de la aplicación, no es necesario que añadas la etiqueta `{% form_theme %}` en todas y cada una de las plantillas.

Symfony define una opción de Twig llamada `form_themes` que permite indicar la ruta de la plantilla que se utilizará para crear los formularios. Su valor por defecto es el siguiente, que hace que los formularios se creen con elementos `<div>`:

```
# app/config/config.yml
twig:
    # ...
    form_themes: ['form_div_layout.html.twig']
```

Además de `form_div_layout.html.twig` Symfony incluye otra plantilla llamada `form_table_layout.html.twig` que genera los formularios con tablas en vez de con `<div>`:

```
# app/config/config.yml
twig:
    # ...
    form_themes: ['form_table_layout.html.twig']
```

Crear una plantilla completa para definir el aspecto de los formularios es muy sencillo. Observa el código de la plantilla `form_table_layout.html.twig` que se encuentra en el directorio `src/Symfony/Bridge/Twig/Resources/views/Form/`. Gracias a la instrucción `use` se importa la plantilla `form_div_layout.html.twig` y después se redefinen solamente aquellos bloques cuyo aspecto se quiere modificar:

```
{# src/Symfony/Bridge/Twig/Resources/views/Form/form_table_layout.html.twig #}
{% use "form_div_layout.html.twig" %}
```

```
{% block field_row %}
    {# ... #}
{% endblock %}

{% block form_errors %}
    {# ... #}
{% endblock %}

{% block hidden_row %}
    {# ... #}
{% endblock %}

{% block form_widget %}
    {# ... #}
{% endblock %}
```

Si en vez de `form_table_layout.html.twig` quieres utilizar tu propia plantilla, indica su ruta dentro de la opción `form_themes`:

```
# app/config/config.yml
twig:
    # ...
    form_themes: ['form/form.html.twig']
```

Esta página se ha dejado vacía a propósito

APÉNDICE B

Inyección de dependencias

La *inyección de dependencias* (del inglés, *dependency injection*) es la clave para entender el funcionamiento de Symfony. Junto al *contenedor de servicios* (del inglés, *service container*) es lo que hace que Symfony sea tan flexible.

Aunque la terminología utilizada en ocasiones puede parecer confusa, entender la utilidad de la inyección de dependencias es muy sencillo. De hecho, el concepto es tan absurdamente simple, que es posible que ya hayas utilizado en tu código técnicas similares sin saberlo.

Si quieres convertirte en un programador experto de Symfony, resulta imprescindible dominar todos los conceptos explicados en este apéndice. Así te costará mucho menos desarrollar tus aplicaciones y su código será mucho mejor.

B.1 Entendiendo la inyección de dependencias

Antes de mostrar cómo utiliza Symfony la inyección de dependencias, esta sección introduce el propio concepto de inyección de dependencias con un ejemplo sencillo. Si estás seguro de que entiendes bien los conceptos de inyección de dependencias y contenedor de servicios, puedes saltarte esta sección.

Imagina que quieres que tu aplicación pueda generar mensajes de *debug* o depuración con información sobre las operaciones que está ejecutando. Lo más sencillo sería crear un objeto de tipo *logger* que permita añadir en un archivo de log los diferentes tipos de mensajes de depuración:

```
$logger = new Logger();

$logger->info('Conectando con la base de datos...');

// ...

$logger->error('No se ha podido conectar con la base de datos');
```

Para que no se pierda la información generada por la aplicación, lo lógico es guardar estos mensajes en un archivo. Para modularizar mejor la aplicación, se crea primero una clase genérica llamada `Archivo` con métodos que permitan manipular los contenidos de un archivo:

```
class Archivo
{
    private $archivo;
```

```
    function __construct()
    {
        $this->archivo = fopen(__DIR__.'/debug.log', 'a');
    }

    function escribir($contenido)
    {
        fwrite($this->archivo, $contenido)
    }

    // ...
}
```

Después, define otra clase llamada `Logger` que utilice la clase `Archivo` anterior para guardar los mensajes en el archivo de log:

```
class Logger
{
    protected $archivo;

    function __construct()
    {
        $this->archivo = new Archivo();
    }

    function error($mensaje)
    {
        $this->archivo->escribir('* [ERROR] '.$mensaje);
    }

    function info($mensaje)
    {
        $this->archivo->escribir('info: '.$mensaje);
    }

    // ...
}
```

Las dos clases `Archivo` y `Logger` son suficientes para que el siguiente código funcione correctamente y los mensajes se guarden en un archivo de log:

```
$logger = new Logger();

$logger->info('Conectando con la base de datos...');

// ...

$logger->error('No se ha podido conectar con la base de datos');
```

El código desarrollado hasta el momento funciona bien y es suficiente para solucionar la funcionalidad solicitada. Sin embargo, si lo utilizas en una aplicación web real, pronto te darás cuenta de una carencia importante. ¿Cómo se modifica el archivo en el que se guardan los mensajes? Por ejemplo para que cada componente utilice su propio archivo de log o para que cada entorno de ejecución (desarrollo o producción) guarden sus mensajes en archivos diferentes.

Una posible solución sería pasar la ruta del archivo en el propio constructor de la clase `Logger`, que a su vez lo pasaría al constructor de la clase `Archivo`:

```
$logger = new Logger(__DIR__.'/debug.log');

// ...

class Logger
{
    protected $archivo;

    function __construct($ruta)
    {
        $this->archivo = new Archivo($ruta);
    }

    // ...
}
```

El problema de esta solución es que tienes que indicar la ruta del archivo de log siempre que quieras escribir un mensaje. Así que en una aplicación web real, acabarías escribiendo cientos de veces la instrucción `$logger = new Logger(__DIR__.'/debug.log');`. Además, si la ruta del archivo cambia alguna vez, tendrías que cambiarla cientos de veces en toda la aplicación.

Suponiendo que lo anterior no sea un problema, imagina el siguiente escenario: desarrollas tu aplicación pensando que los mensajes de log se guardan en un archivo y tu cliente hace un cambio de última hora exigiendo que los mensajes se guarden en una base de datos en vez de en un archivo.

En ocasiones no es un cliente el que cambia la funcionalidad interna de algún componente, sino que lo hacen tus propios tests. Al crear test unitarios en una aplicación es muy común simplificar su código utilizando *mocks* u *objetos falsos* que son una versión simplificada del objeto original. ¿Cómo crear un *mock* del *logger* para que no utilice la clase `Archivo` sino que sólo simule su comportamiento?

La solución a estos problemas consiste en refactorizar la clase `Logger` para que no dependa de la clase `Archivo`. En su lugar, se utiliza una variable llamada `$manejador` que se pasa al constructor de la clase `Logger`:

```
class Logger
{
    protected $manejador;
```

```
    function __construct($manejador)
    {
        $this->manejador = $manejador;
    }

    function error($mensaje)
    {
        $this->manejador->escribir('* [ERROR] '.$mensaje);
    }

    function info($mensaje)
    {
        $this->manejador->escribir('info: '.$mensaje);
    }

    // ...
}
```

Después de este cambio, la clase `Logger` no sólo no depende de `Archivo`, sino que ya no es su responsabilidad ni crear ni configurar el objeto que guarda los mensajes. El `Logger` sólo sabe que alguien le pasará un objeto ya creado y que para guardar los mensajes podrá utilizar el método `escribir()` de ese objeto. Por tanto, ahora puedes guardar fácilmente los mensajes en un archivo, en una base de datos o en cualquier otro lugar:

```
// Guardar mensajes en un archivo
$archivo = new Archivo(__DIR__.'/debug.log');

$logger = new Logger($archivo);
$logger->info('...');

// Guardar mensajes en una base de datos
$db = new BaseDatos(array('dsn' => 'mysql:host=localhost;dbname=logs'));

$logger = new Logger($db);
$logger->info('...');

// Enviar los mensajes por email
$mailer = new Mailer(...);

$logger = new Logger($mailer);
$logger->info('...');
```

Esto es la inyección de dependencias! La inyección de dependencias consiste en crear y configurar primero el objeto `Archivo` y pasárselo después a la clase `Logger`. En otras palabras, la inyección de dependencias consiste en pasar (*inyectar*) a las clases todos los objetos que necesitan (*dependencias*) ya creados y configurados.

Aunque es un cambio aparentemente sutil, observa cómo ahora es muy sencillo cambiar los requerimientos de la aplicación (al `Logger` no le importa si le pasas `Archivo` o `BaseDatos`) y cambiar su comportamiento cuando sea necesario (en los tests se pasa `ArchivoMock` en vez de `Archivo`).

El código anterior inyecta las dependencias mediante el constructor (*"constructor injection"*), que es el caso más común. Pero también se pueden inyectar mediante los métodos *setter* de la clase (*"setter injection"*):

```
class Logger
{
    protected $manejador;

    function setManejador($manejador)
    {
        $this->manejador = $manejador;
    }

    // ...
}

// Guardar mensajes en un archivo
$archivo = new Archivo(__DIR__.'/debug.log');

$logger = new Logger();
$logger->setManejador($archivo);

$logger->info('...');
```

Y también existe la inyección de dependencias directamente a través de las propiedades de la clase (*"property injection"*):

```
class Logger
{
    public $manejador;

    // ...
}

// Guardar mensajes en un archivo
$archivo = new Archivo(__DIR__.'/debug.log');

$logger = new Logger();
$logger->manejador = $archivo;

$logger->info('...');
```

A pesar de que tiene muchas ventajas, el problema de la inyección de dependencias es que antes de utilizar una clase, debes acordarte de crear y configurar correctamente todas sus dependencias.

Afortunadamente esto también tiene solución y se llama **contenedor de inyección de dependencias**.

B.1.1 El contenedor de inyección de dependencias

Cuando el código de una aplicación es tan sencillo como el mostrado anteriormente, es fácil recordar las dependencias entre clases. Cuando quieras utilizar el *logger*, seguramente recordarás que antes debes crear una clase de tipo `Archivo`.

El problema es que en las aplicaciones web reales de Symfony existen docenas de estos objetos principales. Entre todos ellos suman cientos de dependencias cruzadas y cientos de opciones de configuración. Manejar todas estas dependencias a mano es simplemente imposible.

El **contenedor de inyección de dependencias** es un objeto que sabe cómo crear los objetos de tu aplicación. Para ello, conoce todas las relaciones entre tus clases y las configuración necesaria para instanciar correctamente cada clase.

Siguiendo con el ejemplo anterior, este podría ser un contenedor muy sencillo capaz de crear objetos de tipo `Logger`:

```
class Contenedor
{
    public function getManejador()
    {
        return new Archivo(__DIR__.'/debug.log');
    }

    public function getLogger()
    {
        $logger = new Logger($this->getManejador());
        return $logger;
    }
}
```

Haciendo uso del contenedor, puedes simplificar el código de tu aplicación a lo siguiente:

```
$contenedor = new Contenedor();
$logger = $contenedor->getLogger();

$logger->info('...');
```

Ahora tu aplicación simplemente pide un *logger* al contenedor de dependencias, sin preocuparse de cómo se crea o las clases que hay que instanciar y configurar previamente.

El problema del contenedor anterior es que de nuevo no se puede modificar con facilidad su comportamiento. Refactoriza su código para que mantenga la flexibilidad de la inyección de dependencias pero con la facilidad de uso que aporta un contenedor:

```
class Contenedor
{
```

```
    protected $opciones = array();

    public function __construct($opciones = array())
    {
        $this->opciones = $opciones;
    }

    public function getManejador()
    {
        switch($this->opciones['tipo']) {
            case 'db':
                return new BaseDatos(array('dsn' => $this->opciones['dsn']));

            case 'archivo':
            default:
                return new Archivo($this->opciones['ruta']);
        }
    }

    public function getLogger()
    {
        $logger = new Logger($this->getManejador());
        return $logger;
    }
}
```

Gracias a las opciones añadidas en el contenedor, ya puedes volver a modificar el comportamiento del *logger*. La gran ventaja es que tu código sólo debe pasar un array de opciones al contenedor:

```
// Guardar mensajes en un archivo
$contenedor = new Contenedor(array(
  'tipo' => 'archivo',
  'ruta' => __DIR__.'/debug.log',
));
$logger = $contenedor->getLogger();

// Guardar mensajes en una base de datos
$contenedor = new Contenedor(array(
  'tipo' => 'db',
  'dsn'  => 'mysql:host=localhost;dbname=logs',
));
$logger = $contenedor->getLogger();
```

Una última refactorización en el código del contenedor puede hacerlo todavía más útil. Para poder cambiar fácilmente las clases de los manejadores, haz que el propio nombre de la clase sea una opción del contenedor:

```
class Contenedor
{
```

```
    protected $opciones = array();

    public function __construct($opciones = array())
    {
        $this->opciones = $opciones;
    }

    public function getManejador()
    {
        switch($this->opciones['tipo']) {
            case 'db':
                $clase = $this->opciones['clase_db'];
                return new $clase(array('dsn' => $this->opciones['dsn']));

            case 'archivo':
            default:
                $clase = $this->opciones['clase_archivo'];
                return new $clase($this->opciones['ruta']);
        }
    }

    public function getLogger()
    {
        $logger = new Logger($this->getManejador());
        return $logger;
    }
}
```

Observa lo fácil que resulta ahora modificar la clase de cada manejador de mensajes de log:

```
// Guardar mensajes en un archivo
$contenedor = new Contenedor(array(
  'tipo' => 'archivo',
  'ruta' => __DIR__.'/debug.log',
));
$logger = $contenedor->getLogger();

// Simular en un test que se guardan mensajes en un archivo
$contenedor = new Contenedor(array(
  'tipo' => 'archivo',
  'ruta' => __DIR__.'/debug.log',
  'clase_archivo' => 'ArchivoMockTest'
));
$logger = $contenedor->getLogger();
```

Aunque el contenedor sólo define unos pocos parámetros, su uso se ha complicado excesivamente. El motivo es que los parámetros tienen nombres muy dispares. La buena práctica recomendada es nombrar los parámetros agrupándolos por su propósito:

```
// Si cada parámetro define su nombre, manejar el contenedor resulta difícil
$contenedor = new Contenedor(array(
  'tipo' => 'archivo',
  'ruta' => __DIR__.'/debug.log',
  'clase_archivo' => 'ArchivoMockTest'
  'dsn'  => 'mysql:host=localhost;dbname=logs',
  'clase_db' => 'MySQLMockTest'
));

// Buena práctica recomendada: usar namespaces en el nombre de los parámetros
$contenedor = new Contenedor(array(
  'logger.tipo'   => 'archivo',
  'archivo.ruta'  => __DIR__.'/debug.log',
  'archivo.clase' => 'ArchivoMockTest'
  'db.dsn'        => 'mysql:host=localhost;dbname=logs',
  'db.clase'      => 'MySQLMockTest'
));
```

Utilizar un contenedor simplifica mucho el uso de la inyección de dependencias en tu aplicación. El problema es que crear un contenedor con cientos de dependencias es una tarea titánica. Por eso Symfony incluye un completo contenedor de inyección de dependencias listo para usar.

Así, para definir servicios en el contenedor sólo tienes que crear una clase PHP y añadir unas pocas líneas en un archivo de configuración YAML, XML o PHP. Symfony transforma automáticamente esa configuración en el código PHP que realmente se ejecuta para instanciar dependencias, cargar opciones de configuración y crear los objetos solicitados por tu código.

B.2 La inyección de dependencias en Symfony

Hacer uso de la inyección de dependencias en las aplicaciones Symfony es incluso más sencillo de lo que se ha explicado en las secciones anteriores. De hecho, en la mayoría de los casos sólo debes programar tus clases PHP y después añadir unas pocas líneas en un archivo de configuración. El contenedor que incluye Symfony se encarga de procesar esa configuración y prepara todas las dependencias entre clases.

Si quieres utilizar la inyección de dependencias en tus propios proyectos PHP, puedes hacer uso del componente DependencyInjection (http://github.com/symfony/DependencyInjection) , que contiene exactamente el mismo código que utiliza Symfony. Si te basta con un micro contenedor de inyección de dependencias que sólo contenga lo imprescindible, puedes utilizar Pimple (http://pimple.sensiolabs.org/) , un contenedor completamente funcional en menos de 50 líneas de código.

B.2.1 Utilizando servicios desarrollados por terceros

Dentro de una aplicación Symfony, el término **servicio** hace referencia a cualquier clase/objeto manejada por el contenedor de inyección de dependencias. En la práctica los servicios son clases PHP que realizan cualquier tarea global dentro de la aplicación.

Cuando desarrollas una aplicación Symfony, lo habitual no es definir servicios en el contenedor de inyección de dependencias, sino utilizar los servicios que incluye Symfony y los que definen los *bundles* desarrollados por terceros. De hecho, a menos que la aplicación sea muy compleja, es posible desarrollar una aplicación Symfony completa sin crear ningún servicio propio.

Observa el siguiente código de un controlador imaginario de Symfony:

```
class DefaultController extends Controller
{
    public function portadaAction()
    {
        $direccionIp = $this->get('request')->getClientIp();
        $this->get('logger')->info('IP del usuario: '.$direccionIp);

        $em = $this->getDoctrine()->getManager();
        // ... consulta a la base de datos ...
        $this->get('logger')->info('Encontrados 50 resultados en la BD');

        $this->container->get('mailer')->send(...);
        $this->get('session')->setFlash('info',
            'Acabamos de enviarte un email con más información'
        );

        return $this->get('templating')->renderResponse(...);
    }
}
```

El código anterior es un buen ejemplo de cómo funciona y qué ventajas tiene el uso de un contenedor de inyección de dependencias. Los controladores que heredan de la clase `Controller` de Symfony disponen de un acceso directo al contenedor a través de los métodos `$this->get()` y `$this->container->get()`.

Las primeras líneas del controlador crean un nuevo mensaje de log para guardar la dirección IP del usuario que realiza la petición. La IP está disponible a través del método `getClientIp()` del objeto de tipo `Request` que crea Symfony para cada petición. Gracias al contenedor, obtener ese objeto de la petición es tan sencillo como:

```
$direccionIp = $this->get('request')->getClientIp();
```

Tu código no sabe cómo crear un objeto de la petición. No sabe ni qué clase hay que instanciar ni de qué otras clases depende ni qué parámetros utiliza para su configuración. Todo ese trabajo es responsabilidad del contenedor.

La instrucción `$this->get('request')` indica que tu código quiere hacer uso de un servicio llamado `request`. El contenedor de Symfony busca entre todos los servicios definidos aquel que se llame `request` y realiza todas las tareas necesarias para instanciar el objeto correspondiente y entregártelo.

Además de que tu código se vuelve extremadamente conciso y sencillo, observa la flexibilidad que ofrece el contenedor. Si tu aplicación tiene unos requerimientos muy específicos, puede ser necesario crear una clase propia para las peticiones (un objeto `Request` propio). Si ese es el caso, puedes modificar toda la aplicación sin cambiar ni una sola línea de código, tan sólo modificando la configuración que le dice al contenedor cómo crear el servicio `request`.

Siguiendo con el código del controlador anterior, observa cómo crea un mensaje de log:

```
$direccionIp = $this->get('request')->getClientIp();
$this->get('logger')->info('IP del usuario: '.$direccionIp);
```

Otra vez tu código sólo pide al contenedor que cree servicios (`$this->get('logger')`), sin preocuparse de cómo se crean o las dependencias que tienen con otras clases. El servicio que crea mensajes de log es un ejemplo claro de que no basta con decir el servicio que quieres utilizar, sino que, a veces, también es necesario añadir cierta información de configuración.

El servicio `logger` por ejemplo debe saber si los mensajes de log se guardan en un archivo, en una base de datos o si se envían por correo electrónico. Si se guardan en un archivo, también debe conocer por ejemplo su ruta. Así que cuando utilices servicios creados por terceros (o servicios propios de Symfony) lo primero que debes hacer es configurarlos adecuadamente.

Las opciones de configuración disponibles y sus posibles valores se indican en la documentación de Symfony o del *bundle* que estés utilizando. La mayoría de opciones de Symfony están documentadas en el apartado *Reference* de su sitio web (http://symfony.com/doc/current/reference/index.html) . El servicio `logger` por ejemplo se configura mediante las siguientes opciones en el archivo `app/config/config.yml` de Symfony:

```
monolog:
    handlers:
        main:
            type:  stream
            path:  %kernel.logs_dir%/%kernel.environment%.log
            level: debug
```

La opción `type: stream` indica que los mensajes de log se guardan en un archivo, la opción `path` establece la ruta del archivo (en este caso, `app/logs/dev.log` o `app/logs/prod.log`) y la opción `level` indica el nivel de alerta mínimo que deben tener los mensajes de log para guardarlos en el archivo.

Si más adelante quieres cambiar el archivo de los mensajes de log, sólo debes modificar la opción `path`, pero el código de la aplicación sigue siendo `$this->get('logger)`. Igualmente, si en vez de un archivo quieres utilizar una base de datos para guardar los mensajes, sólo debes cambiar la opción `type` del archivo de configuración, manteniendo sin cambios el código PHP de tu aplicación.

El código del controlador mostrado anteriormente incluye otros ejemplos de uso de servicios de Symfony. Las consultas a la base de datos se realizan mediante el *entity manager* de Doctrine:

```
$em = $this->getDoctrine()->getEntityManager();
```

El método `getDoctrine()` es un atajo disponible en todos los controladores que extienden de la clase `Controller` de Symfony. En realidad es equivalente a `$this->get('doctrine')`.

Por último, el controlador envía un email, muestra un mensaje *flash* al usuario y devuelve una plantilla renderizada con Twig. Cada una de estas acciones se realiza en una sola instrucción gracias al contenedor de Symfony:

```
$this->container->get('mailer')->send(...);

$this->get('session')->setFlash('info',
    'Acabamos de enviarte un email con más información'
);

return $this->get('templating')->renderResponse(...);
```

Esta filosofía de programación se denomina SOA, las siglas en inglés de *Arquitectura Orientada a Servicios*. Este es uno de los paradigmas de programación más utilizados y probados en el ámbito de la ingeniería del software.

B.2.2 Desarrollando servicios propios

La mayor parte de las clases de una aplicación Symfony son controladores, entidades de Doctrine, formularios y tests. No obstante, en un *bundle* puedes añadir tantas clases PHP propias como quieras. Estas clases también se pueden convertir con facilidad en servicios gestionados por el contenedor.

Imagina que dispones de una clase PHP llamada `Slugger` que genera el *slug* de cualquier cadena de texto. Esta clase se define en el archivo `src/AppBundle/Util/Slugger.php` y su código simplificado es el que se muestra a continuación:

```
// src/AppBundle/Util/Slugger.php
namespace AppBundle\Util;

class Slugger
{
    public function getSlug($cadena)
    {
        // ...
    }
}
```

Para crear un servicio a partir de esta clase, añade la siguiente configuración en el archivo `config.yml`:

```
# app/config/config.yml
services:
    app.slugger:
        class: AppBundle\Util\Slugger
```

¡Ya está! Ya has creado un servicio listo para el contenedor de inyección de dependencias de Symfony. Para ello sólo hay que asignar al servicio un nombre único (en este caso, `app.slugger`) bajo la opción `services`. Después, indica el *namespace* y la clase en la opción `class` del servicio.

Ahora ya puedes acceder desde tu código a esta clase igual que a cualquier otro servicio de Symfony:

```
class DefaultController extends Controller
{
    public function portadaAction()
    {
        // Acceder al servicio
        $slugger = $this->get('app.slugger');

        // Utilizar directamente un método del servicio
        $slug = $this->get('app.slugger')->getSlug($cadena);

        // ...
    }
}
```

Una ventaja añadida de la definición de servicios para las clases PHP propias es que no penalizan el rendimiento de la aplicación. En efecto, si tu código no hace uso de algún servicio, el contenedor nunca lo crea, por lo que puedes definir tantos servicios como quieras sin preocuparte por el rendimiento de tu aplicación.

B.2.2.1 Argumentos y parámetros

Al trabajar con servicios es muy habitual tener que pasarles información para su correcto funcionamiento. Imagina que al servicio definido anteriormente es necesario indicarle la codificación que utilizan los contenidos de la aplicación. La forma más sencilla de hacerlo consiste en añadir esa información en la opción `arguments` del servicio:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['utf-8']
```

Cuando hagas uso del servicio `app.slugger`, el contenedor instancia la clase indicada y le pasa al constructor todos los argumentos definidos en la opción `arguments` y en ese mismo orden. Así que no olvides añadir o modificar el constructor de la clase:

```
// src/AppBundle/Util/Slugger.php
namespace AppBundle\Util;

class Slugger
{
    private $codificacion;
```

```
    public function __construct($codificacion)
    {
        $this->codificacion = $codificacion;
    }

    // ...
}
```

No existe ningún límite en la cantidad o tipo de argumentos que puedes pasar a un servicio. No obstante, cuando el número de argumentos es muy grande puede llegar a resultar confuso, ya que no se puede indicar el nombre de cada argumento:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['utf-8', true, 10, '...', null]
```

Cuando quieres reutilizar un mismo argumento en varios servicios, la solución consiste en crear un parámetro del contenedor de servicios. Estos parámetros se definen bajo la clave `parameters`:

```
# app/config/config.yml
parameters:
    codificacion: 'utf-8'

services:
    # ...
```

Para referirte a un parámetro en cualquier lugar de un archivo de configuración de Symfony, encierra su nombre entre `%`:

```
# app/config/config.yml
parameters:
    codificacion: 'utf-8'

services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: [%codificacion%]
```

Una buena práctica al definir parámetros consiste en prefijar su nombre con el del servicio en el que se utilizan. Así, en vez de `codificacion` el parámetro debería llamarse `app.slugger.codificacion`. Si se trata de un parámetro utilizado en varios servicios, prefíjalo al menos con el nombre del proyecto (en este ejemplo, `cupon.codificacion`).

Además de los parámetros propios, el contenedor de Symfony define varios parámetros que también puedes utilizar en tu código encerrando su nombre entre `%`:

Parámetro	Valor
kernel.environment	Nombre del entorno en el que se está ejecutando la aplicación (dev, prod, etc.)
kernel.cache_dir	Ruta del directorio de la caché de la aplicación (normalmente app/cache/%kernel.environment%)
kernel.logs_dir	Ruta del directorio de logs de la aplicación (normalmente app/logs)
kernel.debug	Indica si la aplicación se está ejecutando con la depuración activada (por defecto vale false en el entorno prod y true en el entorno dev)
kernel.name	Nombre del kernel de la aplicación (normalmente app)
kernel.bundle	Array con todos los bundles activos en la aplicación

La opción arguments inyecta los parámetros mediante el constructor, ya que es el caso más común. Pero el contenedor de Symfony también soporta la inyección de parámetros mediante métodos *setter*:

```
# app/config/config.yml
parameters:
    cupon.codificacion: 'utf-8'

services:
    app.slugger:
        class: AppBundle\Util\Slugger
        calls:
            - [ setCodificacion, [%cupon.codificacion%] ]
```

La opción calls define la lista de métodos que se ejecutan al instanciar la clase del servicio y los argumentos que se le pasan a cada *setter*. La configuración anterior le indica a Symfony que al crear el servicio app.slugger debe invocar el método setCodificacion() de la clase Util y le debe pasar como argumento el valor del parámetro %cupon.codificacion% del contenedor.

Siguiendo con el ejemplo anterior, sería necesario modificar la clase Slugger para añadir el *setter* que va a invocar el contenedor:

```
// src/AppBundle/Util/Slugger.php
namespace AppBundle\Util;

class Slugger
{
    private $codificacion;

    public function setCodificacion($codificacion)
    {
        $this->codificacion = $codificacion;
    }
```

```
    // ...
}
```

B.2.2.2 Creando dependencias entre servicios

En las aplicaciones web complejas es habitual que existan muchas dependencias entre unos servicios y otros. El servicio `app.slugger` definido anteriormente podría por ejemplo tener que generar URL a partir de las rutas de la aplicación. En ese caso, el servicio debe configurar una dependencia con el servicio `router`.

Si el servicio `app.slugger` también añade información de depuración en los archivos de log, debería configurar otra dependencia con el servicio `logger` de Symfony.

Para definir las dependencias con otros servicios, sólo tienes que *inyectarlos* mediante la opción `arguments` del servicio:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@logger', '@router']
```

Cuando el valor de un argumento contiene el prefijo `@`, el contenedor lo interpreta como el nombre de un servicio. Así, gracias a la configuración anterior, cuando hagas uso del servicio `app.slugger`, el contenedor creará primero los servicios `logger` y `router` para pasárselos al constructor de la clase de tu servicio.

No olvides actualizar el código de tu servicio para recoger en su constructor los nuevos objetos que le pasa el contenedor:

```
// src/AppBundle/Util/Slugger.php
class Slugger
{
    private $logger, $router;

    public function __construct($logger, $router)
    {
        $this->logger = $logger;
        $this->router = $router;
    }

    public function getSlug($cadena)
    {
        $this->logger->info(...);

        $this->router->generate(...);

        // ...
    }
}
```

La siguiente lista muestra el nombre de los servicios de Symfony más utilizados:

- `doctrine`, servicio definido por Doctrine, a través del cual se obtiene el *entity manager*.
- `event_dispatcher`, servicio definido por el componente encargado de crear y notificar los eventos que se producen durante la ejecución de las peticiones de los usuarios.
- `logger`, servicio de logs creado por el componente Monolog.
- `mailer`, servicio de envío de emails creado por el componente Swift Mailer.
- `profiler`, servicio definido por el *profiler* de Symfony y que permite acceder a toda la información sobre la ejecución y rendimiento de cada petición.
- `security.token_storage`, servicio definido por el componente de seguridad que se utiliza para obtener el objeto que representa al usuario *logueado*.
- `security.authorization_checker`, servicio definido por el componente de seguridad que se utiliza para comprobar si el usuario puede acceder al recurso solicitado.
- `service_container`, servicio que representa al propio contenedor de servicios y por tanto, da acceso a cualquier otro servicio. Por motivos de rendimiento, se recomienda inyectar servicios específicos siempre que sea posible en vez del contenedor entero.
- `session`, servicio que representa a la sesión del usuario actual.
- `templating`, servicio del componente de plantillas activo en la aplicación. Por defecto en Symfony este servicio permite acceder a Twig.
- `translator`, servicio del componente de traducción e internacionalización. Muy útil para traducir contenidos en controladores o en clases PHP propias.
- `validator`, servicio del componente de validación con el que puedes validar datos y entidades de forma independiente a los formularios.

B.2.2.3 Fragmentando la configuración de los servicios

A pesar de que el archivo de configuración `app/config/config.yml` es un archivo muy grande y contiene decenas de opciones, si añades muchos servicios puedes convertirlo en un archivo inmanejable.

Gracias a la flexibilidad de los archivos de configuración de Symfony, puedes solucionar fácilmente este problema. Lo más sencillo consiste en colocar toda la información de los servicios en un nuevo archivo de configuración llamado `app/config/services.yml` e importar sus contenidos desde `config.yml`:

```
# app/config/config.yml
imports:
    - { resource: parameters.ini }
    - { resource: security.yml }
    - { resource: services.yml }

# ...
```

```
# app/config/services.yml
parameters:
    # ...

services:
  app.slugger:
      # ...
```

Otra solución recomendada es incluir la configuración de los servicios de cada *bundle* en su propio *bundle*. Así por ejemplo, el servicio `app.slugger` se definiría en el *bundle* `AppBundle`. Si te fijas en el contenido de cualquier *bundle*, verás que Symfony crea un archivo llamado `services.yml` dentro del directorio `Resources/config/`. Por defecto su contenido es el siguiente:

```
# src/AppBundle/Resources/config/services.yml

parameters:
#    oferta.example.class: AppBundle\Example

services:
#    oferta.example:
#        class: %oferta.example.class%
#        arguments: [@service_id, "plain_value", %parameter%]
```

Elimina todas las líneas comentadas y define tus propios parámetros y servicios. Después, sólo tienes que importar la configuración del *bundle* desde el archivo `app/config/config.yml`:

```
# app/config/config.yml
imports:
    - { resource: parameters.ini }
    - { resource: security.yml }
    oferta_bundle:
        resource: @OfertaBundle/Resources/config/services.yml

# ...
```

Para importar un archivo de un *bundle*, añade primero un nombre único (en este caso, `oferta_bundle`) bajo la opción `imports`. Después, indica con la opción `resource` la ruta del archivo utilizando la *notación bundle*: el prefijo `@` indica que la primera palabra es el nombre del *bundle* y el resto, la ruta relativa del archivo dentro del *bundle*.

B.2.3 Definiendo los servicios especiales

Además de los servicios que ya incluye Symfony y de los servicios propios que puedes definir en tu aplicación, existe un tercer tipo especial de servicios. Se trata de servicios que deben definirse de una manera especial impuesta por Symfony.

B.2.3.1 Extensiones de Twig

Las extensiones propias de Twig son el mejor ejemplo de servicio especial, ya que aunque son servicios creados por tu aplicación, para que funcionen bien deben definirse tal y como establece Symfony.

Imagina que has creado una extensión llamada `CuponExtension` dentro del *bundle* `AppBundle`:

```
// src/AppBundle/Twig/Extension/CuponExtension.php
namespace AppBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    // ...
}
```

El siguiente paso sería definir un servicio para la extensión utilizando la misma configuración que se ha explicado en las secciones anteriores:

```
# app/config/config.yml
services:
    twig.extension.cupon:
        class: AppBundle\Twig\Extension\CuponExtension
```

El problema es que de esta forma Twig no activa tu extensión y por tanto, no puedes usar sus filtros y funciones en ninguna plantilla. La clave para activarla es añadir una opción llamada `tags` con el valor `twig.extension` (la opción `tags` se explica más adelante en la sección de opciones avanzadas del contenedor):

```
# app/config/config.yml
services:
    twig.extension.cupon:
        class: AppBundle\Twig\Extension\CuponExtension
        tags:
            - { name: twig.extension }
```

Este mismo método es el que se utiliza para activar las extensiones `Text` y `Debug` que incluye Twig pero que se encuentran desactivadas por defecto:

```
# app/config/config.yml
services:
    twig.extension.text:
        class: Twig_Extensions_Extension_Text
        tags:
            - { name: twig.extension }

    twig.extension.debug:
        class: Twig_Extensions_Extension_Debug
        tags:
            - { name: twig.extension }
```

B.2.3.2 Eventos

Symfony incluye un completo sistema de eventos para que tu aplicación pueda ejecutar código antes o después de que se produzca un evento destacable. Para ello primero debes crear una clase encargada de procesar el evento, como por ejemplo la siguiente:

```
// src/AppBundle/Listener/RequestListener.php
namespace AppBundle\Listener;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        // ...
    }
}
```

Para activar este evento en la aplicación, no basta con definir un servicio simple, sino que debes añadir la opción `tags` con el valor `kernel.event_listener`. Además, en este caso también debes añadir la opción `event` indicando el nombre del evento al que responde tu código:

```
# app/config/config.yml
services:
    cupon.ofertabundle.listener.request:
        class: AppBundle\Listener\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request }
```

Si al procesar la petición del usuario se notifica el evento `kernel.request`, Symfony ejecuta el evento `onKernelRequest()` de tu clase `RequestListener`. Si el nombre del método no sigue esa nomenclatura, añade la opción `method` para indicar el nombre del método:

```
# app/config/config.yml
services:
    cupon.ofertabundle.listener.request:
        class: AppBundle\Listener\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request,
                method: setNuevoFormatoPeticion }
```

Otros de los eventos más populares en Symfony son los eventos definidos por Doctrine, que permiten modificar la información de las entidades antes o después de guardarlas en la base de datos. El siguiente ejemplo hace que se ejecute la clase `Normalizar` cuando se notifique el evento `prePersist`, es decir, justo antes de guardar la entidad en la base de datos:

```
services:
    cupon.ofertabundle.listener.normalizar:
```

```
        class: AppBundle\Listener\Normalizar
        tags:
            - { name: doctrine.event_listener, event: prePersist }
```

Doctrine define los siguientes eventos:

- `preRemove`, se notifica justo antes de que el *entity manager* elimine la entidad.
- `postRemove`, se notifica una vez que el objeto de la entidad ya ha sido borrado. Después de ejecutar las sentencias SQL que borran la información en la base de datos.
- `prePersist`, se notifica justo antes de guardar la información de la entidad en la base de datos.
- `postPersist`, se notifica una vez que la información de la entidad ya ha sido guardada en la base de datos. Después de ejecutar las sentencias SQL que insertan la información, por lo que el atributo `id` ya tiene el valor de la clave primaria generada.
- `preUpdate`, se notifica justo antes de que la base de datos actualice la información de la entidad.
- `postUpdate`, se notifica después de ejecutar las sentencias SQL que actualizan la información de la entidad en la base de datos.
- `postLoad`, se notifica justo después de que el *entity manager* haya cargado desde la base de datos la información de una entidad. También se notifica después de ejecutar `refresh()` sobre la entidad.
- `loadClassMetadata`, se notifica justo después de que se haya cargado la información de *mapping* de Doctrine. Dependiendo del formato que utilices, este evento se notifica después de procesar las anotaciones de la entidad o después de leer el archivo XML o YAML que configura la entidad.

B.2.4 Obteniendo el contenedor de inyección de dependencias

Como los servicios de Symfony se obtienen a través del contenedor de inyección de dependencias, resulta esencial tener acceso al contenedor desde cualquier punto de la aplicación.

A lo largo de todos los capítulos anteriores de este libro se explica detalladamente cómo obtener el contenedor en diferentes situaciones. A continuación se resume brevemente todas esas explicaciones.

B.2.4.1 Controladores

Symfony inyecta automáticamente el contenedor a todos los controladores que heredan de la clase `Controller`. Puedes acceder al contenedor a través de `$this->container`. Por tanto, para obtener un servicio utiliza cualquiera de los siguientes métodos:

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    public function portadaAction()
```

```
    {
        $logger = $this->container->get('logger');

        // atajo equivalente a la línea anterior
        $logger = $this->get('logger');

        // ...
    }
}
```

La clase `Controller` de Symfony añade también varios atajos para obtener los servicios más utilizados en los controladores:

- Petición:
  - Atajo: `$this->getRequest()`
  - Código equivalente: `$this->container->get('request')`
- Doctrine:
  - Atajo: `$this->getDoctrine()`
  - Código equivalente: `$this->container->get('doctrine')`
- Formularios:
  - Atajo: `$this->createForm(...)`
  - Código equivalente: `$this->container->get('form.factory')->create(...)`
- Enrutamiento:
  - Atajo: `$this->generateUrl(...)`
  - Código equivalente: `$this->container->get('router')->generate(...)`
- Redirecciones:
  - Atajo: `$this->forward(...)`
  - Código equivalente: `$this->container->get('http_kernel')->forward(...)`
- Plantillas:
  - Atajo: `$this->renderView(...)`
  - Código equivalente: `$this->container->get('templating')->render(...)`

B.2.4.2 Archivos de datos o *fixtures*

El contenedor de inyección de dependencias no está disponible por defecto en los archivos de datos o *fixtures*. Para acceder al contenedor debes implementar la interfaz `ContainerAwareInterface` y añadir un método `setContainer()` para recoger el objeto contenedor que le pasa Symfony:

```
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class Usuarios implements FixtureInterface, ContainerAwareInterface
{
    private $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }

    public function load(ObjectManager $manager)
    {
        $logger = $this->container->get('logger');

        // ...
    }
}
```

B.2.4.3 Comandos

Para disponer del contenedor de dependencias en un comando, sólo debes cambiar la clase de la que hereda el comando. En vez de la habitual clase `Command`, hereda de la clase `ContainerAwareCommand`:

```
// Antes
use Symfony\Component\Console\Command\Command;

class EmailOfertaDelDiaCommand extends Command
{
    // ...
}

// Ahora
use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;

class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...
}
```

Después de este cambio tan sencillo, ya puedes acceder al contenedor dentro del código del comando mediante el método `$this->getContainer()`:

```
use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;

class EmailOfertaDelDiaCommand extends ContainerAwareCommand
```

```
{
    // ...

    $logger = $this->getContainer()->get('logger');
}
```

B.2.4.4 Clases PHP propias

Imagina que dispones de una clase PHP llamada `Slugger` para generar el *slug* de cualquier cadena de texto. Esta clase se define en el archivo `src/AppBundle/Util/Slugger.php` y su código simplificado es el que se muestra a continuación:

```
// src/AppBundle/Util/Slugger.php
namespace AppBundle\Util;

class Slugger
{
    public function getSlug($cadena)
    {
        // ...
    }
}
```

Aunque de primeras puede resultar confuso, para acceder al contenedor dentro de una clase propia, debes definir un servicio para la clase y después inyectarle como argumento el propio contenedor de dependencias, que también es un servicio. Define en primer lugar un servicio para la clase:

```
# app/config/config.yml
services:
    app.slugger:
        class: AppBundle\Util\Slugger
```

A continuación, inyéctale como argumento el propio contenedor de servicios, disponible a través de un servicio llamado `service_container`:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@service_container']
```

Por último, añade un constructor en la clase `Slugger` para recoger el objeto del contenedor que le pasa Symfony:

```
class Util
{
    private $container;

    public function __construct($container)
    {
```

```
        $this->container = $container;
    }

    static public function getSlug($cadena)
    {
        $logger = $this->container->get('logger');

        // ...
    }
}
```

B.2.5 Características avanzadas

El contenedor de servicios o contenedor de inyección de dependencias de Symfony dispone de muchas otras opciones y características avanzadas, como las que se enumeran a continuación.

B.2.5.1 Depurando el contenedor

Symfony incluye un comando llamado `debug:container` para depurar el contenedor de servicios. Si lo ejecutas verás el listado completo de servicios de la aplicación, por lo que es ideal para comprobar si tus servicios están disponibles en la aplicación o por si el contrario no han sido definidos correctamente:

```
$ php app/console debug:container

# Symfony Container Public Services
-----------------   ---------------------------------------------------------------
Service ID          Class name
-----------------   ---------------------------------------------------------------
annotation_reader   Doctrine\Common\Annotations\CachedReader
cache_clearer       Symfony\Component\HttpKernel\CacheClearer\ChainCacheClearer
cache_warmer        Symfony\Component\HttpKernel\CacheWarmer\CacheWarmerAggregat
e
...
validator           Symfony\Component\Validator\Validator\ValidatorInterface
-----------------   ---------------------------------------------------------------

La información que proporciona el comando `debug:container` es muy completa pero
difícil de procesar, ya que existen decenas de servicios y sus clases tienen
unos nombres demasiado largos como para visualizar sus relaciones.

Si quieres depurar el contenedor de una forma muy visual, instala el *bundle*
[JMSDebuggingBundle](https://github.com/schmittjoh/JMSDebuggingBundle). Una vez
instalado, el *bundle* añade un nuevo panel en el *profiler* de Symfony que
incluye una representación gráfica de las relaciones entre los diferentes
servicios de la aplicación.

#### Servicios privados ####
```

```
El comando anterior solamente muestra los servicios públicos definidos en el
contenedor. Por defecto todos los servicios de Symfony son públicos. No
obstante, en ocasiones un servicio se define sólo para servir de argumento de
otro servicio, por lo que no es necesario que esté disponible para los
programadores de la aplicación a través del contenedor.

Si necesitas definir un servicio como privado, añade la opción `public: false`
al definir el servicio:

```yaml
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@service_container']
        public:    false
```

Los servicios privados no se pueden obtener de ninguna manera a través del contenedor. Así que la configuración anterior provoca que todas las instrucciones `$this->get('app.slugger')` dejen de funcionar y muestren un error de tipo *"You have requested a non-existent service app.slugger"*.

El comando `debug:container` también permite ver los servicios privados de la aplicación. Para ello, añade la opción `--show-private`:

```
$ php app/console debug:container --show-private

[container] Public and private services
Name                 Scope       Class Name
-------------------- ----------- ------------------------------------------------
353a5...f9e7ee1f4_1 container   [...]\Controller\ControllerResolver
353a5...f9e7ec01_10 container   [...]\DependencyInjectionExtension
353a5...f9e7ec0_100 container   Doctrine\ORM\Mapping\Driver\DriverChain
353a5...f9e7ec0_101 container   Doctrine\ORM\Mapping\Driver\AnnotationDriver
annotation_reader   container   Doctrine\Common\Annotations\FileCacheReader
assetic.controller  prototype   Symfony\Bundle\[...]\AsseticController
cache_warmer        container   [...]\CacheWarmer\CacheWarmerAggregate
database_connection n/a         alias for doctrine.dbal.default_connection
doctrine            container   Symfony\Bundle\DoctrineBundle\Registry
# ...
twig                container   Twig_Environment
twig.loader         container   [...]\TwigBundle\Loader\FilesystemLoader
validator           container   Symfony\Component\Validator\Validator
```

La mayoría de servicios privados tienen un nombre muy largo generado aleatoriamente, ya que son servicios internos generados automáticamente por Symfony.
```

B.2.5.2 Servicios opcionales

Aunque no resulta habitual, es posible que alguno de tus servicios pueda funcionar sin alguna de las dependencias definidas en su configuración. El siguiente servicio por ejemplo requiere el uso del servicio `logger`de Symfony:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@logger']
```

Si tu servicio puede funcionar bien sin el servicio `logger`, puedes convertir esta dependencia en opcional añadiendo el carácter `?` entre la `@` y el nombre del servicio:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@?logger']
```

Si el servicio `logger` existe en la aplicación, el contenedor lo inyectará normalmente y tu servicio podrá hacer uso de el. La diferencia se produce cuando el servicio `logger` no exista o no esté activado en la aplicación. El contenedor simplemente ignorará esta dependencia, no la inyectará y no se mostrará ningún mensaje de error.

El único ajuste que debes hacer en el código de tus servicios es prepararlo para cuando no existan las dependencias:

```
// Antes
class Slugger
{
    private $logger;

    public function __construct($logger)
    {
        $this->logger = $logger;
    }

    // ...
}

// Ahora
class Slugger
{
    private $logger;

    public function __construct($logger = null)
    {
        $this->logger = $logger;
```

```
    }

    // ...
}
```

B.2.5.3 Aliases

Como los servicios deben tener un nombre único dentro de la aplicación, el resultado suelen ser nombres muy largos, normalmente formados por varias palabras separadas por puntos, a modo de *namespace*: `twig.extension.debug`, `app.listener.request`, etc.

La solución que ofrece el contenedor es la posibilidad de crear *alias* para hacer referencia a un servicio mediante otro nombre, normalmente mucho más corto. Para ello sólo debes definir un nuevo servicio e indicar en su opción `alias` el nombre del servicio para el que es un alias:

```
# app/config/config.yml
services:
    app.slugger:
        class:     AppBundle\Util\Slugger
        arguments: ['@service_container']

    slugger:
        alias: app.slugger
```

Con la configuración anterior, ya puedes acceder a la clase de utilidades mediante el código `$this->get('slugger')`.

Los alias también funcionan con los servicios privados. Así puedes exponer a la aplicación un servicio propio utilizando un nombre genérico. Esta es la técnica que usa el componente Monolog. Internamente define un servicio privado llamado `monolog.logger`, pero lo expone públicamente mediante el alias `logger`:

```
# ...
services:
    # ...
    monolog.logger:
        # ...
        public: false

    logger:
        alias: monolog.logger
```

B.2.5.4 Etiquetas

Al definir un servicio en el contenedor, se le pueden añadir varias `tags` o etiquetas. Esto hace que el contenedor trate al servicio de una manera especial. El uso más común de las etiquetas en Symfony es la definición de una extensión propia de Twig:

```
# app/config/config.yml
services:
    twig.extension.cupon:
```

```
        class: AppBundle\Twig\Extension\CuponExtension
        arguments: ['@translator']
        tags:
            - { name: twig.extension }
```

La etiqueta `twig.extension` indica a Symfony que este servicio en realidad es una extensión de Twig. Por tanto, cuando en el código de la aplicación se solicite al contenedor el uso del servicio Twig, el contenedor instanciará y activará automáticamente todas sus extensiones, de decir, todos los servicios que hayan sido etiquetados con `twig.extension`.

El uso de las etiquetas es tan específico, que resulta muy poco habitual definir nuevas etiquetas en el contenedor. Normalmente tus servicios hacen uso de alguna de las etiquetas definidas por Symfony o por algún *bundle* desarrollado por terceros.

Las principales etiquetas disponibles en Symfony son las siguientes:

- `console.command`
- `data_collector`
- `form.type`
- `kernel.cache_warmer`
- `kernel.event_listener`
- `kernel.event_subscriber`
- `monolog.logger`
- `routing.loader`
- `security.voter`
- `twig.extension`
- `validator.constraint_validator`

Consulta este artículo (http://symfony.com/doc/2.8/reference/dic_tags.html) para ver el listado completo actualizado de etiquetas Symfony.

B.2.5.5 Incluyendo archivos

En algunas ocasiones puede ser útil incluir un archivo PHP antes de que el servicio se cargue en el contenedor y por tanto, antes de instanciar las clases. Este es el caso de los servicios creados para integrar aplicaciones PHP externas a Symfony como WordPress, Drupal o Magento.

Para incluir un archivo PHP, indica su ruta en la opción `file` del servicio. Symfony hará un `require_once()` del archivo antes de crear el servicio:

```
# app/config/config.yml
services:
    app.slugger:
        class: AppBundle\Util\Slugger
        file:  %kernel.root_dir%/../src/AppBundle/Util/inicial.php
```

Esta página se ha dejado vacía a propósito

Sobre el autor

Javier Eguíluz es un formador especializado en nuevas tecnologías. Su pasión es la programación, sobre todo el desarrollo de sitios y aplicaciones web.

Javier es miembro de la comunidad Symfony desde finales de 2006. En febrero de 2007 fundó el sitio symfony.es (http://symfony.es) y desde entonces trabaja para promocionar y extender el uso de Symfony. Además de traducir la mayor parte de los libros de Symfony 1, ha impartido numerosos cursos y jornadas sobre el framework.

Puedes ver todas sus presentaciones sobre Symfony en slideshare.net/javier.eguiluz (http://www.slideshare.net/javier.eguiluz/presentations) .

Contacto y perfiles

- Email de contacto: **javier.eguiluz@gmail.com**
- twitter.com/javiereguiluz (http://twitter.com/javiereguiluz)
- linkedin.com/in/javiereguiluz (http://www.linkedin.com/in/javiereguiluz)
- connect.sensiolabs.com/profile/javier.eguiluz (http://connect.sensiolabs.com/profile/javier.eguiluz)

Esta página se ha dejado vacía a propósito

Agradecimientos

Este libro no hubiera sido posible sin la ayuda y el trabajo de muchas personas. Gente que publica artículos sobre Symfony en sus blogs, gente que responde a las dudas planteadas en listas de correo y gente que envía *pull requests* para mejorar el código fuente de Symfony.

Entre todas esas personas anónimas, merecen ser destacadas las siguientes por su gran contribución a la documentación oficial de Symfony:

- Fabien Potencier (http://github.com/fabpot)
- Ryan Weaver (http://github.com/weaverryan)
- Wouter de Jong (http://github.com/WouterJ)
- Christian Flothmann (http://github.com/xabbuh)
- Richard Miller (http://github.com/richardmiller)
- Christophe Coevoet (http://github.com/stof)
- Hugo Hamon (http://github.com/hhamon)

Además, también merecen ser destacadas todas aquellas personas que han contribuido de forma directa en este libro con sus mejoras, sugerencias y correcciones de errores ortográficos y técnicos:

- Albert Jessurum (http://github.com/ajessu)
- Jordi Llonch (http://github.com/jordillonch)
- Javier López (http://github.com/loalf)
- Óscar López Carazo (http://github.com/Osukaru)
- Mario Nunes (http://github.com/mariotux)
- Edgar Rojas (http://github.com/brainhell)
- Asier Marqués (http://github.com/asiermarques)
- Raúl Fraile (http://github.com/raulfraile)
- José Francisco Ibarra
- Juan Carlos Romero (http://github.com/jcromero70)
- Sergio Gómez (http://github.com/sgomez)
- Sergio Rael (http://github.com/dorogoy)
- Rafa Couto (http://github.com/rafacouto)
- Raúl Araya (http://github.com/nubeiro)
- Francisco Gimeno (http://github.com/kikov79)
- Antonio García Marín
- Juan Luis Rodríguez Iglesias (http://github.com/aprendizenlared)
- Jorge Fabá Ferrández (http://github.com/jfabaf)

- Sergio Moya (http://github.com/smoya)
- Juan Baixauli (http://github.com/jbaixauli)
- Juan Salas (http://github.com/ratasxy)
- Jonatan Ginory
- Carlos Beato Ortega
- Fernando Mancera (http://github.com/fmancera)
- José Morales Lemus
- Rodrigo Miranda (http://github.com/rodrigomiranda)
- Jesús J. Briceño
- David Durán
- Francisco Suárez Mulero
- David Castelló (http://github.com/dcastello)
- Julio Álvarez Palacios
- Cristian Martin (http://github.com/proclamo)
- José Ramón Afonso (http://github.com/worvast)
- Marcelo Prizmic (http://github.com/mprizmic)
- José Gilberto Mullor (http://github.com/jgmullor)
- Raúl Gómez (http://github.com/raulggm)
- Magd Kudama
- Marcos Matamala
- Alberto Montero
- Fabricio Salinas
- Ricardo Montañana (http://github.com/rmontanana)
- Jaime Suez (http://github.com/jaimesuez)
- Álvaro Martín
- Faustino Vasquez Limon
- Loïc Vernet (http://github.com/COil)
- Francisco Espinosa Fuertes (http://github.com/chechu71)
- Alejandro Leal Cruz
- Francisco Javier Núñez Berrocoso (http://github.com/javiernuber)
- Luis Ignacio Bacas Riveiro
- Marcos Labad (http://github.com/esmiz)
- Neftalí Acosta

- Danny Viana
- Miguel Alcántara
- Miguel Angel Becerra Martín
- Jose Maria Diaz Angulo
- Ernesto José Vargas Paz (http://github.com/ejosvp)
- Rubén González González (http://github.com/rubenrua)
- José Daniel Penin Lovera
- Marcos García (http://github.com/marcosgdf)
- Jose Andrés Puertas (http://github.com/joseandrespg)
- Nicolás Moreira (http://github.com/nicolasmoreira)
- Erick Blangino
- Juan Manuel Rey
- Asier Ramos (http://github.com/uthopiko)
- Jesús Moreno Amor (http://github.com/jmorenoamor)
- Richard R. Pérez Q.
- Carlos Alfonso Pérez Rivera
- Fran Moreno (http://github.com/franmomu)
- Diego Ortega
- Renzo Álvarez
- Fernando Poyato (http://github.com/yatitos)
- Eduardo Antón
- Eduardo Basalo (http://github.com/eduardobape)
- Salvador Galiano
- Alberto Vioque (http://github.com/mashware)
- William Javier Del Valle Meza (http://github.com/williamdelvalle)
- Eduardo Basalo Peña (http://github.com/eduardobape)
- Ricard Clau (http://github.com/ricardclau)
- Alejandro Hiniesta
- Salvador Galiano López
- Ramiro Anacona Meneses (http://github.com/anacona16)
- Félix Carrasco del Pozo
- Jesús Briceño Moya
- Guillermo José Martínez Carmona

- Gerardo Sánchez Sánchez
- Fernando Pascua García
- Juan García Ripa
- Braulio Soncco (http://github.com/soncco)
- Uriel Francisco Romero Redondo
- Arnau González (http://github.com/arnaugm)
- Guillermo González de Agüero (http://github.com/ggam)
- Jose Carlos Liebana (http://github.com/Greibit)
- Luis González (http://github.com/luismagr)
- José Carmelo Molina Castro
- Marc Morales (http://github.com/mrcmorales)

Envía tus comentarios, sugerencias y correcciones a **javier.eguiluz@gmail.com** para aparecer en la lista de la próxima edición.