

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 19-11-2013 haciendo uso del gestor de publicaciones easybook versión 5.0-DEV, una herramienta para publicar libros que ha sido desarrollada con varios componentes de Symfony2 (<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 2013 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 Symfony2	19
1.3 Introducción a PHP 5.3	20
1.4 Introducción a YAML	25
1.5 Introducción a HTML5	27
1.6 Introducción a Git	28
Capítulo 2 El proyecto	33
2.1 Funcionamiento detallado de la aplicación	33
2.2 Wireframes	35
2.3 La base de datos	39
2.4 Aplicando la filosofía de Symfony2	41
2.5 Entidades	41
2.6 Bundles	41
2.7 Enrutamiento	42
Capítulo 3 Instalando y configurando Symfony2	45
3.1 Instalación	46
3.2 Configurando el entorno de ejecución	54
3.3 Actualizando Symfony2	60
3.4 Creando los <i>bundles</i>	60
Sección 2 Frontend	67
Capítulo 4 Creando las primeras páginas	69
4.1 La filosofía de Symfony2	69
4.2 La primera página	70
4.3 Creando todas las páginas estáticas	74
4.4 Configurando <i>la barra del final</i> en las URL	77
Capítulo 5 La base de datos	79
5.1 Entidades	79

5.2 Creando y configurando la base de datos	97
5.3 El <i>Entity Manager</i>	99
5.4 Archivos de datos o <i>fixtures</i>	105
5.5 Alternativas para generar el modelo	113
Capítulo 6 Creando la portada	117
6.1 Arquitectura MVC	117
6.2 El enrutamiento.....	118
6.3 El controlador	119
6.4 La plantilla	120
6.5 Entornos de ejecución.....	126
6.6 Depurando errores	131
6.7 Refactorizando el Controlador	140
6.8 Refactorizando el Modelo.....	145
6.9 Refactorizando la Vista.....	151
6.10 Funcionamiento interno de Symfony2	155
6.11 El objeto Request	157
6.12 El objeto Response.....	161
Capítulo 7 Completando el frontend	165
7.1 Herencia de plantillas a tres niveles.....	165
7.2 Hojas de estilo y archivos JavaScript.....	169
7.3 Seleccionando la ciudad activa	173
7.4 Creando la página de detalle de una oferta	180
7.5 Completando las plantillas con extensiones de Twig	186
7.6 Creando la página de ofertas recientes de una ciudad.....	192
7.7 Creando la portada de cada tienda	196
7.8 Refactorización final.....	200
Capítulo 8 Registrando usuarios	207
8.1 Creando la página de compras recientes	207
8.2 Restringiendo el acceso	212
8.3 Creando proveedores de usuarios	217
8.4 Añadiendo el formulario de <i>login</i>	219
8.5 Modificando las plantillas	230
8.6 Creando los archivos de datos de usuarios	235

8.7 Formulario de registro.....	237
8.8 Visualizando el perfil del usuario	263
Capítulo 9 RSS y los formatos alternativos	271
9.1 Formatos alternativos	271
9.2 Generando el RSS de las ofertas recientes de una ciudad	272
9.3 Generando el RSS de las ofertas recientes de una tienda	277
9.4 Registrando nuevos formatos	280
Capítulo 10 Internacionalizando el sitio web	283
10.1 Configuración inicial.....	283
10.2 Rutas internacionalizadas	284
10.3 Traduciendo contenidos estáticos	289
10.4 Traduciendo contenidos dinámicos.....	297
10.5 Traduciendo páginas estáticas.....	303
10.6 Traduciendo fechas	304
Capítulo 11 Tests unitarios y funcionales.....	307
11.1 Primeros pasos	307
11.2 Tests unitarios	308
11.3 Test funcionales.....	319
11.4 Configurando PHPUnit en Symfony2.....	336
Sección 3 Extranet.....	341
Capítulo 12 Planificación	343
12.1 Bundles	343
12.2 Enrutamiento	344
12.3 Layout.....	345
Capítulo 13 Seguridad	349
13.1 Definiendo la nueva configuración de seguridad.....	349
13.2 Preparando el proveedor de usuarios de las tiendas	352
13.3 Creando el formulario de login	354
13.4 Creando un <i>security voter</i> propio.....	359
Capítulo 14 Creando la parte de administración	365
14.1 Creando la portada de la <i>extranet</i>	365

14.2 Mostrando las ventas de una oferta	369
14.3 Mostrando el perfil de la tienda	371
Capítulo 15 Administrando las ofertas	379
15.1 Creando ofertas	379
15.2 Modificando las ofertas	393
Sección 4 Backend	401
Capítulo 16 Planificación	403
16.1 Bundles	403
16.2 Layout	404
16.3 Seguridad	408
Capítulo 17 Admin generator	413
17.1 <i>Admin generator</i> manual	413
17.2 Generador de código de Symfony2	420
17.3 SonataAdminBundle	432
Capítulo 18 Newsletters y comandos de consola	435
18.1 Creando comandos de consola	435
18.2 Generando la <i>newsletter</i> de cada usuario	448
18.3 Enviando la newsletter	453
Capítulo 19 Mejorando el rendimiento	459
19.1 Desactivando los elementos que no utilizas	459
19.2 Mejorando la carga de las clases	460
19.3 Mejorando el rendimiento del enrutamiento	461
19.4 Mejorando el rendimiento de la parte del cliente	464
19.5 Mejorando el rendimiento de Doctrine2	476
19.6 Mejorando el rendimiento de la aplicación con cachés	486
Capítulo 20 Caché	487
20.1 La caché del estándar HTTP	487
20.2 Estrategias de caché	489
20.3 Cacheando con <i>reverse proxies</i>	500
20.4 ESI	506
Sección 5 Apéndices	515

Apéndice A El motor de plantillas Twig	517
A.1 Sintaxis básica	517
A.2 Twig para maquetadores	518
A.3 Twig para programadores	524
A.4 Usando Twig en proyectos PHP propios	569
A.5 Usando Twig en Symfony2	573
Apéndice B Inyección de dependencias	581
B.1 Entendiendo la inyección de dependencias.	581
B.2 La inyección de dependencias en Symfony2.	589

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 se utiliza como documentación en los cursos presenciales de Symfony2 que imparte su autor. A lo largo de los próximos capítulos se explica paso a paso cómo desarrollar una aplicación web completa utilizando el framework Symfony2.

Los contenidos del libro empiezan desde cero y por tanto, no es necesario que tengas conocimientos previos sobre cómo programar con Symfony. No obstante, para ser un buen programador de Symfony2 es preciso que domines otras tecnologías importantes como Git, YAML y los *namespaces* de PHP 5.3. Si ya las conoces, puedes saltarte este primer capítulo. Si no, sigue leyendo porque esto no es un capítulo de relleno, es imprescindible para entender el resto del libro.

1.1 Cómo leer este libro

Si estás empezando con Symfony2, 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 Symfony2, puedes leer el libro en cualquier orden, empezando por ejemplo por los capítulos que más te interesen (Caché (página 487), internacionalización (página 283), mejorando el rendimiento (página 459), etc.)

1.2 Introducción a Symfony2

Symfony2 es la versión más reciente de Symfony, el popular framework para desarrollar aplicaciones PHP. Se anunció por primera vez a principios de 2009 (<http://www.symfony.es/2009/03/06/asi-seran-las-novedades-de-symfony-20/>) y supone un cambio radical tanto en arquitectura interna como en filosofía de trabajo respecto a sus versiones anteriores.

Symfony2 ha sido ideado para exprimir al límite todas las nuevas características de PHP 5.3 y por eso es uno de los frameworks PHP con mejor rendimiento. Su arquitectura interna está completamente desacoplada, lo que permite reemplazar o eliminar fácilmente aquellas partes que no encajan en tu proyecto.

Symfony2 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 Symfony2.

Symfony 2.3 se publicó en junio de 2013, casi dos años después que Symfony 2.0. Esta nueva versión mejora todos los aspectos de la versión original, al tiempo que mantiene una alta retrocompatibilidad, salvo en el caso de los formularios.

La gran novedad de Symfony 2.3 es que es **la primera versión de Symfony2 con un soporte largo** de tres años. Esto quiere decir que Symfony 2.3 contará con correcciones de errores hasta el mes de mayo de 2016.

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_3/)
- Las *recetas* o artículos breves (<http://symfony.com/doc/2.3/cookbook>)
- La documentación de su API (<http://api.symfony.com/2.3/index.html>)

Para estar al día de las novedades de Symfony2, 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 Introducción a PHP 5.3

De todas las novedades introducidas por PHP 5.3 (http://php.net/releases/5_3_0.php), las más relevantes para los programadores de Symfony2 son las funciones anónimas y los *namespaces*.

1.3.1 Funciones anónimas

Las funciones anónimas, también conocidas como *closures*, son funciones sin nombre que normalmente se utilizan para crear fácil y rápidamente un *callback*. El código fuente de Symfony2 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 de la aplicación. Todas las variables que necesite el código se debe 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));  
    };  
  
    // ...  
  
    $strlen($title);
```

1.3.2 Namespaces

Según la Wikipedia, un *namespace* 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 fuente de la aplicación. Todas las clases de Symfony2 utilizan los *namespaces* y por tanto, es imprescindible entenderlos bien antes de programar una aplicación Symfony2.

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 compartir su 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 `proyecto/src/Cupon/OfertaBundle/Entity/`. Si esta clase forma parte de una aplicación Symfony2, es obligatorio que incluya el siguiente *namespace* como primer contenido de la clase:

```
<?php  
namespace Cupon\OfertaBundle\Entity;  
...
```

En las aplicaciones Symfony2, el *namespace* coincide con la ruta del archivo a partir del directorio `src/` del proyecto, pero sin incluir el propio nombre del archivo. Las diferentes partes del *namespace* siempre se separan con la barra `\`.

Imagina ahora que dispones de otra clase llamada `Tienda.php` que se encuentra en el directorio `proyecto/src/Cupon/TiendaBundle/Entity/`. Para utilizar la clase `Oferta.php` dentro de la clase `Tienda.php` debes escribir lo siguiente:

```
<?php  
namespace Cupon\TiendaBundle\Entity;  
  
class Tienda {  
    $oferta = new \Cupon\OfertaBundle\Entity\Oferta();  
    // ...  
}
```

Como las clases ya no tienen nombres únicos, debes incluir toda la ruta de su *namespace* cuando quieras hacer referencia a una clase. Si utilizas muchas veces una clase, es mejor que importes primero su *namespace* con la instrucción `use`:

```
<?php  
namespace Cupon\TiendaBundle\Entity;  
  
use Cupon\OfertaBundle\Entity;  
  
class Tienda {  
    $oferta = new Entity\Oferta();  
    // ...  
}
```

La instrucción `use` indica el *namespace* de la clase o directorios de clases que están disponibles para usar dentro de otra clase. En el ejemplo anterior se importa un directorio entero (`Cupon\OfertaBundle\Entity`) y por eso el nombre de la clase se prefija con su valor (`new Entity\Oferta()`).

Cuando sólo utilizas una clase específica, es mejor importar la clase en vez de su directorio. Para ello indica en la instrucción `use` el *namespace* y el nombre de la clase (sin el sufijo `.php` o `.class.php`):

```
<?php  
namespace Cupon\TiendaBundle\Entity;  
  
use Cupon\OfertaBundle\Entity\Oferta;  
  
class Tienda {  
    $oferta = new Oferta();  
    // ...  
}
```

El código anterior muestra **la forma recomendada de utilizar el `use` en Symfony2**, ya que el código fuente resultante es más conciso. Por eso en la mayoría de clases de Symfony2 verás al principio muchas instrucciones `use` que importan las clases que se utilizan más adelante en el código.

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 Cupon\TiendaBundle\Entity;  
  
use Cupon\OfertaBundle\Entity\Oferta as Offer;  
  
class Tienda {  
    $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 Cupon\OfertaBundle\Utilidades\DateTime;  
// ...  
  
$fecha = new DateTime(); // Nuestra propia clase  
$fecha = new \DateTime(); // La clase nativa de PHP
```

1.3.3 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 5.5), las aplicaciones Symfony2 pueden hacer uso de ellas gracias a una librería desarrollada por el proyecto Doctrine2.

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 Symfony2:

```
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. El contenido del comentario no es trivial, ya que proporciona información clave sobre la propiedad. La instrucción `@ORM\Column(type="string")` define el tipo de columna de base de datos en la que se guarda la información de la propiedad. Y la instrucción `@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.

Symfony2 recomienda utilizar anotaciones en sus aplicaciones siempre que sea posible, pero no es obligatorio hacerlo. En los siguientes capítulos se emplean las anotaciones para configurar la validación de la información, pero no para el enrutamiento ni para la seguridad.

1.4 Introducción a YAML

Los archivos de configuración de Symfony2 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 tiene la ventaja de que el formato de la configuración 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 Symfony2.

1.4.1 Sintaxis

Los archivos de configuración de Symfony2 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 los arrays asociativos con las llaves `{ }`.
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 Symfony2 (`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, Symfony2 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')
        )
    )
)
)
)
)
```

1.5 Introducción a HTML5

Las páginas de la aplicación que se desarrolla en este libro están diseñadas con HTML5 y CSS3. Aunque este libro se centra solamente en la programación web, a continuación se introducen muy brevemente algunos de los nuevos elementos de HTML5. Observa el siguiente código HTML4:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">

<html lang="es">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Título de la página</title>
</head>
<body>
<div id="contenedor">
    <div id="menu">
        <ul> ... </ul>
    </div>

    <div id="contenido">
        <div id="principal"> ... </div>
        <div id="secundario"> ... </div>
    </div>

    <div id="pie"> ... </div>
</div>
</body>
</html>
```

Y ahora observa cómo se maqueta la misma página con HTML5:

```

<!DOCTYPE html>

<html lang="es">
<head>
    <meta charset="utf-8">
    <title>Título de la página</title>
</head>
<body>
```

```
<div id="contenedor">
  <header>
    <nav id="menu">
      <ul> ... </ul>
    </nav>
  </header>

  <article id="contenido">
    <section id="principal"> ... </section>
    <aside id="secundario"> ... </aside>
  </article>

  <footer> ... </footer>
</div>
</body>
</html>
```

HTML5 incluye muchos nuevos elementos para estructurar los contenidos de las páginas sin tener que recurrir siempre a la etiqueta `<div>`. El elemento `<header>` agrupa todos los contenidos de la cabecera de la página (logotipo, menú principal de navegación, buscador, etc.). El elemento `<nav>` se emplea para marcar la navegación del sitio web y suele encontrarse dentro de `<header>`. El elemento `<footer>` encierra todos los contenidos del pie de página.

Los contenidos interiores de cada página se estructuran con los elementos `<article>` y `<section>`. Una misma página puede contener tantos `<article>` como necesite, aunque resulta común utilizar solamente uno y estructurar su interior con `<section>`.

El último elemento importante es `<aside>`, que ha sido creado a imagen de las "*notas al margen*" de los libros impresos, ya que encierra el contenido relacionado con el contenido principal de la página. Esta relación es en ocasiones tan vaga, que se puede considerar como un contenido aparte.

1.6 Introducción a Git

Git es la herramienta de gestión de código más utilizada por los programadores profesionales. El propio código fuente de Symfony2 se gestiona mediante Git y las actualizaciones se realizan mediante comandos de Git. Así que aunque no utilices Git en tus aplicaciones, tendrás que instalarlo para poder utilizar Symfony2.

1.6.1 Instalación y configuración

Accede a la página oficial del proyecto Git en <http://git-scm.com> y descarga la versión más reciente que exista para tu sistema operativo. Después de instalarlo, ejecuta los siguientes comandos de consola para configurar el nombre y email que se incluye en los *commits*:

```
$ git config --global user.name <tu-nombre> <tus-apellidos>
$ git config --global user.email <tu-email>
```

Otra opción interesante es la que activa el coloreado de sintaxis para los mensajes que muestra Git en la consola: `git config --global color.ui true`. Si lo deseas, también puedes utilizar el si-

guiente comando para modificar el editor que te muestra Git para escribir los mensajes de *commit*:

```
git config --global core.editor <nombre-editor>
```

1.6.2 Comandos básicos

Descargar un proyecto alojado en Github para trabajar en tu propio equipo:

```
$ mkdir <directorio-local-proyecto>
$ git clone http://github.com/[...]/[...].git <directorio-local-proyecto>
```

Actualizar el proyecto local con los posibles cambios producidos en el repositorio remoto:

```
$ git pull
```

Realizar cambios en local y subirlos al repositorio remoto haciendo un *commit*:

```
# conocer qué archivos se han modificado localmente
$ git status

# añadir al repositorio un archivo que todavía no existe
$ git add <nombre-nuevo-archivo>

# añadir todos los archivos nuevos al repositorio
$ git add .

# marcar uno o más archivos como listos para subir
$ git commit <archivo1> <archivo2> ...

# marcar todos los archivos con cambios como listos para subir
$ git commit -a
# (se abre el editor por defecto para escribir el mensaje del commit)

# subir los cambios al repositorio remoto
$ git push
```

Crear una rama para programar una nueva funcionalidad, fusionarla después con la rama principal y subir los cambios al repositorio remoto:

```
# crear la nueva rama
$ git checkout -b <nombre-de-rama>

# ... realizar cambios en el código ...

# añadir los nuevos archivos creados
$ git add .

# commit de todos los cambios realizados
```

```
$ git commit -a

# volver a la rama principal
$ git checkout master

# fusionar los cambios de la nueva rama en la principal
$ git merge <nombre-de-rama>

# subir los cambios al repositorio remoto
$ git push origin master

# ver el historial de los últimos cambios producidos
$ git log
```

Git es una herramienta imprescindible para cualquier programador profesional. Aprovecha las siguientes referencias para mejorar tus conocimientos de Git:

- Pro Git (http://librosweb.es/pro_git/) , un libro gratuito sobre Git escrito por Scott Chacon y traducido a decenas de idiomas.
- Sección de ayuda de GitHub (<http://help.github.com/>) , el sitio web más popular para alojar repositorios Git.

Symfony2 utiliza Git cuando instala y/o actualiza sus componentes y librerías. No obstante, como se trata de comandos internos, el uso de Git en Symfony2 es transparente para el programador.

Por último, si utilizas Git para gestionar el código fuente de tu proyecto Symfony2, no olvides configurar bien el archivo `.gitignore` para no subir al repositorio de código archivos innecesarios. A continuación se muestra un ejemplo de archivo `.gitignore` para aplicaciones Symfony2:

```
# archivos que genera localmente Symfony2 para mejorar el rendimiento
app/bootstrap.php.cache

# directorio con todos los archivos de caché (muchísimos archivos)
app/cache/*

# directorio con los archivos de log (pocos archivos pero muy grandes)
app/logs/*

# directorio con todos los componentes y librerías de Symfony2
vendor/*

# directorios donde Symfony2 vuelca todos los archivos web públicos
web/bundles/*
web/css/*
web/js/*

# archivos creados por los IDEs Eclipse, Zend Studio, Netbeans, PHPStorm y
TextMate2
```

```
.buildpath  
.project  
.settings  
nbproject  
.idea  
.tm_properties
```

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 Symfony2 programando una aplicación web de ejemplo pero completa. La aplicación se llama *Cupon* y es un clon inspirado por *Groupon.com*^[1].

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: <http://github.com/javiereguiluz/Cupon/tree/2.3>. Resulta muy recomendable que instales la aplicación en tu ordenador siguiendo las instrucciones del repositorio. Así te será más fácil seguir 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.

[1] Groupon © es una marca registrada de Groupon, Inc. (<http://www.groupon.com/about>) , que también tiene reservados todos sus derechos. La mención y uso del nombre, marca o cualquier otro elemento de Groupon y de su sitio web se realizan únicamente con fines formativos y bajo el derecho de cita regulado en el artículo 32 del Texto Refundido de la Ley de Propiedad Intelectual (TRLPI) de la legislación española.

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 llenar 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 llenar 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 realista y completa posible, dadas las limitaciones de espacio de este libro resulta necesario definir varios límites prácticos:

- 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 as-

pecto 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

CUPON

[Oferta del día](#) [Ofertas recientes](#) [Mis Ofertas](#)

Barcelona

Oferta lorem ipsum dolor sit amet consectetur adipisicing

COMPRAR

Faltan: 04h 27m 49s **Compras:** 24 **Faltan 71 compras** para activar la oferta

Disfruta de la oferta en [Tienda Nulla scelerisque](#)
Calle Blandit, 83
08.080 Barcelona

Sobre la tienda
Sed malesuada, enim sit amet ultricies semper, elit leo lacinia massa, in tempus nisl ipsum quis libero. Maecenas non erat eu justo rutrum condimentum. Pellentesque ultricies erat ac lorem pharetra vulputate.

Condiciones: lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

00.00 € -00%

Regístrate
Accede a tu cuenta
Email
Contraseña Entrar
 No cerrar sesión

Ofertas en otras ciudades
Madrid: [Oferta at tincidunt enimsit natoque sitamet aut Zaragoza: \[Oferta facilisis pendisse sed imperdiet\]\(#\)](#)
Valencia: [Oferta ut lorem at libero dum aliquam](#)
Sevilla: [Oferta ipsum drerit ligula natoque enim](#)

Condiciones generales
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore.

© 201X - Cupon [Ayuda](#) [Contacto](#) [Privacidad](#) [Sobre nosotros](#) Español - English

DETALLE OFERTA

Figura 2.2 Wireframe de la página de detalle de una oferta

CUPON

[Oferta del día](#) [Ofertas recientes](#) [Mis Ofertas](#)

Sevilla

Ofertas recientes en Sevilla

Oferta lorem ipsum dolor sit amet consectetur adipisicing

COMPRAR

Faltan: 04h 27m 49s

Oferta dapibus lacinia lorem enimsit ipsum vulputae nulla

(... incluir 5 ofertas en el listado ...)

Finalizada el 11 de diciembre de 201X

Regístrate
Accede a tu cuenta
Email
Contraseña Entrar
 No cerrar sesión

Ofertas en otras ciudades

- [Barcelona](#)
- [Bilbao](#)
- [Madrid](#)
- [Valencia](#)
- [Zaragoza](#)

Sobre nosotros
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore.

OFERTAS RECIENTES

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

CUPON

[Oferta del día](#) [Ofertas recientes](#) [Mis Ofertas](#)

Sevilla ▾

Últimas ofertas que has comprado

Oferta lorem ipsum dolor sit amet consectetur adipisicing

- Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
- Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo.
- Duis aute irure dolor in reprehenderit in voluptate.
- Excepte sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt.

Comprada el 10 de diciembre de 201X

Oferta dapibus lacinia lorem enimsit ipsum vulputae nulla

- Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
- Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo.
- Duis aute irure dolor in reprehenderit in voluptate.

Comprada el 20 de noviembre de 201X

(... incluir 5 ofertas en el listado ...)

Disfruta de esta oferta en [Tienda Nulla scelerisque](#)

Disfruta de esta oferta en [Tienda Nec sapien](#)

Conectado como
José García Pérez
[Ver mi perfil](#)
[Cerrar sesión](#)

Ofertas en otras ciudades

- [Barcelona](#)
- [Bilbao](#)
- [Madrid](#)
- [Valencia](#)
- [Zaragoza](#)

Sobre nosotros

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore.

© 201X - Cupon [Ayuda](#) [Contacto](#) [Privacidad](#) [Sobre nosotros](#)

Español - English

MIS COMPRAS

Figura 2.4 Wireframe de la página de tus últimas compras

CUPON

[Oferta del día](#) [Ofertas recientes](#) [Mis Ofertas](#)

Valencia ▾

Tienda Aliquam dapibus

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Últimas ofertas publicadas

Fecha	Oferta	Precio	Descuento	Compras
11/12/201X	Oferta ut lorem at libero dum aliquam	65.16 €	24.11 €	15
10/12/201X	Oferta dum dapibus dum drerit sed sed	21.77 €	7.40 €	18
08/12/201X	Oferta malesuada sitamet libero enimsit	67.24 €	6.72 €	11
07/12/201X	Oferta vulputae ipsum malesuada vel ipsum ...	59.13 €	21.29 €	13
06/12/201X	Oferta ut aliquam enimsit natoque malesuada ...	29.67 €	14.84 €	17
05/12/201X	Oferta ipsum penatibus imperdiet vulputae at	9.75 €	2.83 €	11

Conectado como
José García Pérez
[Ver mi perfil](#)
[Cerrar sesión](#)

Ofertas en otras ciudades

- [Barcelona](#)
- [Bilbao](#)
- [Madrid](#)
- [Valencia](#)
- [Zaragoza](#)

Sobre nosotros

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore.

© 201X - Cupon [Ayuda](#) [Contacto](#) [Privacidad](#) [Sobre nosotros](#)

Español - English

TIENDA

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 básica 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 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	

Columna	Tipo	Comentarios
condiciones	text	
ruta_foto	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
fecha_publicacion	datetime	
fecha_expiracion	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
nombre	varchar(100)	
slug	varchar(100)	
login	varchar(10)	nombre de usuario para acceder a la extranet
password	varchar(255)	
salt	varchar(255)	valor aleatorio utilizado para codificar la contraseña
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 <i>login</i>
password	varchar(255)	
salt	varchar(255)	valor aleatorio utilizado para codificar la contraseña

Columna	Tipo	Comentarios
direccion	text	
permite_email	boolean	indica si el usuario permite comunicaciones publicitarias
fecha_alta	datetime	
fecha_nacimiento	datetime	
dni	varchar(9)	documento nacional de identidad
numero_tarjeta	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 Symfony2

Antes de empezar a programar la aplicación, es necesario adaptar todas las funcionalidades y *wireframes* a la filosofía de trabajo de Symfony2. Planificar bien el proyecto según *la forma de pensar* de Symfony2 te asegura que podrás desarrollarlo lo más eficientemente posible.

El orden recomendado para aplicar la filosofía de Symfony consiste en definir primero las entidades, después los *bundles* y por último el enrutamiento. Si estás empezando con Symfony2, es posible que no entiendas bien alguno de los conceptos explicados en las próximas secciones. No te preocunes demasiado, ya que conceptos como *bundle* son difíciles de explicar en abstracto, pero te resultarán intuitivos cuando hayas desarrollado alguna aplicación con Symfony2.

2.5 Entidades

El código PHP de las aplicaciones Symfony2 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 *traducir* la información de las tablas de la base de datos a clases PHP.

Aunque se trata de lo primero que debes hacer al planificar la aplicación, no podrás hacerlo hasta que no llegues al [capítulo 5](#) ([página 79](#)), donde se explica detalladamente el funcionamiento de las bases de datos en las aplicaciones Symfony2.

2.6 Bundles

Los *bundles* son la base de la nueva filosofía de trabajo de Symfony2. El código de tus aplicaciones y el propio código fuente de Symfony2 se estructura mediante *bundles*. Técnicamente, un *bundle* es un directorio que contiene todo tipo de archivos dentro una estructura jerarquizada de directorios.

Los *bundles* de las aplicaciones Symfony2 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* gigantesco. Lo hacen porque así creen que es más fácil reutilizar el *bundle* en otros proyectos. Otros programadores prefieren usar tantos *bundles* como *divisiones lógicas* tenga la aplicación. En el caso de la aplicación *Cupon* esto significa que habría que crear tres *bundles*, uno para el *frontend*, otro para la *extranet* y otro para el *backend*.

Para desarrollar la aplicación *Cupon* se ha tomado la decisión de utilizar cinco *bundles*, cuatro de ellos corresponden con las entidades principales de la aplicación y el último agrupa todo el *backend*. Esta solución es parecida a la estructura habitual de las aplicaciones symfony 1.x en el que se creaban dos aplicaciones (*frontend* y *backend*) y un módulo para cada una de las entidades. El nombre de los *bundles* siempre acaba con el sufijo *Bundle*, por lo que sus nombres serán:

- [CiudadBundle](#)
- [OfertaBundle](#)
- [TiendaBundle](#)
- [UsuarioBundle](#)
- [BackendBundle](#)

Con esta división, el código del *frontend* se distribuye entre varios *bundles*, el del *backend* se define en un único *bundle* y el código de la *extranet* se incluye dentro del *bundle* [TiendaBundle](#), ya que se trata de una zona restringida relacionada con las tiendas.

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*. Y en el peor de los casos, utilizar demasiados *bundles* sólo implica que tendrás que escribir un poco más de código en tu aplicación.

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 Symfony2 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	<code>/{{ciudad}}</code>
oferta	<code>/{{ciudad}}/ofertas/{{slug}}</code>

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

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	<code>/extranet</code>
extranet_oferta_nueva	<code>/extranet/oferta/nueva</code>
extranet_oferta_editar	<code>/extranet/oferta/editar/{id}</code>
extranet_oferta_ventas	<code>/extranet/oferta/ventas/{slug}</code>
extranet_perfil	<code>/extranet/perfil</code>
extranet_login	<code>/extranet/login</code>
extranet_login_check	<code>/extranet/login_check</code>
extranet_logout	<code>/extranet/logout</code>

Backend

Nombre	URL
backend_portada	<code>/backend</code>
backend_ciudad_cambiar	<code>/backend/cambiar-a-{ciudad}</code>
backend_ciudad	<code>/backend/ciudad/</code>

Nombre	URL
backend_ciudad_show	/backend/ciudad/{id}/show
backend_ciudad_new	/backend/ciudad/new
backend_ciudad_create	/backend/ciudad/create
backend_ciudad_edit	/backend/ciudad/{id}/edit
backend_ciudad_update	/backend/ciudad/{id}/update
backend_ciudad_delete	/backend/ciudad/{id}/delete

Las siete últimas rutas de la tabla anterior se generan automáticamente con una utilidad de Symfony2 que se explicará más adelante. Igualmente, existen otros tres grupos de siete rutas cada uno para gestionar las ofertas (`backend_oferta_*`), las tiendas (`backend_tienda_*`) y los usuarios (`backend_usuario_*`).

CAPÍTULO 3

Instalando y configurando Symfony2

Symfony2 es una aplicación PHP desarrollada a partir de varios elementos individuales llamados componentes y *bundles*, además de diversas librerías desarrolladas por terceros. Su código fuente se encuentra en el **repositorio oficial** del proyecto en GitHub: <http://github.com/symfony/symfony>

Este repositorio es útil para seguir la actividad y el desarrollo del framework, pero no se aconseja su uso para instalar Symfony2. Los creadores de Symfony recomiendan utilizar alguna de sus **distribuciones**. Siguiendo la misma idea del mundo Linux, Symfony2 dispone de varias distribuciones diferentes que se adaptan a cada situación o tipo de programador.

Una distribución puede incluir, además del código fuente de Symfony2, varios paquetes o *bundles* útiles, una estructura de directorios ya creada para que organices bien el código de tu aplicación y varios archivos de configuración adecuados para la mayoría de aplicaciones web.

Hacer uso de una distribución te permite empezar a desarrollar tu aplicación Symfony2 rápidamente y con facilidad. Si utilizaras directamente el código fuente del repositorio de Symfony2, antes de crear tu aplicación deberías realizar muchas configuraciones e instalaciones que además no están documentadas.

La principal distribución de Symfony2 se llama **distribución estándar** (<http://github.com/symfony/symfony-standard>) y además es la única recomendada oficialmente. Si tienes experiencia en el mundo Symfony, esta distribución es parecida al *sandbox* de symfony 1. La distribución estándar incluye un proyecto Symfony2 ya creado, un instalador/configurador vía web y un archivo de configuración simple en formato [*.yml](#).

La otra distribución disponible se llama **hello world** (<http://github.com/symfony/symfony-hello-world>) y solamente incluye lo mínimo imprescindible para crear una página que muestre el texto *hola mundo*. Esta distribución se ha creado para obtener el mejor rendimiento posible en los *benchmarks* que prueban los frameworks web mostrando el texto *hola mundo*, olvidándose de para qué sirve realmente un framework.

Symfony2 todavía no ha aprovechado completamente las posibilidades que ofrece el concepto de distribución. No obstante, en un futuro no muy lejano podrían existir una distribución orientada a proyectos que usan Propel, otra para proyectos Doctrine, otra para aplicaciones que integran Drupal, etc.

Para desarrollar la aplicación *Cupon* de este libro se utiliza la distribución estándar, ya que es la única opción recomendada oficialmente por los creadores de Symfony2.

3.1 Instalación

Actualmente existen tres formas diferentes de instalar Symfony 2.3:

1. Instalación mediante **Composer**, que es la forma recomendada oficialmente.
2. Instalación mediante un **archivo comprimido con vendors**, sólo se recomienda su uso a los programadores principiantes o a los que quieren probar Symfony2 lo más rápidamente posible antes de decidir si utilizan Symfony2 o no para su proyecto.
3. Instalación mediante un **archivo comprimido sin vendors**, es una forma ligeramente diferente de hacer lo mismo que en la instalación mediante Composer.

Composer, que es una herramienta para gestionar las dependencias en proyectos PHP, es bastante útil en la segunda forma de instalación y totalmente obligatorio en las demás. Así que antes de instalar Symfony 2.3, debes dedicar unos minutos a instalar Composer.

3.1.1 Instalando Composer

Symfony2 es un proyecto tan grande que sus creadores han decidido reutilizar varias herramientas y librerías ya existentes, en vez de programar a medida todos sus componentes. De hecho, cada vez que envías un email, creas una plantilla o escribes un mensaje de log en Symfony2, en realidad estás ejecutando el código de alguna librería ajena al proyecto Symfony.

Esta decisión tiene unas implicaciones técnicas muy importantes para Symfony2. En primer lugar, debe comprobar que todas las librerías externas estén correctamente instaladas. Después, debe comprobar que las librerías sean compatibles con la versión de Symfony2 que utilices y que también sean compatibles entre sí. Teniendo en cuenta que a su vez cada librería puede depender de otras librerías, la gestión de estas dependencias se puede convertir rápidamente en una pesadilla.

Symfony 2.0 utilizaba un sistema propio para gestionar sus dependencias. Este sistema se basaba en dos archivos de configuración (`deps` y `deps.lock`) y aunque funcionaba bien, era bastante incómodo para los programadores. El único motivo por el que Symfony 2.0 utilizaba su propio sistema es que Composer no se pudo terminar a tiempo para la publicación de Symfony 2.0.

Composer es la solución a todos los problemas de dependencias de los proyectos PHP. Gracias a Composer, tú sólo tienes que indicar qué librerías necesita el proyecto y Composer se encarga de descargarlas e instalarlas, teniendo en cuenta qué versiones son compatibles con tu proyecto. Además, no debes preocuparte de si una librería requiere de otras para funcionar, ya que Composer realiza esta comprobación e instala recursivamente cualquier otra librería necesaria.

3.1.1.1 Instalando Composer en Windows

La forma más sencilla de instalar Composer en un sistema Windows es utilizar el instalador `Composer-Setup.exe` que puedes descargar al acceder a <http://getcomposer.org/download/>. Si por cualquier motivo no puedes utilizar el instalador, sigue leyendo esta sección.

Las siguientes instrucciones explican cómo instalar Composer de forma global en tu ordenador. Esto supone que la instalación es un poco más larga de lo normal, pero a cambio no tendrás que reinstalar Composer para cada nuevo proyecto.

1. Abre cualquier navegador y accede a la siguiente dirección <http://getcomposer.org/installer>
2. Guarda el contenido de la página anterior en un archivo llamado `instalador.php` (no importa el directorio donde lo guardes).
3. Abre la consola de comandos de Windows y entra en el directorio donde has guardado el archivo `instalador.php`. Por ejemplo:

```
C:\> cd "C:\Document and settings\Javier\Desktop"
```

4. Ejecuta el siguiente comando para instalar Composer:

```
C:\Document and settings\Javier\Desktop> php instalador.php
```

Si no te funciona el comando anterior, seguramente la razón es que no puedes ejecutar programas de PHP directamente desde la consola. Para solucionarlo, consulta la documentación de tu instalación de PHP (XAMPP, Zend Server, etc.).

Si todo ha funcionado bien, verás un nuevo archivo `composer.phar` en ese mismo directorio. Ahora ya puedes borrar el archivo `instalador.php`.

5. Localiza el directorio donde tienes instalado el archivo `php.exe`. Si utilizas XAMPP por ejemplo, este archivo seguramente se encuentra en <C:\xampp\php\php.exe>
6. Mueve el archivo `composer.phar` anterior al mismo directorio donde se encuentra `php.exe`.
7. Por último, crea un archivo llamado `composer.bat` en el mismo directorio de `php.exe` y `composer.phar` y que contenga las dos siguientes líneas:

```
@ECHO OFF  
php composer.phar %*
```

Después de todos estos pasos, abre una nueva consola de comandos y ejecuta el siguiente comando sin ninguna opción:

```
C:\> composer
```

Si todo ha funcionado bien, deberías ver un listado con los comandos de Composer:

```
Composer version 8d7e5cd  
  
Usage:  
  [options] command [arguments]
```

```
Options:
--help           -h Display this help message.
--quiet          -q Do not output any message.
--verbose         -v Increase verbosity of messages.
...
Available commands:
about            Short information about Composer
create-project   Create new project from a package into given directory.
depends          Shows which packages depend on the given package
dump-autoload    dumps the autoloader
dumpautoload     dumps the autoloader
help             Displays help for a command
...
```

3.1.1.2 Instalando Composer en Linux y Mac OS X

Las siguientes instrucciones explican cómo instalar Composer de forma global en tu ordenador. Esto supone que la instalación es un poco más larga de lo normal, pero a cambio no tendrás que reinstalar Composer para cada nuevo proyecto.

1. Abre la consola de comandos y ejecuta el siguiente comando en cualquier directorio:

```
$ curl -s https://getcomposer.org/installer | php
```

Si todo ha funcionado bien, verás un nuevo archivo `composer.phar` en ese mismo directorio.

2. Mueve el archivo `composer.phar` a algún directorio ejecutable del sistema, como por ejemplo:

```
$ sudo mv composer.phar /usr/local/bin/composer
```

Abre una nueva consola de comandos y ejecuta el siguiente comando sin ninguna opción:

```
$ composer
```

Si todo ha funcionado bien, deberías ver un listado con los comandos de Composer:

```
Composer version 8d7e5cd

Usage:
[options] command [arguments]

Options:
--help           -h Display this help message.
--quiet          -q Do not output any message.
--verbose         -v Increase verbosity of messages.
```

```
...  
  
Available commands:  
about           Short information about Composer  
create-project Create new project from a package into given directory.  
depends         Shows which packages depend on the given package  
dump-autoload  dumps the autoloader  
dumpautoload   dumps the autoloader  
help            Displays help for a command  
...  

```

3.1.2 Instalando Symfony 2.3 con Composer

En primer lugar, elige el directorio donde vas a crear tu proyecto Symfony2, pero no crees ese directorio. A continuación, abre la consola de comandos de tu sistema operativo y ejecuta lo siguiente (cambiando el valor <directorío> por la ruta del directorio donde quieras crear el proyecto Symfony2 y [2.3.3](#) por la versión específica de Symfony 2.3 que quieras instalar):

```
composer create-project symfony/framework-standard-edition <directorío> 2.3.3  
  
// ejemplo en Windows  
C:\> composer create-project symfony/framework-standard-edition "D:\Proyectos\Cupon" 2.3.3  
  
// ejemplo en Linux y Mac OS X  
$ composer create-project symfony/framework-standard-edition "/Proyectos/Cupon" 2.3.3  
  
// ejemplo en Linux y Mac OS X si se producen problemas de permisos  
$ sudo composer create-project symfony/framework-standard-edition "/Proyectos/Cupon" 2.3.3
```

Como resultado de la ejecución de ese comando verás una larga lista de paquetes que se descargan e instalan en el directorio que has indicado:

```
Installing symfony/framework-standard-edition (v2.3.3)  
- Installing symfony/framework-standard-edition (v2.3.3)  
  
Created project in /Proyectos/Cupon  
Loading composer repositories with package information  
Installing dependencies (including require-dev)  
- Installing jdorn/sql-formatter (v1.2.9)  
- Installing doctrine/common (2.3.0)  
- Installing doctrine/dbal (2.3.4)  
- Installing psr/log (1.0.0)  
- Installing twig/twig (v1.13.2)  
- ...
```

Como Symfony2 incluye decenas de dependencias, este comando puede tardar varios minutos en terminar su ejecución, dependiendo de la velocidad de tu conexión a Internet.

TRUCO Si en vez de una versión específica de Symfony 2.3 quieres instalar la versión más reciente que exista en ese momento en el repositorio, utiliza como versión el valor `2.3.x-dev`. Utiliza esta versión solamente en desarrollo, ya que la versión descargada es inestable y no se recomienda para el servidor de producción.

Al terminar la instalación de Symfony 2.3, el instalador te pedirá que escribas el valor de varias opciones de configuración para tu proyecto. Como todavía es pronto para explicar estas opciones, pulsa la tecla `<Enter>` para asignar a cada opción el valor por defecto que se muestra entre paréntesis:

```
Creating the "app/config/parameters.yml" file.  
Some parameters are missing. Please provide them.  
database_driver (pdo_mysql): <ENTER>  
database_host (127.0.0.1): <ENTER>  
database_port (null): <ENTER>  
database_name (symfony): <ENTER>  
database_user (root): <ENTER>  
database_password (null): <ENTER>  
mailer_transport (smtp): <ENTER>  
mailer_host (127.0.0.1): <ENTER>  
mailer_user (null): <ENTER>  
mailer_password (null): <ENTER>  
locale (en): <ENTER>  
secret (ThisTokenIsNotSoSecretChangeIt): <ENTER>
```

3.1.3 Instalando Symfony 2.3 mediante un archivo con vendors

Se denomina *vendors* o dependencias al conjunto de librerías y utilidades externas que Symfony2 requiere para su funcionamiento. La versión *con vendors* de Symfony2 es un archivo comprimido de más de 8 MB que incluye el código fuente de todas esas librerías externas.

La gran ventaja de esta versión es que descomprimes un archivo y ya puedes empezar a trabajar en tu aplicación, sin tener que instalar nada más. La desventaja es que la forma de instalar Symfony 2.3 que menos flexibilidad ofrece para instalar otras librerías y componentes de terceros. Tanto los creadores de Symfony2, como el autor de este libro, te recomendamos evitar esta versión **con vendors** siempre que sea posible.

1. Accede a <http://symfony.com/download> y descarga la versión más reciente disponible cuyo nombre **no** contenga las palabras *without vendors*.

2. Copia el archivo descargado (`Symfony_Standard_Vendors_2.3.XX.zip` o `Symfony_Standard_Vendors_2.3.XX.tgz`) dentro del directorio raíz de tu servidor web.

3. Descomprime el archivo directamente en el directorio raíz del servidor. Se creará un directorio llamado [Symfony/](#) con la siguiente estructura de contenidos:

```
htdocs/
└── Symfony_Standard_Vendors_2.3.XX.zip
└── Symfony/
    ├── composer.json
    ├── composer.lock
    ├── LICENSE
    ├── README.md
    ├── UPGRADE.md
    ├── app/
    ├── bin/
    ├── src/
    ├── vendor/
    └── web/
        ├── ...
        ├── app.php
        └── app_dev.php
```

Por último, modifica el nombre de la carpeta descomprimida sustituyéndolo por el nombre de tu proyecto ([Cupon](#) en este caso).

3.1.4 Instalando Symfony 2.3 mediante un archivo sin vendors

La versión *sin vendors* es un archivo de apenas 200 KB, ya que no incluye ni el código fuente de Symfony2 ni el de ninguna librería externa. En su lugar, sólo contiene las instrucciones necesarias para descargar todos los paquetes con Composer.

Esta versión puede parecer peor que la versión *con vendors* porque requiere un paso intermedio para descargar el código antes de poder usar Symfony2. No obstante, esta forma de trabajar ofrece mucha más flexibilidad al utilizar librerías y utilidades creadas por terceros.

1. Accede a <http://symfony.com/download> y descarga la versión *without vendors* más reciente disponible.
2. Copia el archivo descargado ([Symfony_Standard_2.3.XX.tgz](#) o [Symfony_Standard_2.3.XX.zip](#)) dentro del directorio raíz de tu servidor web.
3. Descomprime el archivo directamente en el directorio raíz del servidor. Se habrá creado un directorio llamado [Symfony/](#).
4. Abre una consola de comandos y accede al directorio [Symfony/](#) creado en el paso anterior.
5. Ejecuta el siguiente comando para descargar el código fuente de Symfony2 y el de todas las librerías externas: `composer install` (para que te funcione este comando, debes instalar Composer globalmente en tu ordenador, tal y como se ha explicado anteriormente (página 46)).

Una vez terminada la descarga de los *vendors*, deberías ver la misma estructura de archivos y directorios que en el caso de la distribución *con vendors*:

```
htdocs/
└── Symfony_Standard_2.3.XX.zip
    └── Symfony/
        ├── composer.json
        ├── composer.lock
        ├── LICENSE
        ├── README.md
        ├── UPGRADE.md
        ├── app/
        ├── src/
        ├── vendor/
        └── web/
            ├── ...
            ├── app.php
            └── app_dev.php
```

3.1.5 Comprobando la instalación realizada

Independientemente de la forma que hayas utilizado para instalar Symfony2, antes de empezar a programar tu aplicación debes asegurarte de que tu ordenador cumple con los requisitos para que Symfony2 funcione bien. Para ello, entra en el directorio del proyecto ([Cupon/](#) en este ejemplo) y ejecuta el siguiente comando:

```
$ php app/check.php
```

El script `check.php` muestra por consola una lista de requisitos obligatorios (*Mandatory requirements*) y otra lista de requisitos deseables (*Optional checks*) para ejecutar Symfony2. 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 siguiente comando para comprobar que Symfony2 se ha instalado correctamente: `php app/console`. Si funciona bien, tienes que ver en tu consola algo como lo siguiente:

```
$ php app/console
Symfony version 2.3.3 - app/dev/debug

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose         -v|vv|vvv Increase the verbosity of messages:
```

```

        1 for normal output, 2 for more verbose output
        and 3 for debug
--version      -V Display this application version.
--ansi         Force ANSI output.
--no-ansi       Disable ANSI output.
--no-interaction -n Do not ask any interactive question.
--shell         -s Launch the shell.
--process-isolation Launch commands from shell as a separate process.
--env           -e The Environment name.
--no-debug      Switches off debug mode.

Available commands:
  help          Displays help for a command
  list          Lists commands
  assetic
    assetic:dump   Dumps all assets to the filesystem
  assets
    assets:install  Installs bundles web assets under a public web directory
  cache
    cache:clear     Clears the cache
    cache:warmup    Warms up an empty cache
  ...

```

Resulta imprescindible que el comando `php app/console` se ejecute correctamente, ya que Symfony2 utiliza la línea de comandos para automatizar muchas tareas importantes. Todos los comandos de Symfony2 se ejecutan mediante `php app/console <nombre-del-comando>`

Si no quieres escribir una y otra vez `php app/console` delante del comando a ejecutar, puedes entrar en la *shell* de Symfony2 ejecutando `php app/console --shell`. Ahora ya puedes ejecutar cualquier comando escribiendo simplemente su nombre.

3.1.6 Instalando la aplicación Cupon

Si vas a desarrollar desde cero la aplicación *Cupon* a medida que leas el libro, debes instalar Symfony2 tal y como se ha explicado en las secciones anteriores (no importa cuál de los tres métodos de instalación elijas). Pero si prefieres probar la aplicación y ver su código completo a medida que leas el libro, también puedes instalar la aplicación *Cupon* terminada. 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/javierguiluz/Cupon.git cupon`)
3. Accede al directorio de la aplicación (`cd cupon`)
4. Selecciona la rama correspondiente a Symfony 2.3 (`git checkout 2.3`)
5. Descarga las dependencias de Symfony (`composer install`)
6. Configura tu servidor web para que se pueda acceder a la aplicación *Cupon* desde el directorio en el que se ha instalado.

Para poder probar bien la aplicación, es necesario cargarla con datos de prueba, tal como se explicará detalladamente en los próximos capítulos. Por el momento, ejecuta los siguientes comandos desde dentro del directorio de la aplicación:

1. Crea una nueva base de datos para la aplicación y configura sus datos de acceso (usuario, contraseña, *host*) en el archivo de configuración `app/config/parameters.yml`
2. Crea el esquema de la base de datos (`php app/console doctrine:schema:create`)
3. Carga los datos de prueba (`php app/console doctrine:fixtures:load`)
4. Genera los *web assets* con Assetic (`php app/console assetic:dump --env=prod --no-debug`). Si se produce un error al ejecutar este comando, debes activar la cache APC en tu servidor web.
5. Asegúrate de que el directorio `web/uploads/images/` tiene permisos de escritura.

3.2 Configurando el entorno de ejecución

Una buena práctica para desarrollar y probar aplicaciones web en local consiste en configurar un entorno de ejecución lo más parecido posible al entorno de producción en el que se ejecutará la aplicación real. Para ello, primero se define un dominio ficticio para el sitio web local. Si el proyecto real fuera a alojarse en `cupon.com`, el dominio local podría ser `cupon.dev` o `cupon.local`.

Configura el dominio ficticio `cupon.local` añadiendo las siguientes líneas en el archivo `/etc/hosts` (en Linux y Mac OS X) o `\windows\system32\drivers\etc\hosts` (en Windows):

```
127.0.0.1 cupon.local
# esta segunda línea es opcional, pero configurar la
# dirección local en formato IPv6 puede mejorar el
# rendimiento al probar la aplicación en local
::1 cupon.local
```

A continuación, configura en tu servidor web un *host virtual* asociado a este dominio ficticio para poder ejecutar el proyecto en local. Si utilizas el servidor Apache, basta con añadir lo siguiente:

```
<VirtualHost *:80>
    DocumentRoot "/Proyectos/Cupon/web"
    DirectoryIndex app.php
    ServerName cupon.local

    <Directory "/Proyectos/Cupon/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

NOTA Si utilizas la versión 2.4 o superior de Apache, reemplaza la directiva `Allow from All` por `Require all granted` para evitar errores de tipo `403` con el mensaje "*Client denied by server configuration*".

No olvides cambiar `/Proyectos/Cupon/web` por la ruta del directorio `web/` de tu proyecto (por ejemplo, `/var/www/cupon.com/web` o `C:\XAMPP\htdocs\Cupon\web`). Guarda los cambios, reinicia el servidor web y accede a la URL `http://cupon.local/app_dev.php`. Deberías ver la siguiente página:

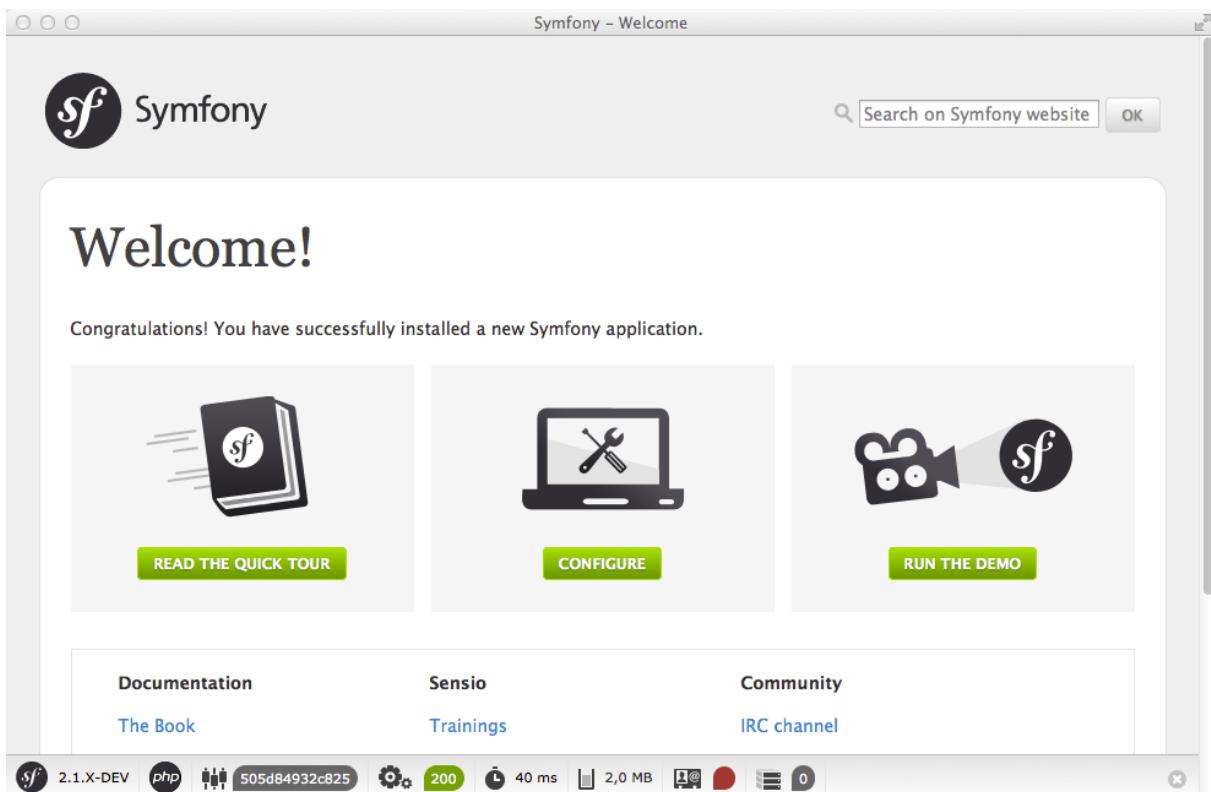


Figura 3.1 Portada por defecto de Symfony2

NOTA Si en vez de la portada de Symfony2 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 72) se explica con detalle la causa de este error y cómo solucionarlo.

La configuración del servidor web también puede servirte para cumplir con los requisitos de instalación de Symfony2. Si utilizas Apache y por cualquier circunstancia no puedes editar el archivo de configuración `php.ini`, añade las siguientes opciones para desactivar las etiquetas cortas y para establecer la zona horaria (que son dos de los requisitos de Symfony2):

```
<VirtualHost *:80>
    DocumentRoot    "/proyectos/cupon/web"
    DirectoryIndex app.php
    ServerName      cupon.local

    <Directory "/proyectos/cupon/web">
        AllowOverride All
        Allow from All
    </Directory>

    php_flag short_open_tag off
```

```
    php_value date.timezone "Europe/Madrid"
</VirtualHost>
```

NOTA A partir de ahora, todos los ejemplos de este libro utilizan el dominio <http://cupon.local> en sus URL. Debes tenerlo en cuenta sobre todo si has decidido utilizar otra estrategia para probar la aplicación en local.

El siguiente paso consiste en configurar la aplicación con ayuda del configurador web de Symfony2. Accede a la URL <http://cupon.local/config.php> y verás la página de bienvenida del configurador:

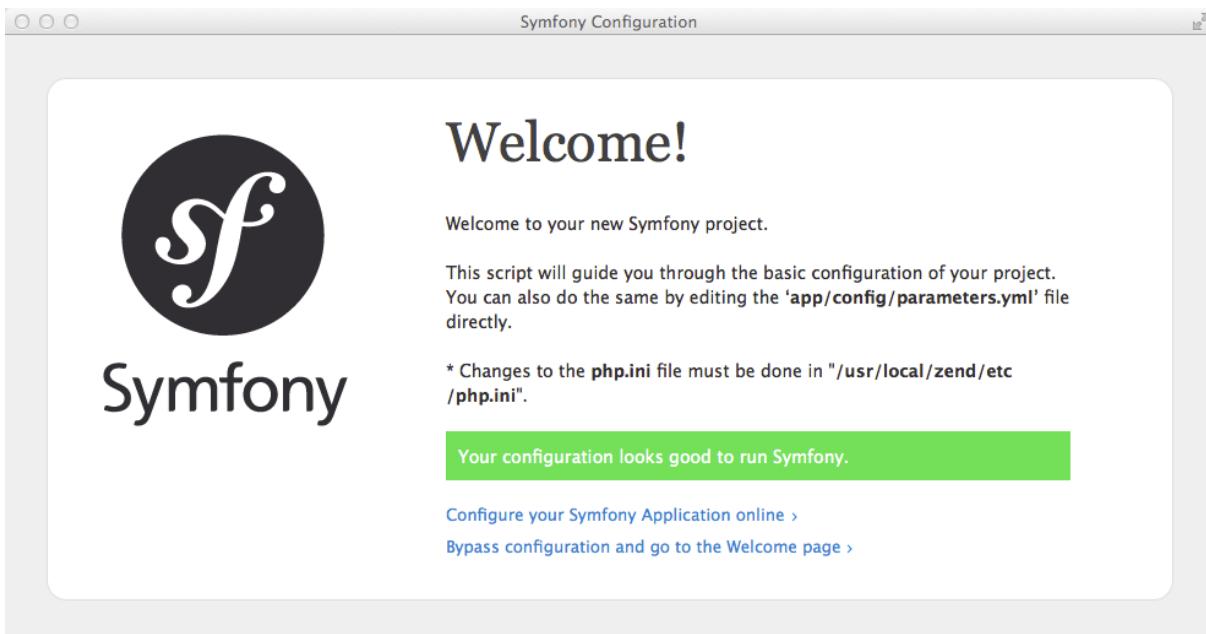


Figura 3.2 Portada del configurador web de Symfony2

Si tu ordenador no cumple con todos los requisitos técnicos para ejecutar Symfony2, en lugar de la portada normal verás la siguiente portada que muestra la lista de errores que debes corregir para poder continuar:

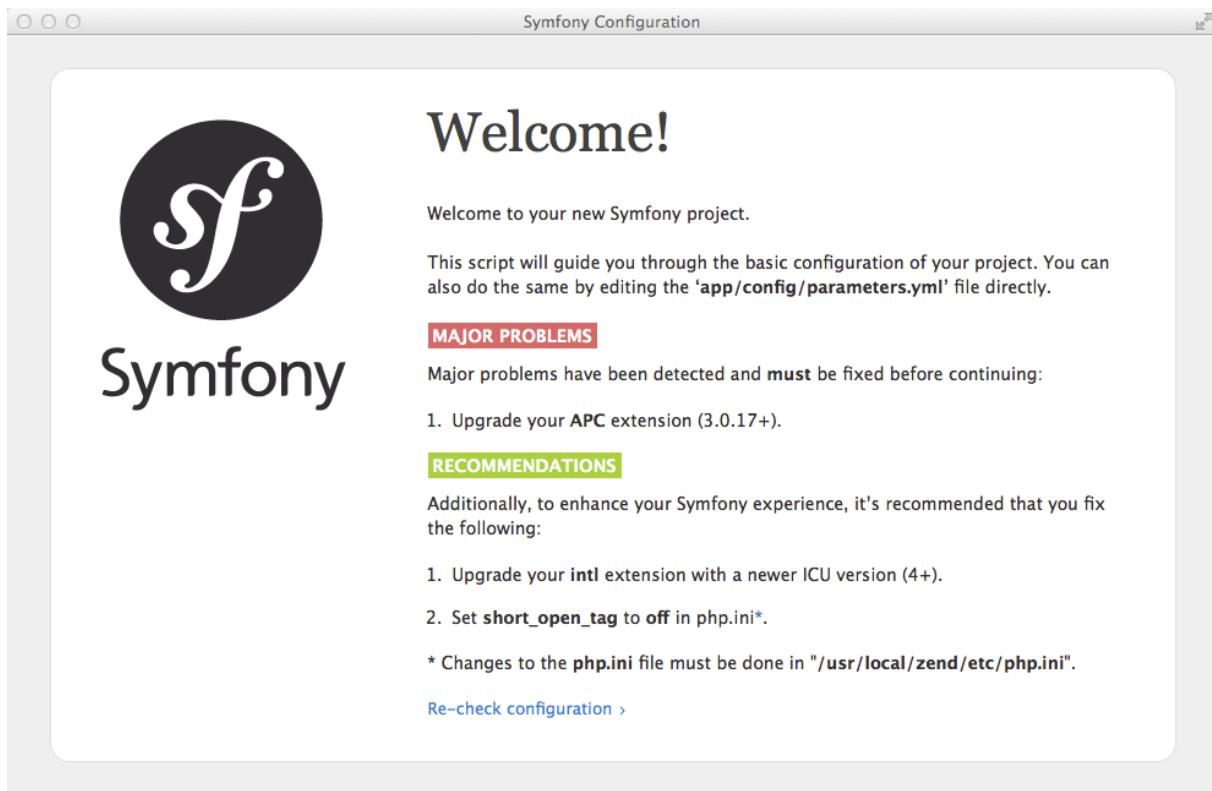


Figura 3.3 Portada del configurador web de Symfony2 con errores

Una vez solucionados todos los problemas, pincha el primer enlace *Configure your Symfony Application online* para acceder al primer paso de la configuración:

The screenshot shows the 'Configure your Database' step of the Symfony configuration wizard. It includes fields for Driver (MySQL PDO), User (root), Host (localhost), Name (symfony), Password, Password again, Path, and Port. A 'NEXT STEP' button is at the bottom.

STEP 1 > STEP 2

Configure your Database

If your website needs a database connection, please configure it here.

Driver *

User

Host

Password

Name

Password again

Path

Port

NEXT STEP

* mandatory fields

System status bar: 2.1.X-DEV, php, 505d89f355c96, 200, 54 ms, 3,0 MB, 0 errors.

Figura 3.4 Primer paso del configurador web de Symfony2

En esta página se configuran los datos de acceso a la base de datos, como el servidor, el nombre de base de datos, el usuario y la contraseña. Rellena los datos necesarios y pulsa el botón *Next Step*.

The screenshot shows the 'Global Secret' step of the Symfony configuration wizard. It includes a 'Secret' field containing '06c7ff1eb1ea025a479f2b1b3f' with a 'GENERATE' button, and a 'NEXT STEP' button.

STEP 1 > STEP 2

Global Secret

Configure the global secret for your website (the secret is used for the CSRF protection among other things):

Secret * **GENERATE**

NEXT STEP

* mandatory fields

System status bar: 2.1.X-DEV, php, 505d8cc1167f, 200, 43 ms, 2,5 MB, 0 errors.

Figura 3.5 Segundo paso del configurador web de Symfony2

La segunda página permite configurar el "*secreto global*", que es una cadena de texto aleatoria utilizada en diversas partes del proyecto, como por ejemplo la protección de los formularios frente a

los ataques de tipo CSRF. Pulsa varias veces el botón *GENERATE* para asegurarte de que sea un valor realmente aleatorio.

Pulsa el botón *NEXT STEP* una vez más para terminar la configuración. En este punto es posible que veas el siguiente error:

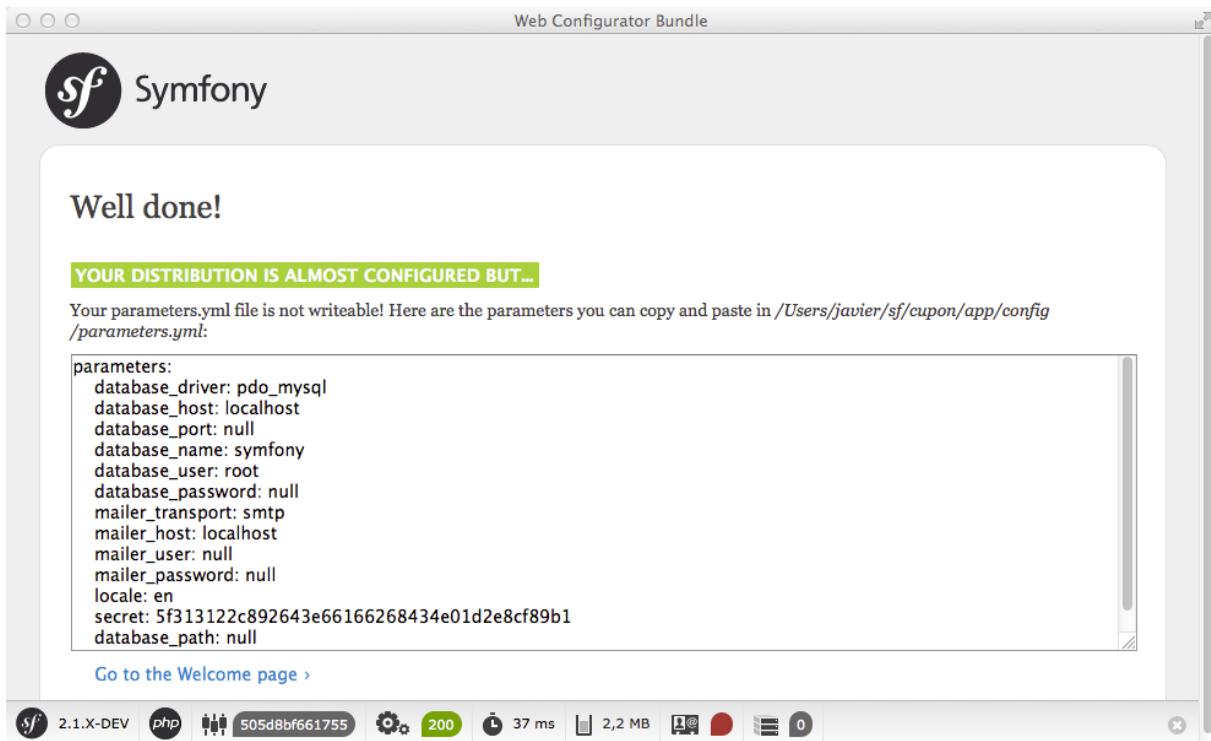


Figura 3.6 Error de permisos en el configurador web de Symfony2

El configurador web trata de guardar todos los datos introducidos en un archivo llamado [app/config/parameters.yml](#). Si el servidor web no tiene los permisos adecuados, no podrá escribir en ese archivo. Cambia los permisos del archivo y repite el último paso o copia directamente la información mostrada por el configurador y pégala en el archivo [app/config/parameters.yml](#).

El configurador web solamente permite configurar la base de datos y el *secreto global*. En las demás opciones de configuración Symfony2 establece por defecto los siguientes valores:

- Twig es el único motor de plantillas activado.
- Doctrine ya está activado tanto en su ORM como en su DBAL (acceso a base de datos).
- Swiftmailer es la librería activada para enviar emails.
- Las anotaciones están activadas para todos los casos (rutas, validación, entidades, etc.)

La distribución estándar de Symfony2 también incluye un pequeño *bundle* de ejemplo llamado [AcmeDemoBundle](#), que sirve para mostrar algunas de las buenas prácticas en el desarrollo con Symfony2. Después de probarlo y *jugar* con su código, se recomienda borrarlo. Para eliminar este *bundle* y todos sus *rastros*:

- Elimina el directorio [src/Acme](#)

- Quita las rutas `_welcome`, `_demo` y `_demo_secured` del archivo `app/config/routing_dev.yml`
- Borra el registro del bundle en el archivo `app/AppKernel.php` (quita la línea `$bundles[] = new Acme\DemoBundle\AcmeDemoBundle();`)

3.3 Actualizando Symfony2

Antes de actualizar la versión de Symfony 2.3 que utiliza tu aplicación, asegúrate de que la nueva versión de Symfony no contiene algún cambio que pudiera afectarte negativamente. Para ello, consulta siempre el documento CHANGELOG (<https://github.com/symfony/symfony/blob/2.3/CHANGELOG-2.3.md>) del repositorio de Symfony 2.3 que explica todos los cambios de cada versión.

Gracias al uso de Composer, actualizar la versión de Symfony 2.3 es tan sencillo como entrar con la consola de comandos en el directorio raíz de tu proyecto y ejecutar el siguiente comando:

```
$ composer update
```

Para que te funcione este comando, debes instalar Composer globalmente en tu ordenador, tal y como se ha explicado anteriormente (página 46).

3.4 Creando los *bundles*

Una vez instalado Symfony2, entra en el directorio del proyecto y observa su jerarquía de directorios. Esta jerarquía es común a todos los proyectos Symfony2 y está compuesta por:

Directorio	Propósito
<code>app/</code>	Contiene los archivos de configuración, la caché, los logs y los recursos globales.
<code>app/config/</code>	Guarda todos los archivos de configuración de la aplicación.
<code>app/cache/</code>	Contiene todos los archivos generados por las numerosas cachés de Symfony2 (clases, enrutamiento, plantillas, entidades, validación, etc.). Junto con el directorio <code>app/logs/</code> es el único en el que Symfony2 debe tener permisos de escritura.
<code>app/logs/</code>	Contiene los archivos de log generados por la aplicación en cualquier entorno de ejecución (desarrollo, producción, tests, etc.). Junto con el directorio <code>app/cache/</code> es el único en el que Symfony2 debe tener permisos de escritura.
<code>app/Resources/</code>	Almacena los recursos que se utilizan globalmente en el proyecto (como por ejemplo el <i>layout</i> de las plantillas) o recursos muy especiales que no encajan en ningún otro sitio (como por ejemplo una librería Java para comprimir archivos CSS y JavaScript)
<code>src/</code>	Guarda todo el código fuente propio del proyecto. Aquí es donde creas los <i>bundles</i> de tu aplicación.
<code>vendor/</code>	Contiene todo el código fuente de Symfony2 y de todas las librerías externas.

Directorio	Propósito
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)

3.4.1 El primer *bundle*

Ahora que ya sabes que todo tu código se guarda en el directorio [src/](#), puedes empezar a crear los *bundles*. Dentro de un *bundle* puedes utilizar cualquier estructura de directorios, pero Symfony2 espera que utilices una estructura muy específica para simplificar su trabajo.

Afortunadamente existe un comando de consola que genera el *esqueleto* completo de un *bundle* con sólo contestar a varias preguntas básicas. Abre una consola de comandos, entra en el directorio del proyecto y genera el bundle [OfertaBundle](#) ejecutando el siguiente comando:

```
$ php app/console generate:bundle
```

El comando plantea cinco preguntas. A continuación se explica el propósito de cada una y la respuesta recomendada:

- *Bundle namespace*, es el *namespace* bajo el que se creará el *bundle* y por tanto, determina el directorio en el que se guarda el código. El *namespace* se compone de tres partes: la primera es el nombre del proyecto o empresa que desarrolla el bundle ([Cupon](#) en este caso), la segunda parte es opcional e incluye una o más categorías en las que se clasifica el *bundle* (casi siempre se deja vacío) y la última parte es el propio nombre del bundle, que se puede elegir libremente pero siempre debe acabar en [Bundle](#). Respuesta: [Cupon/OfertaBundle](#)
- *Bundle name*, es el nombre con el que se hará referencia al *bundle* dentro de la aplicación. Se recomienda que el nombre sea simplemente el *namespace* sin barras separadoras (en este caso, [CuponOfertaBundle](#)). No obstante, como el nombre de los *bundles* se escribe cientos de veces en el código de la aplicación, mi recomendación es que si los *bundles* son sólo para uso interno de tu aplicación, utilices un nombre lo más corto posible, eliminando la primera palabra del *bundle*: [OfertaBundle](#)
- *Target directory*, a menos que sepas bien lo que estás haciendo, debes aceptar el valor por defecto que se muestra, que coincide con el directorio [src/](#) del proyecto. Respuesta recomendada: pulsar *ENTER* para elegir el valor por defecto
- *Configuration format*, determina el formato utilizado en los archivos de configuración del *bundle*. Puedes elegir entre YAML, XML, PHP o las anotaciones. Respuesta recomendada: [yml](#)
- *Do you want to generate the whole directory structure?*, por defecto Symfony2 genera una estructura básica de directorios (que se explica más adelante), aunque si lo deseas puedes generar una estructura más compleja. Respuesta recomendada: [no](#)

Después de contestar a esas preguntas, el comando muestra un resumen de tus respuestas y te pregunta si realmente quieres generar ese *bundle*. Si estás de acuerdo, contesta [yes](#) a la pregunta *Do you confirm generation?*.

Antes de finalizar la generación del *bundle*, el comando realiza dos últimas preguntas:

- *Confirm automatic update of your Kernel?*, contesta *yes* para que el *bundle* se active en la aplicación después de generararlo (más adelante se explica con detalle).
- *Confirm automatic update of the Routing?*, contesta también *yes* para que el archivo de enrutamiento del *bundle* se cargue automáticamente desde el archivo de enrutamiento general de la aplicación.

Tras estas dos últimas preguntas, el *bundle* ya ha sido generado, activado y configurado correctamente:

```
Generating the bundle code: OK

Checking that the bundle is autoloaded: OK

Confirm automatic update of your Kernel [yes]?
Enabling the bundle inside the Kernel: OK

Confirm automatic update of the Routing [yes]?
Importing the bundle routing resource: OK
```

Generar un nuevo *bundle* en la aplicación con el comando `generate:bundle` produce los siguientes tres cambios.

Primer cambio: se crea la siguiente estructura de archivos y directorios dentro del *bundle*:

```
src/
└── Cupon/
    └── OfertaBundle/
        ├── OfertaBundle.php
        ├── Controller/
        │   └── DefaultController.php
        ├── DependencyInjection/
        │   ├── Configuration.php
        │   └── OfertaExtension.php
        ├── Resources/
        │   ├── config/
        │   │   ├── routing.yml
        │   │   └── services.yml
        │   └── views/
        │       └── Default/
        │           └── index.html.twig
        └── Tests/
            └── Controller/
                └── DefaultControllerTest.php
```

No te preocupes demasiado por entender ahora esta estructura creada. Pronto todo esto te parecerá intuitivo. Si al crear el *bundle* indicas que quieres generar la estructura completa de directorios, el contenido del directorio **Resources** sería el siguiente:

```
src/
└ Cupon/
    └ OfertaBundle/
        ├── ...
        └── Resources/
            ├── config/
            │   ├── routing.yml
            │   └── services.yml
            ├── doc/
            │   └── index.rst
            ├── public/
            │   ├── css/
            │   ├── images/
            │   └── js/
            ├── translations/
            │   └── messages.fr.xliff
            └── views/
                └── Default/
                    └── index.html.twig
```

Segundo cambio: se activa el nuevo *bundle*. Para poder utilizar un *bundle* en una aplicación Symfony2 no basta con crearlo, sino que también hay que activarlo. La activación se realiza en la clase más importante de Symfony, llamada [AppKernel.php](#).

Esta clase se llama así porque es el núcleo de tu aplicación. Cada vez que un usuario realiza una petición a un sitio web creado con Symfony, se instancia una clase para crear la aplicación Symfony2 que se encarga de responder a la petición del usuario.

Para activar el *bundle* **OfertaBundle**, añade la siguiente línea al final del array **\$bundles** del método [registerBundles\(\)](#):

```
// app/AppKernel.php
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
```

```

    new Symfony\Bundle\TwigBundle\TwigBundle(),
    // ...

    new Cupon\OfertaBundle\OfertaBundle(),
);

// ...
}
}

```

Cuando generas el *bundle* con el comando `generate:bundle` esta línea de código se añade automáticamente. Pero si instalas *bundles* desarrollados por terceros, como se verá más adelante, tienes que añadir esa línea manualmente.

Tercer cambio: se importa el enrutamiento del *bundle* desde el archivo de configuración global de la aplicación. A diferencia de symfony 1, en las aplicaciones Symfony2 se recomienda dividir el enrutamiento de la aplicación en varios archivos pequeños. Cada *bundle* dispone de su propio archivo (en `Resources/config/routing.yml`) pero para que se pueda utilizar en la aplicación hay que importarlo desde el archivo global de enrutamiento (más adelante se explica con detalle):

```

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

```

3.4.2 Creando el resto de *bundles*

Una vez creado el primer *bundle*, ya no es necesario que pierdas el tiempo repitiendo el mismo proceso y contestando a las mismas preguntas para el resto de *bundles*. El comando `generate:bundle` también se puede utilizar de forma no interactiva, indicando directamente todas las respuestas.

Crea los otros tres *bundles* del *frontend* con los siguientes comandos:

```

$ php app/console generate:bundle --namespace=Cupon/CiudadBundle
--bundle-name=CiudadBundle --dir=src/ --format=yml --structure=no
--no-interaction

$ php app/console generate:bundle --namespace=Cupon/TiendaBundle
--bundle-name=TiendaBundle --dir=src/ --format=yml --structure=no
--no-interaction

$ php app/console generate:bundle --namespace=Cupon/UsuarioBundle
--bundle-name=UsuarioBundle --dir=src/ --format=yml --structure=no
--no-interaction

```

Comprueba que los comandos anteriores se han ejecutado correctamente:

- Se han creado los directorios [CiudadBundle/](#), [TiendaBundle/](#) y [UsuarioBundle/](#) dentro del directorio [src/Cupon/](#) del proyecto.
- Se han activado los tres nuevos *bundles* en el archivo [AppKernel.php](#)
- Se han importado los archivos de enrutamiento de los tres *bundles* en el archivo [app/config/routing.yml](#)

Sección 2

Frontend

Esta página se ha dejado vacía a propósito

CAPÍTULO 4

Creando las primeras páginas

4.1 La filosofía de Symfony2

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. Symfony2 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 Symfony2:

1. El **usuario** solicita una página mediante su URL
2. **Symfony2** convierte la petición en un objeto 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 Symfony2 la respuesta mediante un objeto de tipo `Response`
4. **Symfony2** convierte el objeto `Response` en el contenido que se pasa al servidor web
5. El **servidor web** devuelve el contenido que le pasa Symfony2

Symfony2 se adapta perfectamente al modelo de petición + respuesta HTTP. Toda su arquitectura 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, Symfony2 *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 Symfony2 dispone de

un artículo que explica muy detalladamente por qué es mejor usar un framework como Symfony2 (http://librosweb.es/symfony_2_3/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 Symfony2. Por el momento sólo se explica cómo conseguir que la aplicación muestre el contenido de una página estática.

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://cupon.local/ayuda>. Cada página de una aplicación Symfony2 debe estar asociada a una ruta del sistema de enrutamiento. Además de los archivos de rutas de los *bundles*, también existe un archivo global de enrutamiento en `app/config/routing.yml`.

Abre ese archivo de configuración y añade las siguientes líneas (ten en cuenta que como se trata de un archivo en formato YAML, todo son espacios en blanco, no tabuladores):

```
# app/config/routing.yml
pagina_ayuda:
    path:      /ayuda
    defaults: { _controller: OfertaBundle:Default:ayuda }
```

La clave `pagina_ayuda` es el nombre de la ruta. Puedes elegirlo libremente, pero debe ser único en toda la aplicación y se recomienda que sea muy corto, ya que los enlaces de las plantillas se construyen utilizando este nombre.

La clave `path` indica la expresión regular que debe cumplir la URL de la petición del usuario para considerar que se trata de esta ruta. No es necesario que añadas el dominio de la aplicación por delante, ya que Symfony2 se encarga de incluirlo automáticamente. En este caso, como la ruta siempre es fija (`/ayuda`) la expresión regular en realidad es una simple cadena de texto.

NOTA En Symfony 2.0 y 2.1, la clave `path` se llamaba `pattern`, por lo que es posible que te la encuentres en muchos tutoriales escritos para esas versiones.

Con todo lo anterior, cuando el usuario solicita la página <http://cupon.local/ayuda>, Symfony2 sabe que en realidad está solicitando la ruta llamada `pagina_ayuda`. Sin embargo, Symfony2 todavía no sabe cómo responder a la solicitud del usuario, ya que todavía no se ha indicado el código que se debe ejecutar.

La clave `defaults` se emplea para indicar el valor por defecto de las variables de la ruta. Aunque la ruta no incluye ninguna variable en su patrón, Symfony2 define automáticamente varias variables especiales. Una de ellas se llama `_controller` (importante el guion bajo por delante). Esta variable indica el controlador que se ejecuta para responder a la solicitud del usuario.

Más adelante se explica detalladamente el concepto de *controlador*, pero por el momento piensa que un *controlador* es un método de una clase PHP. La cadena `OfertaBundle:Default:ayuda` indica el nombre y la localización exacta del controlador gracias a la *notación bundle* y se interpreta de la siguiente manera:

- La primera parte ([OfertaBundle](#)), es el nombre del *bundle* en el que se encuentra el controlador. En este ejemplo hace referencia al directorio `src/Cupon/OfertaBundle/`.
- La segunda parte ([Default](#)), es el nombre de la clase del controlador. Para determinar el archivo PHP concreto, se sujiza su nombre con `Controller` y se busca dentro del directorio `Controller` del *bundle*. En este ejemplo el archivo es `src/Cupon/OfertaBundle/Controller/DefaultController.php`. El valor `Default` es el más utilizado en las aplicaciones Symfony2, ya que es el nombre del controlador por defecto que crea Symfony al generar el *bundle*.
- La tercera parte ([ayuda](#)), se denomina *acción* y es el nombre del método que se ejecuta dentro del controlador. Al nombre indicado se le sujiza la palabra `Action`. Así, en este ejemplo el método es `ayudaAction()`.

La notación `bundle:controlador:acción` puede que te parezca confusa o demasiado larga de escribir. No obstante, pronto comprobarás que hace más conciso el código de las aplicaciones y mucho más flexible, ya que no se manejan rutas del sistema de archivos sino *rutas virtuales* referenciadas a los *bundles* de la aplicación. Además de las rutas, esta notación se emplea en todos los demás componentes de Symfony2, así que asegúrate de haber entendido su funcionamiento básico.

Observa de nuevo la ruta que has añadido en el archivo global de enrutamiento:

```
# app/config/routing.yml
pagina_ayuda:
    path:      /ayuda
    defaults: { _controller: OfertaBundle:Default:ayuda }
```

Symfony2 hace que una configuración tan sencilla como la anterior se interprete como: "cuando el usuario solicite una URL con el aspecto `/ayuda`, ejecuta el método `ayudaAction()` de la clase `src/Cupon/OfertaBundle/Controller/DefaultController.php`".

Si abres el archivo `src/Cupon/OfertaBundle/Controller/DefaultController.php` verás el siguiente código:

```
<?php
namespace Cupon\OfertaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{

    public function indexAction($name)
    {
        return $this->render('OfertaBundle:Default:index.html.twig', array(
            'name' => $name
        ));
    }
}
```

Cuando Symfony2 genera un *bundle*, además de crear el controlador `Default`, añade un método de ejemplo llamado `indexAction()`. Así que reemplaza todo el código de la clase `DefaultController` por lo siguiente:

```
<?php  
namespace Cupon\OfertaBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
  
class DefaultController extends Controller  
{  
  
    public function ayudaAction()  
    {  
        return new Response('Ayuda');  
    }  
}
```

Observa que además del método `ayudaAction()` ahora se ha añadido una sentencia `use` para importar la clase `Response` que se utiliza dentro de `ayudaAction()`. Guarda los cambios, abre un navegador, accede a http://cupon.local/app_dev.php/ayuda y verás una página con la palabra **Ayuda** como único contenido (**si se produce algún error, sigue leyendo hasta el próximo apartado sobre permisos**). Enhорabuena, acabas de programar tu primera página con Symfony2!

4.2.1 Configurando los permisos

La principal causa de errores al ejecutar la aplicación o los comandos de Symfony2 es la mala configuración de los permisos de los directorios `app/cache/` y `app/logs/`, que son los únicos en los que escribe Symfony2. El problema es que en ocasiones escribe en ellos el servidor web y otras veces son los comandos de consola los que escriben.

El usuario con el que se ejecuta el servidor web no suele ser el mismo que el usuario con el que se ejecutan los comandos de consola, por lo que no basta con ejecutar el comando `chmod -R 777` en estos directorios. A continuación se indican las posibles soluciones más comunes, pero también puedes consultar la guía publicada en el sitio [symfony.es](http://symfony.es/documentacion/como-solucionar-el-problema-de-los-permisos-de-symfony2/) sobre cómo solucionar el problema de los permisos de Symfony2 (<http://symfony.es/documentacion/como-solucionar-el-problema-de-los-permisos-de-symfony2/>).

Si tu sistema utiliza Windows, debes cambiar los permisos de `app/cache/` y `app/logs/` para todos los usuarios a través de la pestaña *Seguridad* de las propiedades de esas carpetas. Si esta solución no funciona, sigue leyendo hasta que veas la solución basada en la función `umask()`.

Si tu sistema es Linux o Mac OS X, abre una consola de comandos y ejecuta lo siguiente:

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

```
$ sudo chmod +a "usuario1 allow  
delete,write,append,file_inherit,directory_inherit" app/cache app/logs  
$ sudo chmod +a "usuario2 allow  
delete,write,append,file_inherit,directory_inherit" app/cache app/logs
```

Sustituye `usuario1` por el usuario con el que se ejecuta el servidor web (`www-data` por ejemplo) y `usuario2` por el usuario que utilizas para ejecutar los comandos (`root`, tu nombre de usuario, etc.)

Si esta solución no funciona o si tu sistema no soporta la opción `+a` para el comando `chmod`, prueba los siguientes comandos (sustituyendo de nuevo `usuario1` y `usuario2` por los valores correspondientes):

```
$ sudo setfacl -R -m u:usuario1:rwx -m u:usuario2:rwx app/cache app/logs  
$ sudo setfacl -dR -m u:usuario1:rwx -m u:usuario2:rwx app/cache app/logs
```

Independientemente del sistema operativo que utilices, existe una última posible solución al problema de los permisos. Sólo se recomienda su uso cuando fallan todas las demás opciones, ya que se basa en crear archivos con permisos no muy estrictos.

Añade la instrucción `umask(0002)`; justo al principio de los archivos `app/console`, `web/app.php` y `web/app_dev.php`. Esto hace que los archivos se creen con los permisos `0775`, por lo que todos los usuarios del mismo grupo comparten los permisos de escritura. Si el usuario de la consola y el del servidor web pertenecen a diferentes grupos, utiliza la instrucción `umask(0000)`; para establecer los permisos a `0777`.

Por último, para comprobar que los cambios han surtido efecto, 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 Symfony2, lo primero que debes hacer es probar a borrar la caché con el comando `cache:clear`.

4.2.2 Respondiendo con plantillas

La página desarrollada anteriormente sólo muestra como respuesta la palabra `Ayuda`. Aunque se trata de un buen ejemplo para empezar a entender cómo funciona Symfony2, normalmente no se devuelven palabras o frases sino páginas enteras. Modifica el código del método `ayudaAction()` por lo siguiente:

```
public function ayudaAction()  
{  
    return $this->render('OfertaBundle:Default:ayuda.html.twig');  
}
```

Si ahora actualizas la página de tu navegador (http://cupon.local/app_dev.php/ayuda), ya no verás la palabra *Ayuda* sino que se mostrará un error indicando que Symfony2 no ha podido encontrar la plantilla `OfertaBundle:Default:ayuda.html.twig`.

Las páginas de la aplicación se crean mediante plantillas. Estas no son más que los *moldes* con los que se *fabrican* las páginas que realmente se entregan a los usuarios. En este ejemplo resulta absurdo utilizar una plantilla, ya que la página de ayuda siempre es igual. 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 Symfony2 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, Symfony2 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. Todas las plantillas de este libro utilizan Twig. Si nunca lo has utilizado o si quieras perfeccionar tus conocimientos, lee lo antes posible el Apéndice A (página 517).

El mensaje de error anterior indica que Symfony2 no ha sido capaz de encontrar la plantilla `OfertaBundle:Default:ayuda.html.twig`. En el caso de la *notación bundle* de plantillas, la interpretación es ligeramente diferente al caso de los controladores:

- La primera parte (`OfertaBundle`), es el nombre del *bundle* en el que se encuentra la plantilla. En este ejemplo hace referencia al directorio `src/Cupon/OfertaBundle/`
- La segunda parte (`Default`), es el nombre del directorio en el que se guarda el archivo de la plantilla. Symfony2 busca este directorio dentro del directorio `Resources/views` del *bundle*. Así que en este ejemplo el directorio es `src/Cupon/OfertaBundle/Resources/views/Default/`. El valor `Default` es el más utilizado en las aplicaciones Symfony2, ya que es el nombre de la carpeta que crea por defecto Symfony2 al generar el *bundle*.
- La tercera parte (`ayuda.html.twig`), es el nombre del archivo específico de la plantilla. A su vez se divide en el nombre de la plantilla y las dos extensiones que indican el lenguaje con el que ha sido escrito (`html`) y el motor de plantillas utilizado (`twig`).

Así que para solucionar el error que muestra Symfony, simplemente debes crear el archivo `src/Cupon/OfertaBundle/Resources/views/Default/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.

4.3 Creando todas las páginas estáticas

Además de la página de ayuda, los sitios web suelen disponer de varias páginas estáticas (contacto, privacidad, licencia, términos y condiciones, etc.) A continuación se explica cómo partir del código anterior para extenderlo a cualquier página estática del sitio.

En primer lugar crea una nueva regla de enrutamiento que pueda gestionar todas las páginas estáticas. Abre el archivo `app/config/routing.yml`, borra la regla `pagina_ayuda` y añade lo siguiente:

```
# app/config/routing.yml
pagina_estatica:
    path:      /sitio/{pagina}
    defaults: { _controller: OfertaBundle:Sitio:estatica }
```

La regla ahora se llama `pagina_estatica` y su patrón (`path`) está formado por una parte fija y otra variable. Las partes variables se definen encerrando el nombre de una variable entre las llaves `{` y `}`. Así, el patrón `/sitio/{pagina}` se interpreta como la "*URL que empieza por /sitio/ y continúa con cualquier otra cadena de texto, que a su vez será considerada como el valor de una variable llamada pagina*".

Si el usuario solicita la URL `/sitio/ayuda` se le asigna el valor `ayuda` a la variable `pagina`. Si solicita `/sitio/privacidad`, la regla también se cumple y la variable `pagina` ahora contiene el valor `privacidad`. Si el usuario solicita la URL `/ayuda` esta regla no se tendrá en cuenta por no cumplir con el formato definido por `path`.

Aunque la ruta tenga partes variables, se emplea la misma notación para indicar el controlador a ejecutar (opción `_controller`). De esta forma, el valor `OfertaBundle:Sitio:estatica` se interpreta así:

- `OfertaBundle`, el controlador sigue estando en algún archivo dentro del *bundle* `OfertaBundle`
- `Sitio`, es el nombre del nuevo controlador que se encarga de mostrar las páginas estáticas del sitio. Como se trata de un controlador, su nombre completo será `SitioController.php` y se encontrará en el directorio `Controller/` del *bundle*
- `estatica`, es el nombre de la acción que se ejecuta dentro del controlador. Siguiendo los requerimientos de Symfony2, su método asociado se llama `estaticaAction()`

Así que la nueva ruta se interpreta de la siguiente manera: "*cuando el usuario solicite una URL que empieza por /sitio/ y sigue por cualquier cadena de texto, se guarda esa cadena en una variable llamada pagina y se ejecuta el método staticaAction() de la clase src/Cupon/OfertaBundle/Controller/SitioController.php*".

¿Por qué se ha creado un nuevo controlador? ¿No se podría haber añadido simplemente el nuevo método `estaticaAction()` en el controlador `Default`? Técnicamente esto habría sido lo más fácil, ya que no haría falta crear ningún nuevo controlador. Pero desde el punto de vista de organización del código, crear un nuevo controlador es lo mejor.

El *bundle* `OfertaBundle` se diseñó para guardar todo el código y recursos relacionados con la gestión de las ofertas de la aplicación `Cupon`. Sin embargo ahora estamos creando la lógica para gestionar las páginas estáticas del sitio, algo que no tiene nada que ver con las ofertas.

Si los contenidos estáticos del sitio fueran muy numerosos, lo lógico sería crear un nuevo *bundle* para gestionarlos. En este caso simplemente se van a crear unas pocas páginas estáticas, por lo

que no es necesario complicar la aplicación con un nuevo *bundle*. Por eso se ha elegido uno de los *bundles* principales de la aplicación ([OfertaBundle](#)) para añadir todos los elementos comunes que no encajan en ningún otro *bundle*.

Como es de suponer que el controlador `Default` contendrá muchos métodos para la gestión de las ofertas, es preferible crear un nuevo controlador dedicado exclusivamente a la gestión de las páginas estáticas del sitio y que no interfiera con el controlador `Default` del *bundle*. Así que crea un nuevo archivo llamado `src/Cupon/OfertaBundle/Controller/SitioController.php` con el siguiente contenido:

```
<?php
// src/Cupon/OfertaBundle/Controller/SitioController.php
namespace Cupon\OfertaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class SitioController extends Controller
{
    public function estaticaAction($pagina)
    {
        return $this->render('OfertaBundle:Default:'.$pagina.'.html.twig');
    }
}
```

Cuando la ruta contiene variables, estas se pasan automáticamente al controlador. Para ello sólo debes incluir las variables en la declaración del método utilizando el mismo nombre que en la ruta.

La variable `$pagina` contiene la cadena de texto que el usuario escribe en la URL después de `/sitio/`, así que se puede utilizar como el nombre de la página solicitada. Las plantillas de las páginas estáticas siempre se encuentran en la carpeta `OfertaBundle:Default` y su formato es `.html.twig`, por lo que la única parte que cambia es el propio nombre de la plantilla. Como este es precisamente el valor que guarda la variable `$pagina`, la plantilla completa será `'OfertaBundle:Default:'.$pagina.'.html.twig'`

Si ahora creas el archivo `src/Cupon/OfertaBundle/Resources/views/Default/privacidad.html.twig` y leañades algún contenido, el sitio web ya dispondrá de una página estática con información sobre la privacidad en la URL `/sitio/privacidad`.

Si recuerdas, antes se decidió crear un nuevo controlador llamado `Sitio` para que las páginas estáticas no *ensuciaran* el controlador `Default` del *bundle*. De la misma forma, resulta muy recomendable separar las páginas estáticas del sitio de las plantillas de la aplicación. Para ello, crea la carpeta `src/Cupon/OfertaBundle/Resources/views/Sitio/` y copia en su interior todas las páginas estáticas (`ayuda.html.twig`, `privacidad.html.twig`, etc.)

Por último, actualiza el código del controlador para que ahora busque las plantillas en la carpeta `Sitio`, no en la carpeta `Default`:

```
// src/Cupon/OfertaBundle/Controller/SitioController.php

// Antes
return $this->render('OfertaBundle:Default:'.$pagina.'.html.twig');

// Ahora
return $this->render('OfertaBundle:Sitio:'.$pagina.'.html.twig');
```

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 estáticas 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:

```
# app/config/routing.yml
pagina_estatica:
    path:  /sitio/{pagina}/
    defaults: { _controller: OfertaBundle:Sitio:estatica }
```

Con esta nueva configuración, cuando el usuario accede a `/sitio/ayuda/`, la aplicación responde con el controlador de la ruta `pagina_estatica`. Además, cuando el usuario accede a `/sitio/ayuda` (sin la barra del final) Symfony2 redirige automáticamente a la URL `/sitio/ayuda/`, por lo que también funciona bien. Esta redirección se realiza con una respuesta de tipo 301 (*Moved Permanently*).

Esta es la mejor solución para el problema de *la barra del final*, pero existen otras soluciones alternativas. Una de ellas consiste en duplicar la ruta original, asignándole un nuevo nombre y añadiendo la barra al final:

```
# app/config/routing.yml
pagina_estatica:
    path:  /sitio/{pagina}
    defaults: { _controller: OfertaBundle:Sitio:estatica }

_pagina_estatica:
    path:  /sitio/{pagina}/
    defaults: { _controller: OfertaBundle:Sitio:estatica }
```

En tus plantillas puedes seguir utilizando la ruta `pagina_estatica` para generar los enlaces, pero si algún usuario añade una barra al final, la aplicación sigue funcionando bien. Para que sea más fácil localizar las dos rutas asociadas, utiliza el mismo nombre añadiendo un guión bajo por delante, como muestra la configuración anterior.

Esta nueva solución también se puede mejorar para no tener que duplicar la información sobre el controlador de la ruta (variable `_controller`). Para ello, utiliza el siguiente truco que permite indicarle a Symfony2 que la ruta en realidad redirige a otra ruta:

```
# app/config/routing.yml
pagina_estatica:
    path:  /sitio/{pagina}
    defaults: { _controller: OfertaBundle:Sitio:estatica }

_pagina_estatica:
    path:  /sitio/{pagina}/
    defaults: { _controller: FrameworkBundle:Redirect:redirect,
                route: pagina_estatica }
```

La redirección que realiza Symfony2 en este caso es de tipo [302 \(Found\)](#). Para realizar una redirección [301 \(Moved Permanently\)](#), añade la opción `permanent`:

```
# app/config/routing.yml
#
# ...
#
_pagina_estatica:
    path:  /sitio/{pagina}/
    defaults: { _controller: FrameworkBundle:Redirect:redirect,
                route: pagina_estatica, permanent: true }
```

Por último, el problema de *la barra del final* también se puede solucionar externamente a Symfony2. Utilizando la reescritura de URL de tu servidor web puedes hacer que a Symfony2 siempre le lleven URL sin *la barra del final*, por lo que no habría que modificar las rutas de la aplicación.

CAPÍTULO 5



La base de datos

Las aplicaciones Symfony2 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 1 incluía dos ORM *oficiales*: Doctrine (<http://www.doctrine-project.org/>) y Propel (<http://www.propelorm.org/>). Symfony2 solamente incluye Doctrine, aunque se está trabajando para que Propel vuelva a ser un ORM *oficial*. La versión de Doctrine incluida en Symfony 2.3 es la 2.3.*. Si has utilizado previamente Doctrine 1.x, puedes olvidar todo lo aprendido, ya que no se parecen en nada.

Doctrine2 es un proyecto tan grande que serían necesarios varios libros para explicar todas sus funcionalidades. Las siguientes secciones se centran exclusivamente en la explicación de Doctrine2 desde el punto de vista de los programadores Symfony2. Para desarrollar aplicaciones muy complejas, será necesario que profundices en el estudio de Doctrine2 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, Doctrine2 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 Doctrine2 crea tablas adicionales para representar la relación entre dos entidades.

En las aplicaciones symfony 1, estas clases PHP se generaban automáticamente. Primero definías las tablas y columnas de cada una en un archivo llamado `schema.yml` y después generabas todas las clases con el comando `php symfony doctrine:build --model`.

Las clases que utiliza Symfony2 son mucho más sencillas que las de symfony 1. Se trata de simples clases PHP que no sólo no utilizan un formato especial para su nombre sino que ni siquiera heredan de ninguna otra clase. Por eso estas clases se pueden crear a mano, aunque Symfony2 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 Symfony2 funcionarán con

cualquier base de datos (Oracle, MySQL, PostgreSQL, SQL Server, 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. Las entidades se crean en el directorio `Entity` de cada *bundle*. Inicialmente este directorio no existe, así que créalo siempre que vayas a añadir una entidad a un *bundle*. La localización de cada entidad será la siguiente:

- Entidad `Ciudad` en el *bundle* `CiudadBundle`
- Entidad `Tienda` en el *bundle* `TiendaBundle`
- Entidades `Oferta` y `Venta` en el *bundle* `OfertaBundle`
- Entidad `Usuario` en el *bundle* `UsuarioBundle`

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* `CiudadBundle`:

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

class Ciudad
{}
```

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

```
// src/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

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

NOTA Las propiedades de las entidades de Doctrine2 sólo pueden ser `protected` o `private`, nunca `public`. Así se evitan problemas cuando se accede a una propiedad de un objeto relacionado que todavía no se ha cargado. Este comportamiento se denomina *lazy loading* y se explica con detalle más adelante.

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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

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

Las anotaciones de Doctrine2 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 Doctrine2 con el prefijo `@ORM\`.

La anotación `@ORM\Entity` hace que Doctrine2 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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

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

A continuación, añade más anotaciones para que Doctrine2 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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

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

```

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 Doctrine2 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. Doctrine2 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 Doctrine2	Tipo equivalente SQL	Tipo equivalente PHP
<code>smallint</code>	<code>SMALLINT</code>	<code>integer</code>
<code>integer</code>	<code>INT</code>	<code>integer</code>
<code>bigint</code>	<code>BIGINT</code>	<code>string</code>
<code>float</code>	<code>FLOAT</code>	<code>double</code>
<code>decimal</code>	<code>DECIMAL</code>	<code>double</code>
<code>date</code>	<code>DATETIME</code>	<code>DateTime</code>
<code>time</code>	<code>TIME</code>	<code>DateTime</code>
<code>datetime</code>	<code>DATETIME</code> o <code>TIMESTAMP</code>	<code>DateTime</code>
<code>boolean</code>	<code>BOOLEAN</code>	<code>boolean</code>
<code>string</code>	<code>VARCHAR</code>	<code>string</code>
<code>text</code>	<code>CLOB</code>	<code>string</code>
<code>object</code>	<code>CLOB</code>	<code>Object</code>
<code>array</code>	<code>CLOB</code>	<code>Object</code>

Doctrine2 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`.
- `precision`: es la precisión o número de dígitos significativos que se almacenan para los tipos de datos `decimal`. 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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\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 Doctrine2 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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\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 Doctrine2.

Como las propiedades de la entidad siempre son `protected` o `private`, para que Doctrine2 pueda manipular su valor es necesario añadir los métodos `getter` y `setter` correspondientes:

```

// src/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\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;
}
}

```

Doctrine2 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.

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/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

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

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

```

5.1.2 Creando la entidad Tienda

La entidad que representa a las tiendas de la aplicación se define dentro del *bundle* `TiendaBundle`. Crea el directorio `Entity/` dentro del directorio raíz del *bundle* (`src/Cupon/TiendaBundle/Entity`) y añade dentro una clase llamada `Tienda`:

```

// src/Cupon/TiendaBundle/Entity/Tienda.php
<?php
namespace Cupon\TiendaBundle\Entity;

```

```
class Tienda
{
}
```

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

```
// src/Cupon/TiendaBundle/Entity/Tienda.php
namespace Cupon\TiendaBundle\Entity;

class Tienda
{
    protected $id;
    protected $nombre;
    protected $slug;
    protected $login;
    protected $password;
    protected $salt;
    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/Cupon/TiendaBundle/Entity/Tienda.php
namespace Cupon\TiendaBundle\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", length=255) */
protected $password;

/** @ORM\Column(type="string", length=255) */
protected $salt;

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

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

/** @ORM\ManyToOne(targetEntity="Cupon\CiudadBundle\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="Cupon\CiudadBundle\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 `$ciudad`, en la tabla `Tienda` de la base de datos Doctrine2 crea una columna llamada `ciudad_id` relacionada con la columna `id` de la tabla `Ciudad`. La conversión entre `ciudad` y `ciudad_id` es transparente para el programador, ya que en tu código siempre utilizarás el valor `$ciudad` y será Doctrine2 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 Doctrine2, pero puedes cambiarlo con la anotación `@ORM\JoinColumn`. El siguiente código muestra dos ejemplos equivalentes:

```

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

/**
 * @ORM\ManyToOne(targetEntity="Cupon\CiudadBundle\Entity\Ciudad")
 * @ORM\JoinColumn(name="ciudad_id", referencedColumnName="id")

```

```
 */
protected $ciudad;
```

Doctrine2 soporta las relaciones uno-a-uno, muchos-a-uno, uno-a-muchos, muchos-a-muchos, relaciones autoreferentes, etc. El manual oficial de Doctrine2 explica los detalles de cada una: <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, Symfony2 incluye un comando para añadir los *getters* y *setters* básicos automáticamente. Para ello, ejecuta el siguiente comando:

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

Si ahora vuelves al archivo [src/Cupon/TiendaBundle/Entity/Tienda.php](#), verás que su contenido ha sido modificado para incluir todos los *getters* y *setters*:

```
// src/Cupon/TiendaBundle/Entity/Tienda.php
namespace Cupon\TiendaBundle\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(\Cupon\CiudadBundle\Entity\Ciudad $ciudad)
    {
        $this->ciudad = $ciudad;
    }
}
```

```

        return $this;
    }

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

```

NOTA Los *setters* generados por Doctrine a partir de la versión 2.1 de Symfony incluyen al final la instrucción `return $this;` para permitir el uso de "*interfaces fluidas*", como por ejemplo: `$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. Las conversiones para gestionar la información de la base de datos es algo que realiza Doctrine2 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 potencialmente graves. 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/Cupon/TiendaBundle/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 Symfony2 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 Doctrine2 entity generator

This command helps you generate Doctrine2 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: OfertaBundle: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* `OfertaBundle`, indica el nombre corto `OfertaBundle: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 Doctrine2 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): ruta_foto
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): fecha_publicacion
Field type [string]: datetime

New field name (press <return> to stop adding fields): fecha_expiracion
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 "OfertaBundle:Oferta" Doctrine2 entity  
using the "annotation" format.
```

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

La entidad ya ha sido generada en el archivo [src/Cupon/OfertaBundle/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/Cupon/OfertaBundle/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="Cupon\CiudadBundle\Entity\Ciudad") */
    private $ciudad;

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

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

```
// src/Cupon/OfertaBundle/Entity/Oferta.php

class Oferta
{
    // ...

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

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

    public function setTienda(\Cupon\TiendaBundle\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: UsuarioBundle: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): salt
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): permite_email
Field type [string]: boolean

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

New field name (press <return> to stop adding fields): fecha_nacimiento
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): numero_tarjeta
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/Cupon/UsuarioBundle/Entity/Usuario.php`. Para finalizar la entidad, corrige la propiedad `$ciudad` y su *setter*:

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php

// Antes
class Usuario
{
    // ...

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

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

// Ahora
class Usuario
{
    // ...

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

    public function setCiudad(\Cupon\CiudadBundle\Entity\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 `$fecha_alta`:

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php

class Usuario
{
```

```
// ...

public function __construct()
{
    $this->fecha_alta = 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 Doctrine2 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 Doctrine2.

Como la entidad `Venta` pertenece al *bundle* `OfertaBundle`, el directorio `Entity/` ya existe. Crea dentro un archivo llamado `Venta.php` con el siguiente código:

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

use Doctrine\ORM\Mapping as ORM;

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

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

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

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

    public function getFecha()
```

```
{  
    return $this->fecha;  
}  
  
public function setOferta(\Cupon\OfertaBundle\Entity\Oferta $oferta)  
{  
    $this->oferta = $oferta;  
}  
  
public function getOferta()  
{  
    return $this->oferta;  
}  
  
public function setUsuario(\Cupon\UsuarioBundle\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, simplemente 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="MiBundle:MiEntidad"  
--fields="nombre:string(100) apellidos:string(100) email:string(255)  
fecha_alta: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 Symfony2 para acceder a la base de datos es el nombre y contraseña del usuario y el nombre de la base de datos. Esta información ya la indicaste al instalar Symfony2 mediante el configurador web. Si quieras modificar sus valores, puedes hacerlo 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  
  
    # ...
```

En realidad, el archivo `parameters.yml` no es el lugar del que Doctrine2 obtiene su información. El acceso a la base de datos realmente se configura en el archivo `app/config/config.yml`, 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`. Si quieras, puedes olvidarte del archivo `parameters.yml` reemplazando las opciones de `app/config/config.yml` por sus valores correspondientes:

```
# app/config/config.yml
doctrine:
    dbal:
        driver:    pdo_mysql
        host:      localhost
        port:      ~
        dbname:   cupon
        user:      cupon
        password: cupon
        charset:  UTF8
```

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 Symfony2:

```
$ 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 Doctrine2 (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/) (<http://www.hibernate.org/>) de Java, la mayoría de conceptos te resultarán familiares. Si conoces Doctrine 1, esta es la parte en la que más se diferencia con Doctrine2.

5.3.1 Obteniendo el *entity manager*

Symfony2 crea este objeto automáticamente y lo pone a tu disposición a través del "*contenedor de inyección de dependencias*", un concepto clave del funcionamiento de Symfony2 que se explica detalladamente en el [apéndice B](#) ([página 581](#)) (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 Symfony2.

NOTA En las versiones anteriores de Doctrine2, 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('OfertaBundle: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 (es decir, la tabla de la base de datos en la que se busca). Como es habitual, el nombre de la entidad se indica con la *notación bundle*, siendo la primera parte el nombre del *bundle* y la segunda el nombre de la clase de la entidad.

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, Doctrine2 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('OfertaBundle:Oferta')->find(1);
```

Este método es un poco más largo de escribir, pero es recomendable utilizarlo porque todos los demás métodos de Doctrine2 son similares. 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('OfertaBundle: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('OfertaBundle:Oferta')->findBy(array(
    'revisada' => true
));

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

// Encontrar usuarios de Vitoria-Gasteiz que permitan el envío de emails
$usuarios = $em->getRepository('UsuarioBundle:Usuario')->findBy(array(
    'ciudad' => $ciudad->getId(),
    'permite_email' => 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 Doctrine2 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('UsuarioBundle:Usuario')->findBy(
    array(),
    array('fecha_alta' => 'ASC'),
    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('UsuarioBundle:Usuario')->findBy(
    array('permite_email' => true),
    array('nombre' => 'ASC', 'fecha_nacimiento' => 'DESC'),
    10, // 10 resultados
    30 // empezando en la posición 30
);
```

Imagina que ahora quieras 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 Doctrine2. Este comportamiento hace que Doctrine2 busque automáticamente en la base de datos cualquier información que solicites y no esté disponible. La instrucción `$usuario->getCiudad()` por ejemplo hace que Doctrine2 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, Doctrine2 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('OfertaBundle:Oferta')
    ->findBy(array('revisada' => true));

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

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

// Misma búsqueda, pero utilizando el atajo
$ciudad = $em->getRepository('CiudadBundle: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()`, `fecha_publicacion` 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 Cupon\OfertaBundle\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();
```

Crear una oferta es tan sencillo como instanciar la clase correspondiente a la entidad `Oferta`. Después, se utilizan los *setters* de la entidad para llenar toda la información de la oferta. Para guardarla en la base de datos, se obtiene el *entity manager* y se 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 Doctrine2. Para reducir el número de sentencias SQL ejecutadas, Doctrine2 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('OfertaBundle:Oferta')->find(3);

$oferta->setPrecio($oferta->getPrecio() + 5);

$em->persist($oferta);
$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('UsuarioBundle:Usuario')->findOneByDni('1234567L');

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

Técnicamente, el funcionamiento de Doctrine2 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 Doctrine2

Aunque los *fixtures* se utilizan en todas las aplicaciones web reales, Symfony2 no los soporta *de fábrica*, algo que sí hacía symfony 1. El motivo es que, en realidad, la responsabilidad de los *fixtures* es de Doctrine2, que ha decidido separarlos en un proyecto diferente.

Así que el primer paso para usar los *fixtures* en un proyecto consiste en instalarlos. Gracias al sistema de instalación y actualización de Symfony2 esto es realmente sencillo. Abre el archivo `composer.json` que se encuentra en el directorio raíz del proyecto y añade las siguientes dependencias en la opción `require`:

```
{  
    ...  
  
    "require": {  
        ...  
        "doctrine/doctrine-fixtures-bundle": "dev-master",  
        "doctrine/data-fixtures": "dev-master"  
    }  
}
```

NOTA El archivo `composer.json` y el uso de Composer se explicaron en la sección [Instalando Composer](#) (página 46) del capítulo 3.

Después, ejecuta el siguiente comando para descargar e instalar los nuevos componentes:

```
// si has instalado Composer globalmente  
$ composer update  
  
// si has instalado Composer solamente en tu proyecto  
$ php composer.phar update
```

Por último, activa en el archivo `app/AppKernel.php` el *bundle* `DoctrineFixturesBundle`. Este paso se realiza automáticamente para los *bundles* generados por Symfony2, pero tienes que hacerlo manualmente para todos los demás *bundles* externos que instales:

```
// app/AppKernel.php

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            // ...
            new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
        );

        // ...
    }
}
```

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

Los *fixtures* de Doctrine 1 eran simples archivos YAML, muy fáciles de leer y de escribir. Doctrine2 define los *fixtures* mediante clases PHP, que aunque cuestan un poco más de trabajo crearlas, ofrecen una flexibilidad casi ilimitada.

Por convención, los *fixtures* de Symfony2 se colocan 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* `CiudadBundle` y añade dentro un archivo llamado `Ciudades.php` con el siguiente código:

```
// src/Cupon/CiudadBundle/DataFixtures/ORM/Ciudades.php
namespace Cupon\CiudadBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Cupon\CiudadBundle\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, simplemente ejecuta el siguiente comando:

```
$ php app/console doctrine:fixtures:load
> purging database
> loading Cupon\CiudadBundle\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: *purguing 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 Cupon\CiudadBundle\DataFixtures\ORM\ciudades
```

Como el comando `doctrine:fixtures:load` es bastante largo de escribir, puedes utilizar los atajos de la consola de Symfony2. 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/Cupon/CiudadBundle/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 recomendable 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 `Util.php`. A continuación crea un directorio llamado también `Util/` dentro del *bundle* `OfertaBundle` y guarda dentro la clase:

```

// src/Cupon/OfertaBundle/Util/Util.php
namespace Cupon\OfertaBundle\Util;

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

```

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

```
use Cupon\OfertaBundle\Util\Util;
```

```

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

        // ...
    }
}

```

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

```

// src/Cupon/CiudadBundle/Entity/Ciudad.php
namespace Cupon\CiudadBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Cupon\OfertaBundle\Util\Util;

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

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

```

// src/Cupon/CiudadBundle/DataFixtures/ORM/Ciudades.php
namespace Cupon\CiudadBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Cupon\CiudadBundle\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/Cupon/OfertaBundle/DataFixtures/ORM/Ofertas.php
namespace Cupon\OfertaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Cupon\OfertaBundle\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/Cupon/OfertaBundle/
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=...
--fixtures=...
```

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/Cupon/CiudadBundle/DataFixtures/ORM/Ciudades.php
namespace Cupon\CiudadBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Cupon\CiudadBundle\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 Symfony2. 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 (parecido a symfony 1) 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/Cupon/CiudadBundle/Resources/config/doctrine/Ciudad.orm.yml
Cupon\CiudadBundle\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: Cupon\CiudadBundle\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 CiudadBundle
$ php app/console doctrine:generate:entities CiudadBundle

// Genera sólo la clase de la entidad Ciudad del bundle CiudadBundle
$ php app/console doctrine:generate:entities CiudadBundle: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/Cupon/CiudadBundle/Resources/config/doctrine/Ciudad.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/
doctrine-mapping"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://doctrine-project.org/schemas/orm/
doctrine-mapping http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

  <!-- el atributo repository-class es opcional y define el repositorio
  asociado
      con esta entidad, tal y como se explicará más adelante -->
<entity name="Cupon\CiudadBundle\Entity\Ciudad" table="ciudad"
  repository-class="Cupon\CiudadBundle\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 Doctrine2.

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 OfertaBundle annotation

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

Como todas las entidades de la aplicación se han creado en el *bundle* `OfertaBundle`, seguramente el último paso consistirá en colocar el archivo de cada entidad en su *bundle* correspondiente. No olvides modificar el valor del *namespace* en todas las clases que muevas de sitio.

Generar las entidades mediante ingeniería inversa es un proceso sencillo y que funciona muy bien, ya que Doctrine2 tiene en cuenta la información de las tablas, columnas, índices, claves primarias y externas. No obstante, el manual de Doctrine2 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.

CAPÍTULO 6

Creando la portada

El capítulo 4 (página 69) introdujo el funcionamiento interno de Symfony2 creando las páginas estáticas de la aplicación. En este capítulo se explica cómo desarrollar la portada de la aplicación, una página compleja y completamente dinámica.

Recordando el funcionamiento de la aplicación explicado en el capítulo 2 (página 33), 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

Symfony2 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: "*Symfony2 no es un framework MVC. Symfony2 sólo proporciona herramientas para la parte del Controlador y de la Vista. La parte del Modelo es responsabilidad tuya, aunque existen librerías para integrar fácilmente los ORM más conocidos, como Doctrine y Propel*" (fuente: <http://fabien.potencier.org/article/49/what-is-symfony2>).

En cualquier caso, resulta esencial conocer cómo se aplican los principios fundamentales de la arquitectura MVC a las aplicaciones Symfony2. Observa el siguiente esquema simplificado del funcionamiento interno de Symfony2 (al final de este capítulo se muestra un esquema mucho más técnico y detallado):

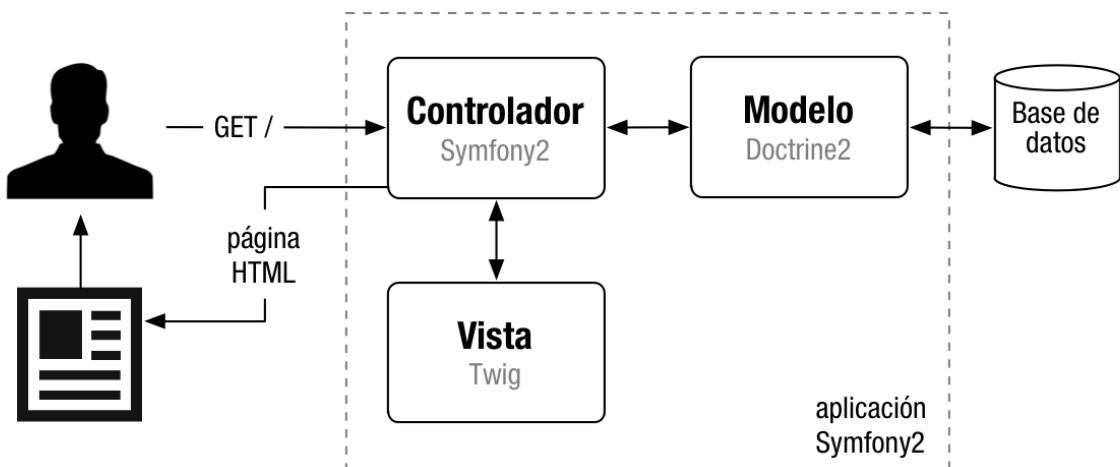


Figura 6.1 Esquema simplificado de la arquitectura interna de Symfony2

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. Symfony2 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 Symfony2, el funcionamiento interno siempre es el mismo: 1) el **Controlador** manda y ordena, 2) el **Modelo** busca la información que se le pide, 3) la **Vista** crea páginas con plantillas y datos.

6.2 El enrutamiento

El sistema de enrutamiento *transforma* URL en *controladores*. Más en concreto, determina cuál es el controlador que se debe ejecutar para cada URL solicitada por los usuarios. Esta transformación se realiza mediante la información de las rutas, normalmente configurada en archivos YAML (también se pueden configurar mediante anotaciones y archivos XML).

El archivo principal de enrutamiento de Symfony2 es [app/config/routing.yml](#). Aunque puedes incluir todas las rutas de la aplicación en ese archivo, cuando una aplicación tiene muchas rutas se recomienda distribuirlas en varios archivos. Para ello, cada *bundle* puede incluir un archivo de rutas en [Resources/config/routing.yml](#).

Siguiendo esta recomendación, sólo vamos a incluir en [app/config/routing.yml](#) las rutas *principales* de la aplicación, como por ejemplo la portada. Abre ese archivo de configuración y añade las siguientes líneas al final del todo:

```
# app/config/routing.yml
portada:
    path:      /
    defaults: { _controller: OfertaBundle:Default:portada }
```

Tal y como se explicó al crear la primera página (página 70), cada ruta se define mediante un nombre único ([portada](#) en este caso), una expresión regular que deben cumplir las URL (en el caso de la portada es simplemente `/`) y el nombre del controlador que se ejecuta para responder a la petición (mediante una clave especial llamada [_controller](#)).

Recuerda que los controladores se referencian mediante la *notación bundle*, por lo que `OfertaBundle:Default:portada` se interpreta como el método `portadaAction()` de la clase `DefaultController.php` que se encuentra en `src/Cupon/OfertaBundle/Controller/`.

¿Por qué la portada se genera dentro del *bundle* `OfertaBundle`? La portada es una de las páginas más especiales de la aplicación. A menos que tengas un *bundle* dedicado a gestionar todas las páginas especiales (la portada, las páginas estáticas, la página de contacto, etc.) se recomienda que generes la página dentro del *bundle* cuyos contenidos se parezcan más a los de la portada. Como en este caso la portada muestra la información de la oferta del día, lo más lógico es colocarla dentro de `OfertaBundle`.

6.3 El controlador

El *controlador* es la parte de la aplicación que contiene lo que se llama la *lógica de negocio*, que es una forma elegante de decir que cada *controlador* se encarga de una funcionalidad completa de la aplicación. Como se verá en los siguientes capítulos, los *controladores* hacen uso de otros componentes para obtener la información y para generar las páginas, por lo que su código no suele ser muy largo.

Los *bundles* generados automáticamente por Symfony2 incluyen un controlador por defecto llamado `Default`. Así que abre el archivo de ese controlador y copia el siguiente código:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
namespace Cupon\OfertaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    public function portadaAction()
    {
    }
}
```

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/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();
}
```

Para buscar la oferta del día, puedes hacer uso del método `findOneBy()` de Doctrine2 estableciendo como condiciones que la oferta sea de una determinada ciudad y que se haya publicado hoy:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction()
```

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

    $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(array(
        'ciudad'          => 1,
        'fecha_publicacion' => 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 Symfony2, tal y como se explicó en la sección de *namespaces* (página 22) del capítulo 1.

El objeto `$oferta` ya contiene todos los datos de una entidad de tipo `Oferta` que representa a 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()` del controlador. Su primer argumento indica, en *notación bundle*, la plantilla a utilizar. Su segundo argumento es opcional y permite indicar un array con todas las variables que se pasan a la plantilla:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction()
{
    $em = $this->getDoctrine()->getManager();

    $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(array(
        'ciudad'          => 1,
        'fecha_publicacion' => new \DateTime('today')
    ));

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

El código del controlador ya está completo. Al usuario se le devolverá la página que Symfony2 genere mediante la plantilla `portada.html.twig` del *bundle* `OfertaBundle` 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 517) en el que se explica detalladamente toda su sintaxis. De lo contrario no entenderás nada de lo que se desarrolla a continuación.

Las plantillas del *bundle* se encuentran por defecto en el directorio `Resources/views/Default/`. Crea un archivo llamado `portada.html.twig` dentro de ese directorio 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">
            
            <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 empezar a añadir la programación con 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 dia</a></li>
            <li><a href="#">Ofertas recientes</a></li>
            <li><a href="#">Mis ofertas</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 dinámicamente en el mismo instante que se *renderiza* la plantilla. La gran ventaja respecto a escribir a mano las URL es que si modificas el archivo de enrutamiento, 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_estatica` 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_estatica', { 'pagina': 'ayuda' }) }}">
        Ayuda
    </a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'contacto' }) }}">
        Contacto
    </a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'privacidad' }) }}">
        Privacidad
    </a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'sobre-nosotros' }) }}">
        Sobre nosotros
    </a>
</footer>
```

Los arrays asociativos de Twig obligan a incluir entre comillas tanto el valor como la clave de cada elemento. Cuando se *renderice* la plantilla, Symfony2 utilizará el patrón de la ruta (`/sitio/{pagina}`) y sustituirá todas las variables por los valores indicados en el array asociativo que se pasa. 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 interpretable como una fecha y hora (`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">
    

    <p class="precio">{{ oferta.precio }} &euro; <span>{{ oferta.descuento
}}</span></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 Symfony2 la carpeta pública es `web/` (el valor por defecto que utiliza Symfony) por lo que las URL generadas por `asset()` son del tipo `/web/...`

Como las tiendas podrán crear ofertas desde la intranet, 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>0ferta activada</strong> por superar las
        <strong>{{ oferta.umbral }}</strong> compras necesarias
    {% endif %}
</div>
</section>
```

La propiedad `fecha_expiracion` 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 Doctrine2 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 Doctrine2, 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, Doctrine2 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 Doctrine2 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://cupon.local/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

Symfony2 mantiene una de las características más populares y poderosas de symfony 1: los entornos de ejecución. Una aplicación de Symfony2 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://cupon.local/app_dev.php y después accede a <http://cupon.local/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://cupon.local/app_dev.php/esto-no-existe y después accede a <http://cupon.local/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. Symfony2 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 las pruebas unitarias y funcionales (como se explica en el capítulo 11 (página 307)).

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 Symfony2 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, Symfony2 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 }  
# ...
```

El mecanismo de importación es una versión mejorada de la *configuración en cascada* de symfony 1. 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.

Además de modificar el valor de las opciones existentes, el archivo `app/config/config_dev.yml` también añade opciones que no existen en el archivo `app/config/config.yml` general: opciones `web_profiler`, `toolbar` y `intercept_redirects`.

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 casi infinita que ofrece Symfony2.

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 Symfony2 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

Symfony2 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 Doctrine2, 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 quieras 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 --no-warmup --env=prod
```

Si se produce algún error al ejecutar los comandos anteriores, repasa el apartado de configuración de permisos ([página 72](#)) del capítulo 4.

6.5.4 Creando un nuevo entorno

Symfony2 incluye inicialmente tres entornos de ejecución: `dev`, `prod`, `test`. A estos tres, puedes añadir tantos entornos propios como quieras. En primer lugar, elige un nombre único para el entorno, como por ejemplo `integracion`.

Después, copia el archivo `web/app.php` o `web/app_dev.php` en función de si el nuevo entorno se parece más a desarrollo o a producción respectivamente. Guarda el nuevo archivo con el nombre `web/app_integracion.php` y haz los cambios oportunos en su código. El único cambio obligatorio es el de la siguiente línea que establece el nombre del entorno:

```
// web/app_integracion.php

// ...
$kernel = new AppKernel('integracion', false);
```

Por último, crea un nuevo archivo de configuración llamado `app/config/config_integracion.yml`. El contenido más simple posible para que la aplicación funcione es el siguiente:

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

Ahora ya puedes ejecutar la aplicación bajo el nuevo entorno accediendo a la URL `http://cupon.local/app_integracion.php`. Si no quieres que cualquier usuario pueda acceder libremente a este entorno, puedes emplear el mismo truco que el entorno de desarrollo:

```
// web/app_dev.php
```

```

if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !in_array(@$_SERVER['REMOTE_ADDR'], array(
        '127.0.0.1',
        '::1',
    )))
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).
        ' for more information.');
}

// ...

```

6.6 Depurando errores

Cometer errores es inevitable al desarrollar una aplicación, por lo que Symfony2 no trata de evitar que los cometas sino que te ayuda a encontrar lo más fácil y rápido posible la causa del error. Las herramientas de depuración que incluye Symfony2 son tan avanzadas que lo sitúan un escalón por encima de cualquier otro framework PHP:

- **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 Symfony2. 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 Symfony2.

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 que se está visualizando. Cuando se encuentra activa, en la parte inferior de todas las páginas se incluye una barra con la siguiente información:



Figura 6.2 Barra de depuración web normal de Symfony2

La información de la barra se muestra en diferentes paneles. Por defecto, y de izquierda a derecha, estos son los paneles que incluye la barra:

- **Panel de Symfony:** versión de Symfony2 sobre la que se ejecuta la aplicación ([2.3.X](#)).
- **Panel de PHP:** si colocas el ratón encima del icono de PHP, verás la versión de PHP que utiliza el servidor. Además, se indica si el servidor tiene instalado Xdebug y alguna de las extensiones para acelerar el rendimiento de PHP, como por ejemplo APC.
- **Panel de configuración:** entorno de ejecución (`dev`) y el *token* de la página ([506d50a227883](#)), que es una cadena de texto aleatoria de trece caracteres alfanuméricos que identifica de forma única no sólo la página sino la ejecución específica que ha generado la página actual. Si actualizas la misma página, se genera un nuevo *token*.
- **Panel de la petición:** nombre del controlador (`DefaultController`) y la acción (`portadaAction`) que se ejecuta; nombre de la ruta seleccionada para atender la petición (`portada`); código de estado HTTP devuelto por el servidor ([200](#)).
- **Panel de excepciones:** indica el número de excepciones producidas durante la ejecución de la aplicación. En la imagen anterior no aparece, ya que sólo se muestra cuando se produce al menos una excepción.
- **Panel del tiempo:** milisegundos que ha tardado la página en ser generada ([31](#)).
- **Panel de memoria:** memoria consumida para generar la página ([4.0 MB](#)).
- **Panel de seguridad:** *login* del usuario conectado a la aplicación o `anon.` (de *anónimo*) si el usuario no está logueado.
- **Panel de la base de datos:** número de consultas necesarias para generar la página ([2](#)) y tiempo total consumido por las consultas ([4.96 ms](#)). El número de consultas puede ser mayor de lo que esperas debido al *lazy loading* y las consultas automáticas de Doctrine2, tal y como se explicó anteriormente.

La barra de depuración web está diseñada de forma *responsive* o adaptativa, así que muestra más o menos información en función de la anchura de tu navegador. Esto es muy útil cuando accedes a la aplicación a través de un móvil o *tablet*:



Figura 6.3 Barra de depuración web simplificada de Symfony2

Aunque por defecto la barra de depuración se muestra en la parte inferior de cada página, si lo prefieres puedes mostrarla en la parte superior estableciendo el valor `top` para la opción `position`:

```
# app/config/config_dev.yml

# ...
web_profiler:
    toolbar: true
    position: top
    intercept_redirects: false
```

En la configuración anterior, la opción `toolbar: true` es la que hace que se muestre la barra y la opción `intercept_redirects` permite detener las redirecciones automáticas de páginas para mostrar toda la información disponible hasta el momento de la redirección.

Cuando se produce un error en la página el panel más importante es el de las excepciones. Pero cuando visualizas una página sin errores, los paneles más importantes son los del tiempo, memoria y base de datos. Si sus valores se disparan sabrás que los últimos cambios introducidos están penalizando el rendimiento de la aplicación.

6.6.2 El *profiler*

La mayoría de iconos que muestran los paneles de información se pueden pinchar para mostrar más información relacionada. Pincha sobre el *token* de la página (la cadena de texto aleatoria que se muestra en el panel de configuración de la barra) para acceder al panel principal del *profiler*.

El *profiler* 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*.

The screenshot shows the Symfony profiler interface. On the left is a sidebar with icons for CONFIG, REQUEST, EXCEPTION, EVENTS, LOGS, TIMELINE, ROUTING, SECURITY, E-MAILS, and DOCTRINE. Below this is a search bar and an IP input field. The main content area has a header "View all Profile for: GET http://cupon.local/app_dev.php/es/barcelona by 127.0.0.1". It contains three sections: "Project Configuration" (with keys like Symfony version, Application name, Environment, and Debug), "PHP configuration" (with keys like PHP version, Xdebug, PHP acceleration, XCache, APC, EAccelerator, and Full PHP configuration), and "Active bundles" (listing AsseticBundle with its path). A search bar at the top right allows searching the Symfony website.

Figura 6.4 Panel principal del profiler de Symfony2

NOTA Para poder acceder al *profiler* vía web sus rutas deben estar activadas. Por defecto Symfony2 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
```

El *profiler* divide su información en diez secciones:

- **Sección config:** configuración de PHP; configuración básica del proyecto (versión de Symfony2, entorno, etc.); lista de los *bundles* que están activos en la aplicación.
- **Sección request:** parámetros de las variables `$_GET`, `$_POST`, `$_COOKIE` y `$_SERVER`; cabeceras de la petición y la respuesta; atributos de la sesión.
- **Sección exception:** muestra toda la información de la excepción producida en la aplicación (vacía en el que caso de que no se haya producido ningún error).

- **Sección events:** listado de los eventos producidos durante la ejecución de la aplicación y los *listeners* asociados que se han ejecutado; listado de los eventos que no se han producido y por tanto, listado de *listeners* que no se han ejecutado durante la petición.
- **Sección logs:** todos los mensajes de log generados por esta petición. Muestra la misma información que los archivos de log, pero así es mucho más cómodo consultarla, ya que en los archivos de log se mezclan los mensajes de todas las peticiones.
- **Sección timeline:** muestra visualmente cómo se reparte el tiempo de ejecución entre las diferentes partes de la aplicación. Así es más fácil detectar los elementos lentos de tu aplicación.
- **Sección routing:** información básica sobre la ruta y listado completo de todas las rutas que ha descartado Symfony2 antes de considerar que esta ruta es la que debe servir la petición. Esta información es útil para ajustar el orden de las rutas y mejorar el rendimiento de la aplicación.
- **Sección security:** información del usuario logueado en la aplicación.
- **Sección emails:** si la petición ha enviado algún email, se muestra toda su información.
- **Sección doctrine:** consultas realizadas durante la petición y tiempo consumido por cada una; nombre de la conexión utilizada para la base de datos; nombre del *entity manager* utilizado para las consultas.

La información que se muestra sólo es válida para una ejecución determinada de esta petición. Cada vez que se recarga la página, se genera un nuevo *token* y se guarda nueva información. Obviamente, si recargas la misma página muchas veces sin hacer cambios en su código, la información de los diferentes *tokens* será prácticamente la misma.

El *profiler* también incluye en la parte inferior izquierda un buscador con el que puedes obtener cualquier información pasada y así compararla con la actual. Si quieras comparar el rendimiento de una página, utiliza el campo *URL* del buscador para encontrar toda la información guardada sobre esa página:

The screenshot shows the Symfony2 Profiler's search interface. On the left, there is a search form with fields for IP, Method, URL (which is set to '/es/barcelona' and highlighted with a red box), Token, and Limit (set to 10). Below this is an 'ADMIN' section with links for Purge, Export, Import, and file upload buttons. On the right, a 'View all' button is followed by the URL and the date ('Thu, 04 Oct 201X 11:12:52'). The main area is titled 'Search Results' and contains a table of log entries. The table has columns for Token, IP, Method, URL, and Time. The data shows multiple entries for the URL '/es/barcelona' at IP 127.0.0.1, all using GET method and occurring on Thu, 04 Oct 201X 11:12:52.

Token	IP	Method	URL	Time
506d53144e679	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:12:52
506d53144c946	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:12:52
506d530081766	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:12:32
506d53007ea81	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:12:32
506d530082f8d	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:12:32
506d50a225d2b	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:02:26
506d50a221eff	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:02:26
506d50a227883	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 11:02:26
506d4c44e3da0	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 10:43:48
506d4c44d71d1	127.0.0.1	GET	http://cupon.local/app_dev.php/es/barcelona	Thu, 04 Oct 201X 10:43:48

Figura 6.5 Buscador histórico del profiler de Symfony2

El buscador también te permite buscar un *token* específico (difícil de usar en la práctica) y buscar por IP, lo que es muy útil por ejemplo para encontrar información sobre todas las pruebas locales realizadas desde la IP `127.0.0.1`.

Por defecto el *profiler* sólo guarda la información de las peticiones que producen errores. Por eso en el entorno de desarrollo Symfony2 añade la opción de configuración `only_exceptions` con el valor `false` para guardar la información de todas las páginas:

```
# app/config/config_dev.yml
framework:
    profiler: { only_exceptions: false }
```

Para no penalizar el rendimiento, en el entorno de producción sólo se guardan los datos de las últimas 500 ejecuciones que han producido algún error. Si lo necesitas, también puedes restringir las zonas del sitio web o las direcciones IP desde las que se registran datos en el *profiler*. La documentación oficial de Symfony2 explica la configuración necesaria para ello: <http://symfony.com/doc/current/book/internals.html#configuration>

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 Symfony2 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 decenas de 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, Symfony2 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 Symfony2.

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 Symfony2 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
```

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 los siguientes tipos de *handlers*:

- `stream`, guarda los mensajes en un *stream* de PHP, normalmente un archivo.
- `rotating_file`, guarda los mensajes en un archivo y crea un nuevo archivo de log cada día. El número máximo de archivos se puede controlar con la opción `max_files`, aunque se recomienda el uso de herramientas como `logrotate`.

- `firephp`, muestra los mensajes a través de FirePHP (<http://www.firephp.org/>) , lo que permite visualizarlos dentro de la consola de mensajes de Firebug.
- `native_mailer`, envía los mensajes por email utilizando la función `mail()` de PHP.
- `swift_mailer`, envía los mensajes por email mediante una instancia de SwiftMailer.
- `syslog`, envía los mensajes al `syslog` del sistema.
- `fingers_crossed`, se queda con todos los mensajes de log hasta que se produce un error del nivel establecido. En ese caso, envía todos los mensajes guardados hasta ese momento al `handler` que tiene asociado. En otras palabras, mientras no se produzca un error del nivel establecido, no se ve ningún mensaje de log. Si se produce el error, se ven todos sus detalles.

El entorno de desarrollo utiliza un `handler` simple de tipo `stream`. El entorno de producción utiliza el tipo `finger_crossed`, por lo que no verás ningún mensaje a menos que se produzca una situación de error.

Monolog también define otros `handlers` menos utilizados, de los que puedes obtener más información en su sitio web: `NullHandler`, `BufferHandler`, `GroupHandler` y `TestHandler`. 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: error
            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 Symfony2, 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()`

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 Symfony2 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 Doctrine2.
- `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:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction()
{
    // ...

    $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(array(
        'ciudad'          => 1,
        'fecha_publicacion' => new \DateTime('today')
    ));

    // ...
}
```

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:
    cupon.ciudad_por_defecto: '1'
```

El nombre del parámetro puedes elegirlo libremente, pero se recomienda seguir una estructura de tipo *namespace: vendor.bundle.opcion* (ejemplo: `cupon.ofertabundle.ciudad_por_defecto`). Si el parámetro se utiliza en diferentes partes de la aplicación, no hace falta que incluyas el nombre del *bundle*.

Para obtener el valor de un parámetro global, utiliza el método `getParameter()` dentro de la acción del controlador:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction()
{
    // ...

    $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(array(
        'ciudad' => $this->container->getParameter('cupon.ciudad_por_defecto'),
        'fecha_publicacion' => 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://cupon.local/> 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.

El primer cambio necesario consiste en definir dos rutas para la portada, una con la ciudad y otra sin la ciudad:

```
# app/config/routing.yml  
  
# Ruta completa de la portada (con el slug de la ciudad)  
portada:  
    path:      /{ciudad}  
    defaults: { _controller: OfertaBundle:Default:portada }  
  
# Ruta simple de la portada  
_portada:  
    path:      /  
    defaults: { _controller: OfertaBundle:Default:portada }
```

La ruta llamada `portada` es la *ruta oficial* de la portada, ya que se utilizará en todas las plantillas de la aplicación. Y la ruta `_portada` es la que utilizan los usuarios para acceder a la aplicación. Como no se va a utilizar internamente, no importa demasiado su nombre y por eso simplemente se añade el guión `_` por delante.

En cualquier caso, las dos rutas están asociadas al mismo controlador (`OfertaBundle:Default:portada`), que es el que se encargará de redirigir de una a otra. Para ello, añade el siguiente código en el controlador de la portada:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php  
namespace Cupon\OfertaBundle\Controller;  
// ...  
use Symfony\Component\HttpFoundation\RedirectResponse;  
  
class DefaultController extends Controller  
{  
    public function portadaAction($ciudad = null)  
    {  
        if (null == $ciudad) {  
            $ciudad = $this->container  
                ->getParameter('cupon.ciudad_por_defecto');  
  
            return new RedirectResponse(  
                // URL de la portada con la ciudad
```

```

        $this->generateUrl('portada', array('ciudad' => $ciudad))
    );
}

// ...
}
}

```

Si desde la ruta no se pasa ningún parámetro (como sucede en la ruta `_portada`), la variable `$ciudad` será `null`. En este caso, se obtiene el valor de la ciudad por defecto y se redirige al usuario a la portada de esa ciudad. Las redirecciones se realizan devolviendo un objeto de tipo `RedirectResponse` (no olvides importar su clase con la instrucción `use`).

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://cupon.local/app_dev.php`, serás redirigido automáticamente a la portada de la ciudad por defecto (`http://cupon.local/app_dev.php/1` en este ejemplo).

Por último, observa que ahora la ruta `portada` requiere que se le pase un valor que se asignará a la variable `ciudad`. Si no puedes modificar todas las plantillas de la aplicación, puedes asignar un valor por defecto a esta variable mediante la opción `defaults` de la ruta:

```
# app/config/routing.yml

portada:
    path: /{ciudad}
    defaults: { _controller: OfertaBundle:Default:portada, ciudad: null }
```

Otra solución más elegante consiste en que la plantilla determine automáticamente la ciudad activa de la aplicación y se la pase a la ruta `portada`. Esta es la técnica que se explica más adelante al final de la sección *Refactorizando la Vista* (página 151) de este capítulo.

A partir de la versión 2.1 de Symfony es posible incluir los parámetros de configuración directamente en las rutas. De esta forma, puedes hacer que la ruta `_portada` redirija a la ruta `portada` incluyendo el valor adecuado para el parámetro `ciudad`:

```
# app/config/routing.yml

# ...

_portada:
    path: /
    defaults: { _controller: FrameworkBundle:Redirect:redirect,
                route: portada, ciudad: '%cupon.ciudad_por_defecto%' }
```

```
# Ruta de la portada completa del sitio (con _locale y el slug de la ciudad)
portada:
    path:      /{ciudad}
    defaults: { _controller: OfertaBundle:Default:portada }
```

Para utilizar un parámetro de configuración como valor de alguna variable de la ruta, encierra su nombre entre los caracteres % y %. Así, el valor `%cupon.ciudad_por_defecto%` se interpreta como la opción `cupon.ciudad_por_defecto` definida bajo la clave `parameters` del archivo `app/config/config.yml`.

Después de este cambio, también se puede simplificar el código del controlador que genera la portada. En concreto, puedes eliminar el código que comprueba si la variable `$ciudad` vale `null` y después redirige a la ruta `portada`:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php

class DefaultController extends Controller
{
    public function portadaAction($ciudad)
    {
        // if (null == $ciudad) {
        //     $ciudad = $this->container
        //             ->getParameter('cupon.ciudad_por_defecto');
        //
        //     return new RedirectResponse(
        //         $this->generateUrl('portada', array('ciudad' => $ciudad))
        //     );
        // }

        // ...
    }
}
```

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/Cupon/OfertaBundle/Controller/DefaultController.php
namespace Cupon\OfertaBundle\Controller;
// ...

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

```

$oferta = $em->getRepository( ... );

if (!$oferta) {
    throw $this->createNotFoundException(
        'No se ha encontrado la oferta del día en la ciudad seleccionada'
    );
}

// ...
}
}

```

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 Symfony2.

Después de todos estos cambios, el código del controlador resultante es el siguiente:

```

// src/Cupon/OfertaBundle/Controller/DefaultController.php
namespace Cupon\OfertaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Exception\NotFoundException;

class DefaultController extends Controller
{
    public function portadaAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();

        $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(array(
            'ciudad' => $ciudad,
            'fecha_publicacion' => new \DateTime('today')
        ));

        if (!$oferta) {
            throw $this->createNotFoundException(
                'No se ha encontrado la oferta del día'
            );
        }

        return $this->render('OfertaBundle:Default: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 o en otros controladores. Por tanto, la siguiente sección se centra en refactorizar la parte del modelo.

6.8 Refactorizando el Modelo

Las consultas de Doctrine2 se realizan a través de un objeto de tipo `EntityRepository` obtenido mediante el método `getRepository()` del *entity manager*:

```
| $oferta = $em->getRepository('OfertaBundle:Oferta')->findOneBy(...);
```

Todas las entidades definidas en Doctrine2 disponen de su propio repositorio, con el que se pueden realizar las consultas `find()`, `findAll()`, `findBy()` y `findOneBy()`. El inconveniente de estos métodos 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 = $em->getRepository('OfertaBundle:Oferta')->findOfertaDelDia($ciudad);
```

Como `findOfertaDelDia()` no es un método definido en Doctrine2, es necesario crearlo en el repositorio propio de las entidades de tipo `Oferta`. Crea un archivo llamado `OfertaRepository.php` en el directorio `Entity/` del bundle `OfertaBundle` y añade el siguiente código:

```
// src/Cupon/OfertaBundle/Entity/OfertaRepository.php  
namespace Cupon\OfertaBundle\Entity;  
  
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 Doctrine2.

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 Doctrine2.

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/Cupon/OfertaBundle/Entity/Oferta.php

/**
 * @ORM\Entity(repositoryClass="Cupon\OfertaBundle\Entity\OfertaRepository")
 */
class Oferta
{
    // ...
}
```

Añadiendo simplemente 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

Doctrine2 define un lenguaje propio llamado DQL con el que se pueden 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 OfertaBundle: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 OfertaBundle:Oferta o`', la sección `FROM` indica las entidades sobre las que se realiza la consulta. La entidad se especifica mediante su *notación bundle* (`nombre-bundle:nombre-entidad`). 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 OfertaBundle: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 OfertaBundle:Oferta o
    WHERE o.ciudad = 1
    AND o.fecha_publicacion = "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 OfertaBundle:Oferta o
     WHERE o.ciudad = :ciudad
       AND o.fecha_publicacion = :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 OfertaBundle:Oferta o
     WHERE o.ciudad = :ciudad
       AND o.fecha_publicacion = :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 Doctrine2, 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 OfertaBundle:Oferta o
     JOIN o.ciudad c
      WHERE c.slug = :ciudad
        AND o.fecha_publicacion = :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 `OfertaBundle:Oferta` y `CiudadBundle: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. Doctrine2 define otro tipo de `JOIN` mucho más interesante para las aplicaciones web: los *fetch JOIN*. Observa las dos siguientes consultas DQL:

```
SELECT o FROM OfertaBundle:Oferta o JOIN o.ciudad c
WHERE c.slug = '...' AND o.fecha_publicacion = '...'

SELECT o, c FROM OfertaBundle:Oferta o JOIN o.ciudad c
WHERE c.slug = '...' AND o.fecha_publicacion = '...'
```

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 OfertaBundle: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 UsuarioBundle:Usuario u WHERE u.email LIKE 'anonimo%'
```

NOTA El manual oficial de Doctrine2 incluye muchos más ejemplos de consultas DQL: <http://www.doctrine-project.org/docs/orm/2.1/en/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/Cupon/OfertaBundle/Entity/OfertaRepository.php
namespace Cupon\OfertaBundle\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 OfertaBundle:Oferta o
                JOIN o.ciudad c JOIN o.tienda t
                WHERE o.revisada = true
                AND o.fecha_publicacion < :fecha
                AND c.slug = :ciudad
                ORDER BY o.fecha_publicacion 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:
    cupon.ciudad_por_defecto: '1'

# ahora
parameters:
    cupon.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_estatica', { 'pagina': 'ayuda' }) }}">Ayuda</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'contacto' }) }}>Contacto</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'privacidad' }) }}>Privacidad</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'sobre-nosotros' }) }}>Sobre nosotros</a>
</footer>
</div></body>

</html>

```

Después se guarda ese código común en la plantilla del *layout*. Como se trata de una plantilla utilizada en todos los *bundles* de la aplicación, se recomienda guardarla 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 539):

```

<!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. Como el *layout* no se encuentra dentro de ningún *bundle*, la notación necesaria para localizar su plantilla es `::nombre-plantilla.html.twig`:

```

{#
    src/Cupon/OfertaBundle/Resources/views/Default/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:

```

{#
    src/Cupon/OfertaBundle/Resources/views/Default/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">
    {{ 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 }}</strong> compras<br/> para activar
la oferta
        {% else %}
            <strong>Oferta activada</strong> por superar las <strong>{{
oferta.umbral }}</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.has('ciudad')
    ? app.request.attributes.get('ciudad')
    : ciudad_por_defecto %}

{# ... #}

<header>
    <h1><a href="{{ path('portada', { 'ciudad': ciudadSeleccionada }) }}>CUPON</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 Symfony2

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 Symfony2. 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:

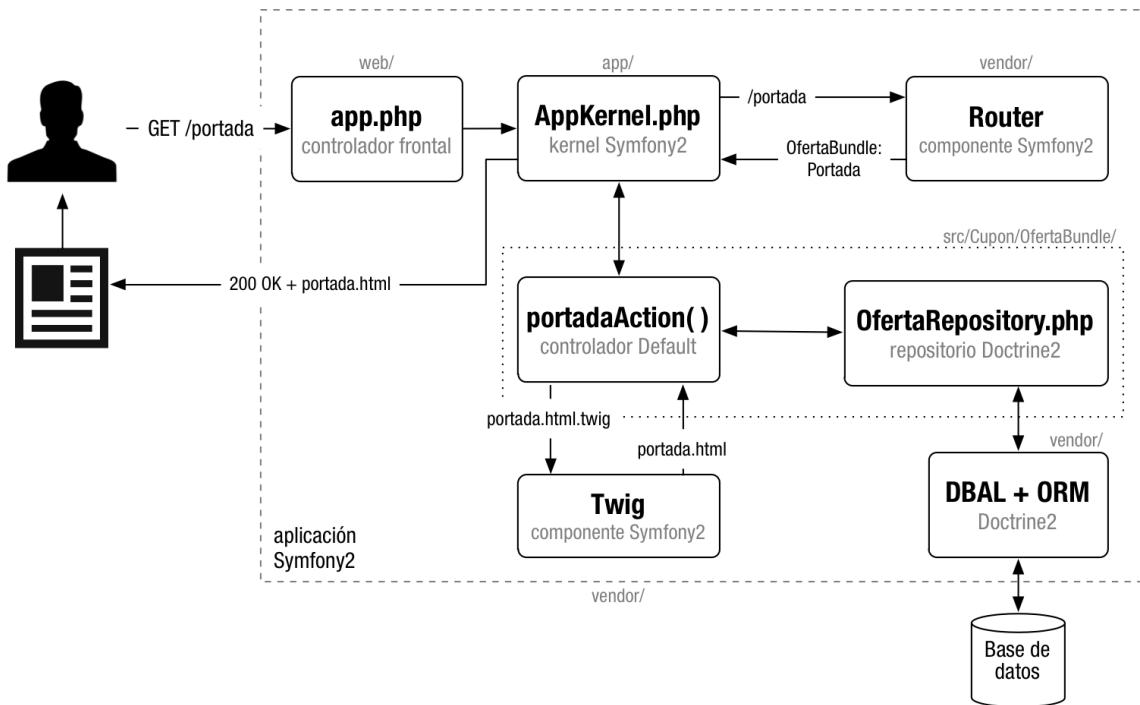


Figura 6.6 Esquema del funcionamiento interno de Symfony2

El usuario solicita la portada del sitio a través de la URL <http://cupon.local/>. 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* activos 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` llenándolo 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 Doctrine2.

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

Symfony2 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 mediante el método `getRequest()`:

```
class DefaultController extends Controller
{
    public function portadaAction()
    {
        $peticion = $this->getRequest();
    }
}
```

El código anterior sólo funciona para los controladores por defecto de Symfony2, es decir, para aquellos que heredan de la clase `Controller`. Este controlador base añade varios atajos útiles para las tareas más comunes. Si tu controlador no hereda de `Controller`, puedes obtener el objeto a través del servicio `request` del contenedor de inyección de dependencias: `$this->container->get('request');`

En primer lugar, `Request` define varias propiedades públicas con la misma información que las variables globales de PHP, pero con un acceso más cómodo:

Propiedad de <code>Request</code>	Tipo de variable	Equivalente de PHP
<code>\$query</code>	<code>ParameterBag</code>	variable global <code>\$_GET</code>
<code>\$request</code>	<code>ParameterBag</code>	variable global <code>\$_POST</code>
<code>\$attributes</code>	<code>ParameterBag</code>	parámetros incluidos en <code>\$_SERVER['PATH_INFO']</code>
<code>\$cookies</code>	<code>ParameterBag</code>	variable global <code>\$_COOKIE</code>
<code>\$files</code>	<code>FileBag</code>	variable global <code>\$_FILES</code>
<code>\$server</code>	<code>ServerBag</code>	variable global <code>\$_SERVER</code>
<code>\$headers</code>	<code>HeaderBag</code>	cabeceras incluidas en la variable global <code>\$_SERVER</code>

Las variables de tipo `*Bag` son contenedores que almacenan pares clave/valor y que incluyen varios métodos útiles:

Método de <code>ParameterBag</code>	Explicación
<code>get(\$clave, \$defecto = null)</code>	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
<code>has(\$clave)</code>	Devuelve <code>true</code> si existe el parámetro cuya clave se indica como argumento. Devuelve <code>false</code> en cualquier otro caso

Método de ParameterBag	Explicación
<code>set(\$clave, \$valor)</code>	Modifica o añade el parámetro indicado mediante su clave y valor
<code>add(\$array)</code>	Añade los parámetros pasados como array, reemplazando los valores de los parámetros existentes
<code>all()</code>	Devuelve un array con todos los parámetros
<code>keys()</code>	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 objeto de la petición en un controlador
$peticion = $this->getRequest();

// Obtener el valor del parámetro GET llamado 'ciudad'
// $ciudad valdrá 'null' si no existe el parámetro 'ciudad'
$ciudad = $peticion->query->get('ciudad');

// Mismo ejemplo, pero asignando un valor por defecto por si
// no existe el parámetro de tipo GET llamado 'ciudad'
$ciudad = $peticion->query->get('ciudad', 'paris');

// Obtener el valor del parámetro POST llamado 'ciudad'
// $ciudad valdrá 'null' si no existe el parámetro 'ciudad'
$ciudad = $peticion->request->get('ciudad');

// Mismo ejemplo, pero asignando un valor por defecto por si
// no existe el parámetro de tipo POST llamado 'ciudad'
$ciudad = $peticion->request->get('ciudad', 'paris');

// Saber qué navegador utiliza el usuario mediante la cabecera HTTP_USER_AGENT
$navegador = $peticion->server->get('HTTP_USER_AGENT');

// Mismo ejemplo, pero más fácil directamente a través de las cabeceras
$navegador = $peticion->headers->get('user-agent');

// Obtener el nombre de todas las cabeceras enviadas
$cabeceras = $peticion->headers->keys();

// Saber si se ha enviado una cookie de sesión
$hayCookieSesion = $peticion->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`:

```
$peticion = $this->getRequest();
$ciudad = $peticion->query->get('ciudad');
$ciudad = $peticion->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://cupon.local/app_dev.php/sitio/ayuda?parametro=valor`:

Método	Explicación
<code>getMethod()</code>	Devuelve el nombre en mayúsculas del método utilizado para la petición (<code>GET</code> , <code>POST</code> , <code>PUT</code> , <code>DELETE</code>). Ejemplo: <code>GET</code>
<code>getRequestUri()</code>	Devuelve la URI de la petición. Ejemplo: <code>/app_dev.php/ayuda?parametro=valor</code>
<code>getUri()</code>	Devuelve la URI <i>normalizada</i> (con todas sus partes). Ejemplo: <code>http://cupon.local/app_dev.php/ayuda?parametro=valor</code>
<code>getScheme()</code>	Devuelve el esquema utilizado (<code>http</code> o <code>https</code>)
<code>getHost()</code>	Devuelve el <i>host</i> de la URI. Ejemplo: <code>cupon.local</code>
<code>getHttpHost()</code>	Igual que <code>getHost()</code> , pero también incluye el puerto si no es <code>80</code> o <code>443</code> . Ejemplo: <code>cupon.local</code>
<code>getPort()</code>	Devuelve el puerto al que se realiza la petición. Ejemplo: <code>80</code>
<code>getQueryString()</code>	Devuelve la <i>query string</i> normalizada de la URI (los parámetros se ordenan alfabéticamente y se <i>escapan</i> los caracteres problemáticos). Ejemplo: <code>parametro=valor</code>
<code>getScriptName()</code>	Devuelve el nombre del <i>script</i> PHP que se ejecuta. Ejemplo: <code>/app_dev.php</code> (incluye la barra <code>/</code> por delante)
<code>getPathInfo()</code>	Devuelve la parte de la ruta en relación al script que se ejecuta. Ejemplo: <code>/ayuda</code> (siempre incluye la barra <code>/</code> por delante)
<code>getBasePath()</code>	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 <code>http://cupon.local/frontend/app_dev.php/ayuda</code> , se devuelve <code>/frontend</code> (siempre incluye la barra <code>/</code> por delante)
<code>getBaseUrl()</code>	Igual que <code>getBasePath()</code> pero también incluye el nombre del script que se ejecuta. Ejemplo: <code>/app_dev.php</code> (nunca se incluye la barra <code>/</code> al final)

Además de los métodos anteriores, la clase `Request` define otros atajos útiles para obtener información de la petición:

Método	Explicación
<code>getClientIp()</code>	Devuelve la dirección IP del usuario que ha realizado la petición. Si el usuario se encuentra detrás de un <i>proxy</i> , busca su dirección en los parámetros <code>HTTP_CLIENT_IP</code> o <code>HTTP_X_FORWARDED_FOR</code> en vez de <code>REMOTE_ADDR</code>
<code>getLanguages()</code>	Devuelve un array con el código de los idiomas preferidos por el usuario y ordenados por importancia. Ejemplo: <code>array('en_US', 'en', 'es')</code>
<code>getPreferredLanguage()</code>	Devuelve el idioma preferido por el usuario, que coincide con el primer idioma de <code>getLanguages()</code> . 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 coincide en el array pasado y en el array devuelto por <code>getLanguages()</code> . Si no se producen coincidencias, se devuelve el primer idioma del array pasado.
<code>isXmlHttpRequest()</code>	Devuelve un valor <i>booleano</i> 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
<code>isSecure()</code>	Devuelve un valor <i>booleano</i> 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' ||
        $this->server->get('HTTPS') == 1)
        ||
        ($self::$trustProxy && strtolower($this->headers->get('SSL_HTTPS')) ==
        'on' || $this->headers->get('SSL_HTTPS') == 1)
        ||
        ($self::$trustProxy &&
        strtolower($this->headers->get('X_FORWARDED_PROTO')) == '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
<code>getRequestFormat()</code>	Devuelve el formato de la petición en minúsculas. Ejemplo: <code>html</code>
<code>getMimeType()</code>	Devuelve el tipo MIME asociado al formato de la petición. Ejemplo: <code>text/html</code>

A continuación se muestran los formatos predefinidos por Symfony2 y sus tipos MIME asociados, aunque se pueden definir nuevos formatos fácilmente (como se explica en la sección *registrando nuevos formatos* (página 280) del capítulo 9):

```
static::$formats = array(
    'html' => array('text/html', 'application/xhtml+xml'),
    'txt' => array('text/plain'),
    'js' => array('application/javascript', 'application/x-javascript',
                  'text/javascript'),
    'css' => array('text/css'),
    'json' => array('application/json', 'application/x-json'),
    'xml' => array('text/xml', 'application/xml', 'application/x-xml'),
    'rdf' => array('application/rdf+xml'),
    'atom' => array('application/atom+xml'),
    'rss' => array('application/rss+xml'),
);
```

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 Symfony2 (<http://api.symfony.com/2.3/>) o directamente puedes ver su código fuente en `vendor/symfony/src/Symfony/Component/HttpFoundation/Request.php`.

6.12 El objeto Response

Symfony2 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('OfertaBundle:Default: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 Symfony2. Utilizando el sistema de plantillas definido en la apli-

cación, el método `render()` crea un objeto de tipo `Response` que Symfony2 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 = <<<EOF
<html>
    <head> <title>Portada del sitio</title> </head>
    <body> <h1>Portada</h1> </body>
</html>
EOF;

        $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 = <<<EOF
<html>
    <head> <title>Portada del sitio</title> </head>
    <body> <h1>Portada</h1> </body>
</html>
EOF;

        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 487): `(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 Symfony2, 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 Symfony2 (<http://api.symfony.com/2.3/>) o directamente puedes ver su código fuente en `vendor/symfony/src/Symfony/Component/HttpFoundation/Response.php`.

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:

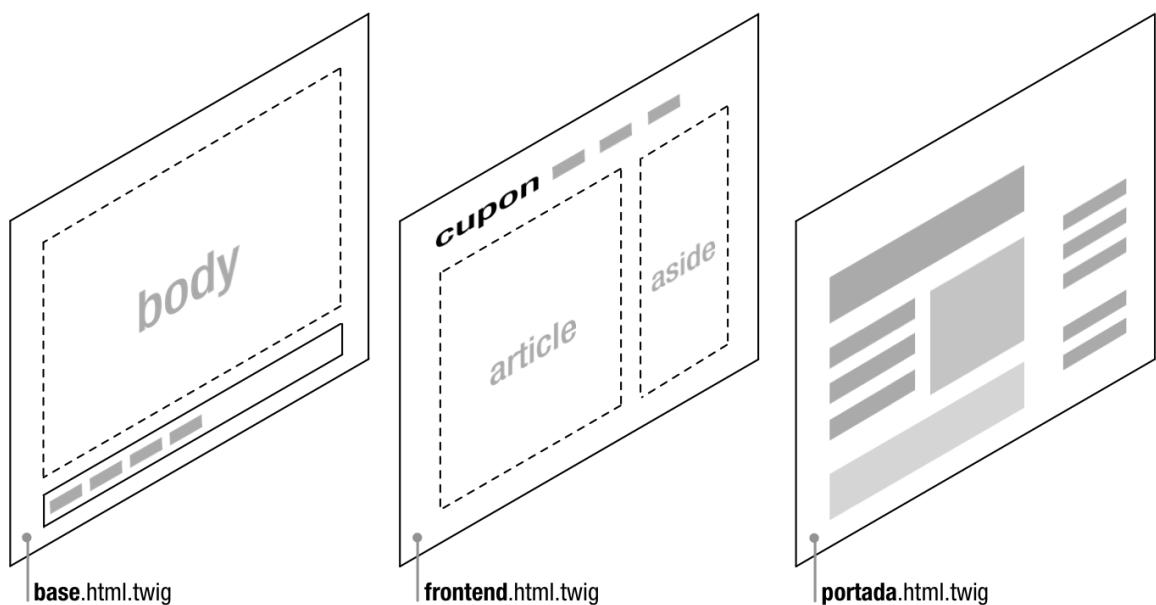


Figura 7.1 Herencia de plantillas a tres niveles utilizada para generar la portada

- **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>

{# src/Cupon/OfertaBundle/Resources/views/Default/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_estatica', { 'pagina': 'ayuda' }) }}">Ayuda</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'contacto' }) }}"
}}>Contacto</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'privacidad' }) }}"
}}>Privacidad</a>
    <a href="{{ path('pagina_estatica', { 'pagina': 'sobre-nosotros' }) }}"
}}>Sobre nosotros</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 Symfony2, 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 Symfony2, 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 dia</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:

```
{# src/Cupon/OfertaBundle/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 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* según la terminología de Symfony2. Para crear buenas aplicaciones web resulta esencial gestionar los archivos web correctamente.

Los estilos de las aplicaciones web reales suelen dividirse en varios archivos CSS para hacerlos más manejables y para reutilizar estilos comunes en las diferentes partes del sitio web. En la aplicación *Cupon* se utilizan cinco archivos CSS:

- `normalizar.css`, se encarga de neutralizar las diferencias de los estilos por defecto de los diferentes navegadores. Esta hoja de estilos también se suele denominar `reset.css`. Su contenido se ha obtenido a partir del archivo CSS base del proyecto *HTML5 Boilerplate* (<https://github.com/h5bp/html5-boilerplate/tree/master/css/style.css>) .
- `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`, recoge los estilos específicos de las páginas que forman la *extranet*.
- `backend.css`, define los estilos propios de las páginas de la parte de administración.

Los archivos `normalizar.css`, `comun.css` y `frontend.css` se incluyen dentro del *bundle OfertaBundle*. El archivo `extranet.css` se guarda en el *bundle TiendaBundle*. Por último, el archivo `backend.css` se incluye en el *bundle BackendBundle*.

Igualmente, los archivos JavaScript también suelen distribuirse por los *bundles* de toda la aplicación. En el caso de la aplicación *Cupon* sólo existe un archivo JavaScript llamado `frontend.js` y que se guarda en el *bundle OfertaBundle*.

7.2.1 Organizando los archivos web de los bundles

Si la hoja de estilos de tu aplicación es muy sencilla, puedes definir un único archivo, guardarlo en el directorio `web/css/` del proyecto y enlazarlo directamente desde todas las plantillas. Lo mismo se puede aplicar a los archivos JavaScript muy sencillos, que se pueden guardar y enlazar directamente desde `web/js`.

No obstante, la buena práctica recomendada para aplicaciones Symfony2 consiste en incluir en cada *bundle* todos sus archivos web (CSS, JavaScript e imágenes). Aunque puede parecer que así la aplicación acaba teniendo decenas de archivos CSS y JavaScript, más adelante se explica cómo Symfony combina automáticamente todos los archivos en uno solo para mejorar el rendimiento.

Por convención, los archivos CSS del *bundle* se guardan en su directorio `Resources/public/css/`, los archivos JavaScript en `Resources/public/js/` y las imágenes en `Resources/public/images/`. Inicialmente esos directorios no existen, por lo que debes crearlos a mano.

Por el momento sólo se necesitan los archivos CSS, así que crea el directorio `Resources/public/css/`, añade los siguientes archivos y copia los estilos de los archivos que se indican a continuación:

- `src/Cupon/OfertaBundle/Resources/public/css/normalizar.css`, copiar los estilos de <https://github.com/javierregiluz/Cupon/blob/master/src/Cupon/OfertaBundle/Resources/public/css/normalizar.css>
- `src/Cupon/OfertaBundle/Resources/public/css/comun.css`, copiar los estilos de <https://github.com/javierregiluz/Cupon/blob/master/src/Cupon/OfertaBundle/Resources/public/css/comun.css>
- `src/Cupon/OfertaBundle/Resources/public/css/frontend.css`, copiar los estilos de <https://github.com/javierregiluz/Cupon/blob/master/src/Cupon/OfertaBundle/Resources/public/css/frontend.css>

7.2.2 Enlazando los archivos web desde las plantillas

Para que la aplicación sea lo más flexible posible, no enlaces a mano los archivos CSS, JavaScript e imágenes. En su lugar, utiliza siempre la función `asset()` de Twig, que tiene en cuenta la configuración actual del proyecto para generar las URL adecuadas.

Si utilizas un único archivo CSS creado en el directorio `web/css/`, basta con indicar lo siguiente:

```
<link href="{{ asset('css/estilos.css') }}" rel="stylesheet" type="text/css" />
```

Sin embargo, como los archivos CSS de las aplicaciones Symfony2 se encuentran dentro de los *bundles*, se enlazan de la siguiente manera:

```
<link href="{{ asset('bundles/oferta/css/comun.css') }}" rel="stylesheet"
      type="text/css" />
```

Con todo lo anterior, abre la plantilla `frontend.html.twig` e incluye el siguiente código para enlazar las tres hojas de estilo de las páginas del *frontend*:

```
{# app/Resources/views/frontend.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    <link href="{{ asset('bundles/oferta/css/normalizar.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/oferta/css/comun.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/oferta/css/frontend.css') }}"
          rel="stylesheet" type="text/css" />
{% endblock %}

{# ... #}
```

Guarda los cambios y después accede a la portada del sitio (<http://cupon.local/>). No verás ninguna diferencia, ni en el entorno de desarrollo ni en el de producción. Si observas el código HTML de la página, verás que la URL del archivo `comun.css` es `/bundles/oferta/css/comun.css`.

Como el directorio raíz público del proyecto es `web/`, la URL anterior corresponde al directorio `web/bundles/oferta/css/comun.css` del proyecto. Obviamente ese directorio no existe, ya que los CSS se guardan en los directorios de cada *bundle*, no en el directorio `web/`.

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 Symfony\Bundle\FrameworkBundle into web/bundles/framework
Installing assets for Cupon\OfertaBundle into web/bundles/oferta
Installing assets for Symfony\Bundle\WebProfilerBundle into web/bundles/
webprofiler
Installing assets for Sensio\Bundle\DistributionBundle into web/bundles/
sensiodistribution
```

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/algún/directorio
```

El comando `assets:install` no solo instala los archivos web de tus *bundles*, sino que también instala cualquier archivo web de los *bundles* internos de Symfony2.

La instalación de los archivos simplemente 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 Versionando los archivos web

Para mejorar el rendimiento del sitio web, muchas aplicaciones suelen marcar los archivos web como elementos *cacheables* por los navegadores. Como el tiempo en la caché suele ser un período de tiempo muy largo, algunas aplicaciones añaden un parámetro a las URL de todos los archivos web. El valor de este parámetro no afecta en nada al archivo web, pero cuando se modifica su valor, la URL ya no es la misma y por tanto, se fuerza a que el navegador vuelva a solicitar el archivo y no utilice el de la caché.

Symfony2 soporta esta característica a través de la opción `assets_version`:

```
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'], assets_version: 1.3 }
```

Con la configuración anterior, las URL de todos los archivos web enlazados con la función `asset()` incluirán al final `?1.3`, como si fuera una *query string*:

```
<!-- Antes -->
<link href="/bundles/oferta/css/normalizar.css" rel="stylesheet" type="text/
css" />
...


<!-- Ahora -->
<link href="/bundles/oferta/css/normalizar.css?1.3" rel="stylesheet" type="text/
css" />
...

```

La opción `assets_version` permite elegir libremente el formato de versiones que prefieras. Ahora, cuando los archivos web cambien, sólo debes indicar en la opción `assets_version` cualquier otro valor que no sea `1.3` y así los navegadores se verán forzados a solicitar de nuevo todos los archivos web.

NOTA En las versiones anteriores a Symfony 2.3 era posible ejecutar código PHP dentro de un archivo YAML, por lo que el valor de la opción `assets_version` se podía generar dinámicamente:

```
# app/config/config.yml
framework:
    # ...
    templating: { ... , assets_version: <?php echo date('Ymd') ?> }
```

Symfony 2.3 eliminó esta característica por motivos de seguridad.

Por último, también puedes modificar el formato con el que se indica la versión del archivo web mediante la opción `assets_version_format`:

```
# app/config/config.yml
framework:
    # ...
    templating: { ... , assets_version_format: %%s?v=%%s }
```

Con esta nueva configuración, la URL de los archivos web tendrá el siguiente aspecto: `/uploads/images/foto1.jpg?v=1.3`. El valor de `assets_version_format` puede ser cualquier cadena de texto compatible con la función `sprintf()`. El carácter `%` debe doblarse siempre que aparezca, para no interferir con otras configuraciones. Para formatear la cadena de la opción

`assets_version_format` se pasa como primer argumento la ruta original del archivo y como segundo parámetro su versión.

7.3 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 nueva ruta dentro del *bundle CiudadBundle*.

El patrón de la ruta es `/ciudad/cambiar-a-{ciudad}` y su acción asociada es `cambiarAction()` dentro del controlador por defecto del *bundle CiudadBundle*:

```
# src/Cupon/CiudadBundle/Resources/config/routing.yml
ciudad_cambiar:
    path:      /ciudad/cambiar-a-{ciudad}
    defaults: { _controller: CiudadBundle:Default:cambiar }
```

El código del controlador es muy sencillo, ya que sólo debe obtener el valor de la variable `$ciudad` que se pasa desde el enrutamiento y redirigir al usuario a la portada de esa ciudad:

```
// src/Cupon/CiudadBundle/Controller/DefaultController.php
namespace Cupon\CiudadBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse;

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

```
{
    return new RedirectResponse($this->generateUrl(
        'portada',
        array('ciudad' => $ciudad)
    ));
}
}
```

Accede ahora por ejemplo a la URL http://cupon.local/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 el valor de la propiedad `pattern` de la ruta `ciudad_cambiar`:

```
ciudad_cambiar:
    pattern: /ciudad/cambiar-a-{ciudad}
    # ...
```

Observa que el patrón no sigue el formato básico habitual (`/ciudad/cambiar/{ciudad}`) sino que una de sus partes mezcla un contenido estático `cambiar-a-` con una parte variable `{ciudad}` que puede contener cualquier cosa. Para evitar problemas, cuando Symfony procesa un patrón de este tipo, 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. Las rutas pueden definir una opción llamada `requirements` para añadir condiciones que se deben cumplir para que la ruta se tenga en cuenta. Si quieres por ejemplo que la ruta `ciudad_cambiar` sólo funcione para el método `GET` de HTTP, añade lo siguiente:

```
ciudad_cambiar:
    pattern: /ciudad/cambiar-a-{ciudad}
    defaults: { _controller: CiudadBundle:Default:cambiar }
    requirements:
        methods: [GET]
```

Las condiciones definidas con `requirements` también se pueden aplicar a las partes variables de las rutas mediante expresiones regulares. Así por ejemplo, si quieres que esta ruta sólo funcione cuando la ciudad sea Madrid o Barcelona, utiliza la siguiente condición:

```
ciudad_cambiar:
    pattern: /ciudad/cambiar-a-{ciudad}
    defaults: { _controller: CiudadBundle:Default:cambiar }
    requirements:
        ciudad: madrid|barcelona
```

Con todo lo anterior, la solución al problema de que las ciudades no puedan contener guiones medios en su nombre consiste en indicar que el nombre de una ciudad puede contener cualquier carácter. Modifica la ruta `ciudad_cambiar` por lo siguiente, recarga la página y verás que el error ha desaparecido:

```
ciudad_cambiar:
    pattern:  /ciudad/cambiar-a-{ciudad}
    defaults: { _controller: CiudadBundle:Default:cambiar }
    requirements:
        ciudad: .+
```

El siguiente paso consiste en crear la lista desplegable que muestra a los usuarios del sitio web todas las ciudades disponibles en la aplicación. Si suponemos que la lista de ciudades es muy larga y evoluciona con el tiempo, no es recomendable definir el listado como una opción de configuración de la aplicación. Resulta mucho más conveniente hacer una consulta a la base de datos para obtener todas las ciudades.

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><a href="#">Ofertas recientes</a></li>
            <li><a href="#">Mis ofertas</a></li>

            <li>{{ render(controller('CiudadBundle:Default:listaCiudades')) }}
```

La instrucción `render()` hace que en ese punto de la plantilla se inserte el resultado de la acción `listaCiudadesAction()` del controlador `DefaultController.php` del *bundle* `CiudadBundle`. Así que en ese punto se incluirá el código de la plantilla asociada a la acción mencionada.

La acción `listaCiudadesAction()` es realmente sencilla, ya que sólo debe obtener el listado de todas las ciudades de la aplicación:

```
// src/Cupon/CiudadBundle/Controller/DefaultController.php
namespace Cupon\CiudadBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse;

class DefaultController extends Controller
{
    // ...

    public function listaCiudadesAction()
    {
        $em = $this->getDoctrine()->getManager();
        $ciudades = $em->getRepository('CiudadBundle:Ciudad')->findAll();

        return $this->render(
            'CiudadBundle:Default:listaCiudades.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:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/listaCiudades.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`:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/listaCiudades.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://cupon.local/app_dev.php/ciudad/cambiar-a-alicante">
        Alicante
    </option>

    <option value="badalona"
           data-url="http://cupon.local/app_dev.php/ciudad/cambiar-a-badalona">
        Badalona
    </option>

    <option value="barcelona"
           data-url="http://cupon.local/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('CiudadBundle:Default:listaCiudades', {
    'ciudad': ciudadSeleccionada
})) }}</li>
```

Después, recoge la variable en la acción y pásala a la plantilla que muestra la lista desplegable:

```
// src/Cupon/CiudadBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    // ...

    public function listaCiudadesAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();
        $ciudades = $em->getRepository('CiudadBundle:Ciudad')->findAll();

        return $this->render(
            'CiudadBundle:Default:listaCiudades.html.twig',
            array(
                'ciudadActual' => $ciudad,
                'ciudades'      => $ciudades
            )
        );
    }
}
```

Por último, haz que la lista muestre seleccionado el elemento que coincide con la ciudad actual:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/listaCiudades.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="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 quieras 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_defecto) %}
```

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 Symfony2 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:
    cupon.ciudad_por_defecto: 'barcelona'
```

Solucionar esta duplicidad es muy sencillo, ya que Symfony2 permite hacer referencia al valor de cualquier parámetro encerrando su nombre entre `%` y `%`:

```
# app/config/config.yml

twig:
    # ...
globals:
    ciudad_por_defecto: %cupon.ciudad_por_defecto%

# ...

parameters:
    cupon.ciudad_por_defecto: 'barcelona'
```

7.4 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.

El primer elemento que hay que definir para crear la página es su ruta:

```
# app/config/routing.yml

oferta:
    path:      /{ciudad}/ofertas/{slug}
    defaults: { _controller: OfertaBundle:Default:oferta }
```

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 }) }}
```

Para mostrar el detalle de una oferta, define una nueva acción llamada `ofertaAction()` dentro del controlador por defecto del *bundle OfertaBundle*:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
namespace Cupon\OfertaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse;

class DefaultController extends Controller
{
    // ...

    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('OfertaBundle:Oferta')
            ->findOferta($ciudad, $slug);

        return $this->render('OfertaBundle:Default:detalle.html.twig', array(
            'oferta' => $oferta
        ));
    }
}
```

La lógica de la acción es muy sencilla, ya que primero busca la oferta en la base de datos y después, pasa sus datos a la plantilla asociada con la acción. 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 Doctrine2, añádelo en el repositorio de la entidad `Oferta`:

```
// src/Cupon/OfertaBundle/Entity/OfertaRepository.php
namespace Cupon\OfertaBundle\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 OfertaBundle: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 `detalle.html.twig`. Como esta página es parecida a la portada, puedes copiar su código y actualizar los bloques `title` e `id`:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/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. Además, gracias a Twig también es una práctica innecesaria. Cuando dos o más plantillas comparten código, se extrae el código común en una nueva plantilla. Después, se incluye el contenido de esta plantilla en cualquier otra mediante la instrucción `{% include %}`.

Para organizar mejor las plantillas y para demostrar la flexibilidad de Symfony2, se van a guardar todos los trozos comunes en un nuevo directorio llamado `includes/` dentro del directorio de plantillas del *bundle* `OfertaBundle`.

Crea el directorio `src/Cupon/OfertaBundle/Resources/views/Default/includes/` y en su interior, crea un archivo llamado `oferta.html.twig` con todo el código común de la portada y de la página de detalle:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/includes/oferta.html.twig #}
<section class="descripcion">
    <h1><a href="{{ path('oferta', { 'ciudad': oferta.ciudad.slug, 'slug' :
oferta.slug }) }}>{{ oferta.nombre }}</a></h1>

    {{ oferta.descripcion }}

```

```

<a class="boton" href="#">Comprar</a>
</section>

<section class="galeria">
    {{ oferta.precio }} &euro; <span>{{ oferta.descuento }}</span></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 `detalle.html.twig`:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/detalle.html.twig #-}
{% extends '::frontend.html.twig' %}

{% block title %}Detalles de {{ oferta.nombre }}{% endblock %}
{% block id 'oferta' %}

{% block article %}
    {% include 'OfertaBundle:Default:includes/oferta.html.twig' %}
{% endblock %}

{% block aside %}
    {# ... #-}
{% endblock %}

```

La instrucción `{% include %}` de Twig inserta en ese mismo punto el código de la plantilla indicada mediante la *notación bundle*. En este caso, como la plantilla se encuentra en un subdirectorío de `Resources/views/Default/`, el nombre de la plantilla también incluye ese subdirectorío.

La plantilla de la portada también se puede simplificar haciendo uso de la plantilla común que muestra la información de una oferta:

```
{# src/Cupon/OfertaBundle/Resources/views/Default/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 'OfertaBundle:Default:includes/oferta.html.twig' %}
{% endblock %}

{% block aside %}
    {# ... #}
{% endblock %}
```

7.4.1 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 pasárlas a la plantilla:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    // ...

    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('OfertaBundle:Oferta')
            ->findOferta($ciudad, $slug);
        $relacionadas = $em->getRepository('OfertaBundle:Oferta')
            ->findRelacionadas($ciudad);

        return $this->render('OfertaBundle:Default:detalle.html.twig', array(
            'oferta'      => $oferta,
            'relacionadas' => $relacionadas
        ));
    }
}
```

Como es evidente, el método `findRelacionadas()` para encontrar ofertas relacionadas no está definido por Doctrine2, por lo que se debe incluir en el repositorio propio de la entidad `Oferta`:

```
// src/Cupon/OfertaBundle/Entitiy/OfertaRepository.php
class OfertaRepository extends EntityRepository
{
```

```
// ...

public function findRelacionadas($ciudad)
{
    $em = $this->getEntityManager();

    $consulta = $em->createQuery('
        SELECT o, c
        FROM OfertaBundle:Oferta o
        JOIN o.ciudad c
        WHERE o.revisada = true
            AND o.fecha_publicacion <= :fecha
            AND c.slug != :ciudad
        ORDER BY o.fecha_publicacion 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:

```
{# src/Cupon/OfertaBundle/Resources/views/Default/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', { 'ciudad': 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`).

7.5 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 (``).

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 en Symfony2* (página 554) del apéndice A.

7.5.1 Activando las extensiones opcionales de Twig

Symfony2 incluye varias extensiones oficiales de Twig que no están activadas por defecto. Las dos más útiles son `Text` y `Debug`. La primera incluye los dos siguientes filtros:

- `truncate($longitud = 30, $respetar = false, $separador = '...')`, recorta un contenido de texto para que su longitud sea como máximo la indicada por el parámetro `$longitud`. Si el parámetro `$respetar` es `false`, el texto se corta sin importar si una palabra se deja a medias. Si vale `true`, se respetan las palabras enteras. El parámetro `$separador` es el texto que se añade después del texto recortado para indicar que el texto original era más largo.
- `wordwrap($longitud = 80, $separador = "\n", $respetar = false)`, formatea un contenido de texto en varias líneas, cada una de las cuales tiene como máximo el número de caracteres indicados por el parámetro `$longitud`. Después de cada línea se añade el valor del parámetro `$separador`. Por último, el parámetro `$respetar` indica si se deben respetar las palabras enteras al partir el texto original en varias líneas.

Por su parte, la extensión `Debug` incluye una nueva función llamada `dump()` que permite mostrar el contenido de cualquier variable de la plantilla. Se trata del equivalente de la función `var_dump()` de PHP.

Las extensiones de Twig se activan como cualquier otro servicio del contenedor de inyección de dependencias (explicado con detalle en el apéndice B (página 581)). Abre el archivo `app/config/config.yml` y añade lo siguiente al final del todo:

```
# 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 }
```

Ahora ya puedes hacer uso de las extensiones [Text](#) y [Debug](#) en tus plantillas:

```
{# Truncar a 100 caracteres la descripción de la oferta y respetar palabras #}
{{ oferta.descripcion | truncate(100, true) }}

{# Mostrar la descripción de la tienda en varias líneas cortas #}
{{ oferta.tienda.descripcion | wordwrap(40) | nl2br }}

{# Mostrar todas las propiedades de la variable 'oferta' de la plantilla #}
{{% dump(oferta) %}}
```

Además de las extensiones de Twig, la aplicación *Cupon* define varios servicios adicionales, así que el archivo de configuración global puede acabar siendo demasiado grande. Aprovechando la flexibilidad de los archivos de configuración de Symfony2, crea un nuevo archivo llamado [services.yml](#) e incluye en él toda la configuración de los servicios:

```
# app/config/services.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 }
```

Para que los servicios sigan estando disponibles en la aplicación, no olvides importar el nuevo archivo de configuración mediante la opción [imports](#):

```
# app/config/config.yml
imports:
    - { resource: parameters.ini }
    - { resource: security.yml }
    - { resource: services.yml }

# ...
```

7.5.2 Creando tus propias extensiones

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 ``. El tipo de lista es por defecto ``, 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`. La extensión se define dentro del *bundle* `OfertaBundle` porque los filtros se utilizan mayoritariamente en las plantillas de ese *bundle*.

Para seguir las convenciones utilizadas en el código de Symfony2, crea el directorio `Twig/Extension/` dentro de `src/Cupon/OfertaBundle/`. En su interior, añade un archivo llamado `CuponExtension.php` con el siguiente contenido:

```
# src/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php
namespace Cupon\OfertaBundle\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:
    # ...
    twig.extension.cupon:
        class: Cupon\OfertaBundle\Twig\Extension\CuponExtension
        tags:
            - { name: twig.extension }
```

Ahora ya puedes añadir la primera función `descuento()` en la extensión:

```
# src/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    public function getFunctions()
    {
```

```

        return array(
            'descuento' => new \Twig_Function_Method($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:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/includes/oferta.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/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    // ...

```

```

public function getFilters()
{
    return array(
        'mostrar_como_lista' => new \Twig_Filter_Method($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:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/includes/oferta.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/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;

class CuponExtension extends \Twig_Extension
{
    // ...

    public function getFilters()
    {
        return array(
            // ...
            'cuenta_atras' => new \Twig_Filter_Method($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()) /
1000 );

    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> ' +
cuentaAtras;
}

muestraCuentaAtras();
</script>
EOJ;

return $html;
}
}

```

Ahora ya puedes mostrar cualquier fecha futura como una cuenta atrás del tiempo restante:

```

{# src/Cupon/OfertaBundle/Resources/views/Default/includes/oferta.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.

7.6 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. Como la página está relacionada con las ciudades, todo se incluirá dentro del *bundle CiudadBundle*.

Abre el archivo de enrutamiento de ese *bundle* y añade la siguiente ruta:

```
# src/Cupon/CiudadBundle/Resources/config/routing.yml
ciudad_recientes:
    path:      /{ciudad}/recientes
    defaults: { _controller: CiudadBundle:Default:recientes }
```

Después, añade la acción `recientesAction()` dentro del controlador por defecto del *bundle* y copia el siguiente código:

```
// src/Cupon/CiudadBundle/Controller/DefaultController.php
namespace Cupon\CiudadBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    public function recientesAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();

        $ciudad = $em->getRepository('CiudadBundle:Ciudad')
            ->findOneBySlug($ciudad);
        $cercanas = $em->getRepository('CiudadBundle:Ciudad')
            ->findCercanas($ciudad->getId());
        $ofertas = $em->getRepository('OfertaBundle:Oferta')
            ->findRecientes($ciudad->getId());
    }
}
```

```

        return $this->render('CiudadBundle:Default: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 Doctrine2, 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/Cupon/CiudadBundle/Entity/Ciudad.php
/**
 * @ORM\Entity(repositoryClass="Cupon\CiudadBundle\Entity\CiudadRepository")
 */
class Ciudad
{
    // ...
}

```

Después, crea el archivo `CiudadRepository` y añade el código de la consulta `findCercanas()`:

```

// src/Cupon/CiudadBundle/Entity/CiudadRepository.php
namespace Cupon\CiudadBundle\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 cualesquier 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/Cupon/OfertaBundle/Entity/OfertaRepository.php
namespace Cupon\OfertaBundle\Entity;
use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository
{
    // ...

    public function findRecientes($ciudad_id)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t
            FROM OfertaBundle:Oferta o
            JOIN o.tienda t
            WHERE o.revisada = true
            AND o.fecha_publicacion < :fecha
            AND o.ciudad = :id
            ORDER BY o.fecha_publicacion 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 `recientes.html.twig` para mostrar sus resultados:

```

{% src/Cupon/CiudadBundle/Resources/views/Default/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.html.twig`, se define una plantilla llamada `minioferta.html.twig` en ese mismo directorio. Las dos plantillas son similares, pero `minioferta.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:

```

#{ src/Cupon/OfertaBundle/Resources/views/Default/includes/minioferta.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">
        

        <p class="precio">{{ oferta.precio }} &euro; <span>{{ descuento(oferta.precio, 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:

```

{# src/Cupon/CiudadBundle/Resources/views/Default/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 'OfertaBundle:Default:includes/minioferta.html.twig' %}
{% else %}
    <p>Esta ciudad todavía no ha publicado ninguna oferta</p>
{% endfor %}
{% endblock %}

{% block aside %}
{# ... #}
{% endblock %}
```

7.7 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 abre el archivo de enrutamiento del *bundle TiendaBundle* y añade la siguiente ruta:

```
# src/Cupon/TiendaBundle/Resources/config/routing.yml
tienda_portada:
    path:      /{ciudad}/tiendas/{tienda}
    defaults: { _controller: TiendaBundle:Default:portada }
```

La URL de la ruta incluye el *slug* de la ciudad a la que pertenece la tienda y el *slug* de la propia tienda. La acción asociada a la ruta se llama `portadaAction()` y se define en el controlador por defecto del *bundle*:

```
// src/Cupon/TiendaBundle/Controller/DefaultController.php
namespace Cupon\TiendaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class DefaultController extends Controller
{
    public function portadaAction($ciudad, $tienda)
    {
        $em = $this->getDoctrine()->getManager();

        $ciudad = $em->getRepository('CiudadBundle:Ciudad')
            ->findOneBySlug($ciudad);

        $tienda = $em->getRepository('TiendaBundle:Tienda')->findOneBy(array(
            'slug' => $tienda,
            'ciudad' => $ciudad->getId()
        ));

        if (!$tienda) {
            throw $this->createNotFoundException('No existe esta tienda');
        }

        $ofertas = $em->getRepository('TiendaBundle:Tienda')
            ->findUltimasOfertasPublicadas($tienda->getId());

        $cercanas = $em->getRepository('TiendaBundle:Tienda')->findCercanas(
            $tienda->getSlug(),
            $tienda->getCiudad()->getSlug()
        );

        return $this->render('TiendaBundle:Default:portada.html.twig', array(
            'tienda' => $tienda,
            'ofertas' => $ofertas,
            'cercanas' => $cercanas
        ));
    }
}
```

Como es habitual, el código del controlador es muy conciso, ya que sólo pide información al modelo (los repositorios de Doctrine2) y después la pasa 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/Cupon/TiendaBundle/Entity/Tienda.php
namespace Cupon\TiendaBundle\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Cupon\TiendaBundle\Entity\TiendaRepository")
 */
class Tienda
{
    // ...
}
```

Después, crea la clase del repositorio y copia el siguiente código para definir los dos métodos necesarios:

```
// src/Cupon/TiendaBundle/Entity/TiendaRepository.php
namespace Cupon\TiendaBundle\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 OfertaBundle:Oferta o
             JOIN o.tienda t
             WHERE o.revisada = true
               AND o.fecha_publicacion < :fecha
               AND o.tienda = :id
             ORDER BY o.fecha_expiracion 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 TiendaBundle: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 `portada.html.twig` y añade el siguiente código:

```

{# src/Cupon/TiendaBundle/Resources/views/Default/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, 'slug': 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>

```

```

        {% 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.slug, 'tienda': tienda.slug}) }}">{{ tienda.nombre }}</a></li>
        {% endfor %}
    </ul>
</section>
{% endblock %}

```

7.8 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.

7.8.1 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 `minioferta.html.twig`:

```

{{-- src/Cupon/OfertaBundle/Resources/views/Default/includes/oferta.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.slug }) }}">{{ 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 }}</strong> compras<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 `minioferta.html.twig` se puede utilizar la misma estrategia para mostrar información diferente según la oferta haya expirado o no:

```
{# src/Cupon/OfertaBundle/Resources/views/Default/includes/minioferta.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': oferta.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>
```

7.8.2 Valores configurables

Resulta muy poco recomendable que el código incluya valores que deberían definirse en archivos de configuración. Esta recomendación es todavía más importante cuando se repiten los mismos valores en varias clases o plantillas diferentes.

Si observas el código de las plantillas desarrolladas anteriormente, cuando se muestra la foto de una oferta, se utiliza la siguiente expresión:

```

```

7.8.3 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/Cupon/OfertaBundle/Controller/DefaultController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundException;

class DefaultController extends Controller
{
    // ...

    public function ofertaAction($ciudad, $slug)
    {
        $em = $this->getDoctrine()->getManager();
        $oferta = $em->getRepository('OfertaBundle:Oferta')
            ->findOferta($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/Cupon/CiudadBundle/Controller/DefaultController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundException;
```

```

class DefaultController extends Controller
{
    public function recientesAction($ciudad)
    {
        $em = $this->getDoctrine()->getManager();
        $ciudad = $em->getRepository('CiudadBundle:Ciudad')
            ->findOneBySlug($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/Cupon/TiendaBundle/Controller/DefaultController.php
// ...
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class DefaultController extends Controller
{
    public function portadaAction($ciudad, $tienda)
    {
        // ...

        $tienda = $em->getRepository('TiendaBundle: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:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/recientes.html.twig #}
```

```
{# ... #}

{% for oferta in ofertas %}
    {% include 'OfertaBundle:Default:includes/miniofertha.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:

```
{# src/Cupon/TiendaBundle/Resources/views/Default/portada.html.twig #-}

{# ... #}

{% for oferta in ofertas %}
<tr>
    <td>{{ oferta.fechaPublicacion | date() }}</td>
    <td><a href="{{ path('oferta', { 'ciudad': oferta.ciudad.slug, 'slug': oferta.slug }) }}">{{ 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 %}
```

Esta página se ha dejado vacía a propósito

CAPÍTULO 8

Registrando usuarios

La mayoría de aplicaciones web reales disponen de zonas restringidas a las que sólo pueden acceder los usuarios registrados y *logueados*. A pesar de ello, symfony 1 no incluía esta funcionalidad, por lo que era necesario instalar el plugin [sfGuardPlugin](#) (o su versión para Doctrine [sfGuardDoctrinePlugin](#)).

Afortunadamente, el componente de seguridad de Symfony2 es la parte más mejorada respecto a symfony 1. Sin necesidad de instalar ningún *bundle* externo, Symfony2 permite la autenticación de usuarios (con contraseña, con certificados X.509, etc.), la autorización segmentada del sitio web, la opción *remember me* en los formularios de *login*, las listas de control de acceso o ACL, la integración de los usuarios con las entidades de Doctrine2, etc.

En el capítulo anterior se desarrollaron todas las páginas públicas del *frontend*. 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 esta página está relacionada con los usuarios, abre el archivo de enrutamiento del *bundle UsuarioBundle* y añade la siguiente ruta:

```
# src/Cupon/UsuarioBundle/Resources/config/routing.yml
usuario_compras:
    path:      /compras
    defaults: { _controller: UsuarioBundle:Default:compras }
```

Observa que el patrón de la ruta anterior no es [/usuario/compras](#) sino solamente [/compras](#). Sin embargo, durante la planificación del enrutamiento del proyecto (página 42) en el capítulo 2, se decidió que todas las rutas relacionadas con los usuarios comienzan por [/usuario](#). Para no repetir la misma cadena en todas las URL, es mejor utilizar la opción [prefix](#) al importar las rutas del *bundle* desde el archivo de enrutamiento global:

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

```
resource: "@UsuarioBundle/Resources/config/routing.yml"
prefix: /usuario
```

La configuración anterior hace que Symfony2 pefije automáticamente la cadena `/usuario` a todas las rutas definidas en el archivo de enrutamiento del *bundle UsuarioBundle*. Por tanto, la URL de la página que muestra las compras recientes de un usuario es `/usuario/compras` y no solamente `/compras`.

La acción asociada a la ruta es `comprasAction()` y simplemente debe buscar las últimas compras del usuario *logueado* en la aplicación y pasar la información a la plantilla correspondiente:

```
// src/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    public function comprasAction()
    {
        $usuario_id = 1;

        $em = $this->getDoctrine()->getManager();
        $compras = $em->getRepository('UsuarioBundle:Usuario')
            ->findTodasLasCompras($usuario_id);

        return $this->render('UsuarioBundle:Default:compras.html.twig', array(
            'compras' => $compras
        ));
    }
}
```

Como todavía no es posible *loguear* usuarios en la aplicación, el código anterior utiliza siempre el mismo usuario y establece directamente el valor de su atributo `id`. En las próximas secciones se explica cómo obtener el usuario *logueado* y por tanto, se refactorizará el código anterior.

Aunque la consulta a la base de datos es muy sencilla, se emplea un método propio para seguir así la buena práctica de crear repositorios que permitan reutilizar las consultas. Por tanto, modifica el código de la entidad `Usuario` para indicar que utiliza un repositorio propio:

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php
// ...
/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @ORM\Entity(repositoryClass="Cupon\UsuarioBundle\Entity\UsuarioRepository")
 */
class Usuario
{
```

```
// ...
}
```

A continuación, crea el repositorio `UsuarioRepository` y añade el código de la consulta `findTodasLasCompras()`:

```
// src/Cupon/UsuarioBundle/Entity/UsuarioRepository.php
namespace Cupon\UsuarioBundle\Entity;
use Doctrine\ORM\EntityRepository;

class UsuarioRepository extends EntityRepository
{
    public function findTodasLasCompras($usuario)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT v, o, t
            FROM OfertaBundle: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 Doctrine2, también se incluye la información de las ofertas y de las tiendas).

Para finalizar la página, crea la plantilla `compras.html.twig` y añade el siguiente código:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/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 'OfertaBundle:Default:includes/miniofera.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 `minioferta.html.twig`. No obstante, si pruebas la página en el navegador (http://cupon.local/app_dev.php/usuario/compras/) verás un mensaje de error.

Recuerda que al utilizar la etiqueta `{% include %}` la plantilla incluida tiene acceso a todas las variables de la plantilla principal. El problema es que `minioferta.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 definir la variable que necesita la plantilla mediante la palabra reservada `with`. Como los datos de la oferta se obtiene a través de la propiedad `oferta` de la variable `compra`, el código resultante es:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/compras.html.twig #}  
  
{# ... #}  
  
{% for compra in compras %}  
    {% include 'OfertaBundle:Default:includes/minioferta.html.twig'  
        with { '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*. Solúcnalo volviendo a cargar los *fixtures* o cambiando el valor de la variable `$usuario_id` en la acción `comprasAction()`.

Antes de dar por completada esta página, es posible refactorizar ligeramente el código de las plantillas para hacerlo más completo. Aunque la misma plantilla `minioferta.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 `minioferta.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`:

```
{#
src/Cupon/OfertaBundle/Resources/views/Default/includes/minioferta.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):

```
{#
src/Cupon/OfertaBundle/Resources/views/Default/includes/minioferta.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:

```
{#
src/Cupon/OfertaBundle/Resources/views/Default/includes/minioferta.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 a la expresión `with` de la etiqueta `{% include %}`:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/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 'OfertaBundle:Default:includes/minioferta.html.twig' with {
            '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 seguridad de las aplicaciones Symfony2 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:      ^/*
            anonymous:   ~
            form_login:  ~

        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: Cupon\UsuarioBundle\Entity\Usuario, property:
email }

    encoders:
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512, iterations: 10
}

```

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:      ^/*
            anonymous:   ~
            form_login:  ~

        # ...
```

Según la terminología de Symfony2, 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, al margen de la configuración de autorización definida más adelante, 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 cuatro tipos de autenticación disponibles son los siguientes:

```
security:
    firewalls:
        firewall_1:
            # ...
            http_basic:
                provider: nombre-del-proveedor

        firewall_2:
            # ...
            http_digest:
                provider: nombre-del-proveedor

        firewall_3:
```

```
# ...
x509:
    provider: nombre-del-proveedor

firewall_4:
    # ...
    form_login: ~
```

Las opciones `http_basic` y `http_digest` hacen que el navegador muestre la típica caja en la que se solicita usuario y contraseña; la opción `x509` solicita al usuario un certificado de tipo X.509 (como por ejemplo el que contiene el DNI electrónico de España); la opción `form_login` muestra una página con un formulario para introducir el usuario y la contraseña.

Los tres primeros tipos requieren definir una opción llamada `provider`, que indica el nombre del proveedor de usuarios utilizado por el `firewall`, tal y como se explica más adelante.

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:      ^/*
            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 responsable de impedir el acceso a los usuarios normales (ya que es obligatorio que sean de tipo `ROLE_USUARIO`) y el `firewall frontend` es el encargado de mostrar el formulario de *login*, ya que la URL bloqueada se encuentra bajo su responsabilidad y el tipo de autenticación se establece a `form_login`.

El tipo de usuario, denominado *role*, 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 *role* 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 el formulario de *login* se encuentra en `/usuario/login` por lo que 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 *role* especial llamado `IS_AUTHENTICATED_ANONYMOUSLY` que define Symfony2 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 Symfony2 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* los usuarios de la aplicación:

```
# app/config/security.yml
security:
    # ...
    providers:
        usuarios:
            entity: { class: Cupon\UsuarioBundle\Entity\Usuario,
                      property: email }
```

Symfony2 define dos tipos de proveedores: `memory` (los usuarios se crean en memoria con los datos incluidos en el propio archivo `security.yml`) y `entity` (los usuarios se crean mediante entidades de Doctrine2). Por tanto, la configuración anterior indica que los usuarios se crean a partir de la

entidad `Usuario` del *bundle* `UsuarioBundle` y el nombre de usuario será la propiedad `email` de la entidad.

Se pueden definir varios proveedores e incluso encadenarlos en un *super-proveedor*, como muestra el siguiente ejemplo:

```
# app/config/security.yml
security:
    # ...
    providers:
        todos:
            providers: [usuarios, tiendas]
        usuarios:
            entity: { class: Cupon\UsuarioBundle\Entity\Usuario,
                       property: email }
        tiendas:
            entity: { class: Cupon\TiendaBundle\Entity\Tienda,
                       property: login }
```

Si empleas el proveedor llamado `todos`, Symfony2 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`:

```
# app/config/security.yml
security:
    # ...
    encoders:
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512, iterations: 10
    }
```

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:
        Cupon\UsuarioBundle\Entity\Usuario: plaintext
```

El tipo de codificación recomendado para las aplicaciones web reales se llama `sha512`:

```
security:
    # ...
    encoders:
        Cupon\UsuarioBundle\Entity\Usuario: sha512
```

La codificación `sha512` codifica la contraseña 5.000 veces seguidas utilizando cada vez el *hash* SHA de 512 bits del paso anterior. Además, el resultado final se codifica mediante BASE64. En realidad, la configuración anterior es equivalente a lo siguiente:

```
security:  
    # ...  
    encoders:  
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512, iterations:  
            5000, encode_as_base64: true }
```

Como más adelante se van a crear cientos de usuarios de prueba mediante los archivos de datos o *fixtures*, realizar 5.000 iteraciones para cada contraseña provoca una espera considerable. Por ese motivo, en el archivo `security.yml` se indica que sólo se realizarán 10 iteraciones:

```
# app/config/security.yml  
security:  
    # ...  
    encoders:  
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512,  
            iterations: 10 }
```

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 `Cupon\UsuarioBundle\Entity\Usuario` y añade la instrucción `implements` correspondiente (no olvides tampoco importar la interfaz `UserInterface` mediante la instrucción `use`):

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php  
namespace Cupon\UsuarioBundle\Entity;  
  
use Symfony\Component\Security\Core\User\UserInterface;  
use Doctrine\ORM\Mapping as ORM;  
  
/**  
 * Cupon\UsuarioBundle\Entity\Usuario  
 *  
 * @ORM\Entity  
 */  
class Usuario implements UserInterface  
{  
    ...  
}
```

Implementar la interfaz `UserInterface` requiere 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. En la mayoría de aplicaciones, puedes dejar este método vacío sin problemas.
- `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.
- `getUsername()`, se invoca para obtener el *login* o nombre de usuario que se utiliza para autenticar a los usuarios. De esta forma es muy sencillo utilizar cualquier propiedad de la entidad como *login*, como por ejemplo su email.

TRUCO Symfony2 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/Cupon/UsuarioBundle/Entity/Usuario.php
class Usuario implements UserInterface
{
    function eraseCredentials()
    {

    }

    function getRoles()
    {
        return array('ROLE_USUARIO');
    }

    function getUsername()
    {
        return $this->getEmail();
    }

    // ...
}
```

El método `eraseCredentials()` se deja vacío, tal y como se explicó anteriormente. El método `getRoles()` devuelve un array con un único valor (`ROLE_USUARIO`) porque todos los usuarios del *frontend* son del mismo tipo.

```
function getRoles()
{
    return array('ROLE_USUARIO');
}
```

Aunque en las aplicaciones Symfony2 es habitual utilizar los *roles* `ROLE_USER`, `ROLE_ADMIN` y `ROLE_SUPER_ADMIN`, el nombre del *role* 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 *role* 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`, `ROLE_USER_MAX`, etc.

El último método añadido es `getUsername()` que devuelve el valor del nombre de usuario que utilizan los usuarios para hacer el *login*. Los usuarios del *frontend* utilizan su email como nombre de usuario, así que el código simplemente es:

```
function getUsername()
{
    return $this->getEmail();
}
```

De los cinco métodos requeridos por la interfaz solamente se han definido tres. La razón es que el método `getPassword()` ya estaba definido por ser el *getter* de la propiedad `password` de la entidad. Igualmente, la entidad también dispone de una propiedad llamada `salt` y de un *getter* `getSalt()`, por lo que no es necesario volver a definirlo.

TRUCO Cuando definas las entidades de la aplicación, no olvides añadir los atributos `password` y `salt` en todas las entidades que vayas a utilizar para crear usuarios. Así evitarás una refactorización cuando llegues a la configuración de la seguridad y no tendrás que volver a generar las entidades.

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, Symfony2 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 Symfony2 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*.

La única ruta que es obligatorio definir es `/login`, ya que el formulario de *login* y la acción que lo procesa tienes que crearlos manualmente. Las rutas `/login_check` y `/logout` las procesa automáticamente el componente de seguridad de Symfony2, pero es recomendable definirlas para poder incluirlas fácilmente en las plantillas.

Abre el archivo de enrutamiento del *bundle UsuarioBundle* y añade las siguientes rutas:

```
# src/Cupon/UsuarioBundle/Resources/config/routing.yml

# ...

usuario_login:
    path:      /login
    defaults: { _controller: UsuarioBundle:Default:login }

usuario_login_check:
    path:      /login_check

usuario_logout:
    path:      /logout
```

Recuerda que al importar las rutas del *bundle UsuarioBundle*, se les añade el prefijo `/usuario`. Así que en realidad, los patrones completos de las rutas anteriores son: `/usuario/login`, `/usuario/login_check` y `/usuario/logout`.

Symfony2 espera que la ruta de *login* sea `/login`, pero en realidad 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

```

Según la configuración anterior, la ruta `usuario_login` es atendida por el controlador `UsuarioBundle:Default:login`. Así que abre el controlador por defecto del *bundle* `UsuarioBundle` y añade el siguiente código:

```

// src/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class DefaultController extends Controller
{
    public function loginAction()
    {
        $peticion = $this->getRequest();
        $sesion = $peticion->getSession();

        $error = $peticion->attributes->get(
            SecurityContext::AUTHENTICATION_ERROR,
            $sesion->get(SecurityContext::AUTHENTICATION_ERROR)
        );

        return $this->render('UsuarioBundle:Default:login.html.twig', array(
            'last_username' => $sesion->get(SecurityContext::LAST_USERNAME),
            'error'           => $error
        ));
    }
}

```

La clave para entender el código anterior es la clase `SecurityContext`. Esta clase es el punto de entrada al componente de seguridad de Symfony2. A través de `SecurityContext` se obtiene el *token* que representa al usuario de la aplicación (incluso a los usuarios anónimos).

La variable `$error` obtiene el valor del último error producido relacionado con la seguridad. Primero trata de obtener esa información a través de la propiedad `attributes` de la petición (que equivale a la variable `$_SERVER['PATH_INFO']` de PHP). Si no la encuentra, busca el posible error en la sesión activa.

No es obligatorio que la acción de *login* busque estos errores, pero es recomendable hacerlo en las aplicaciones web reales. Así se mejora la experiencia de usuario, ya que la plantilla podrá mostrar al usuario cuál es exactamente el último error producido (la contraseña está mal escrita, el usuario está deshabilitado, etc.)

Después, se pasan a la plantilla `login.html.twig` las variables `$error` y `$last_username`. Esta última contiene el valor del último nombre de usuario utilizado para intentar hacer *login*.

La plantilla necesaria para mostrar el formulario de *login* es la siguiente:

```
{# src/Cupon/UsuarioBunle/Resources/views/Default/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="Acceder" />
    </form>
{% endblock %}

{% block aside %}{% endblock %}
```

Puedes utilizar cualquier plantilla para mostrar el formulario de *login* siempre que cumplas las siguientes tres condiciones:

- El atributo `action` del formulario es la ruta que apunta a la acción `login_check`. No es necesario crear esta acción porque Symfony2 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, añade una plantilla llamada `cajaLogin.html.twig` en el *bundle UsuarioBundle*:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.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('UsuarioBundle:Default:cajaLogin')) }}
</section>
{%
    endblock %
}</aside>
{%
    endblock %}
```

Para que esta plantilla pueda funcionar, no olvides añadir la acción `cajaLoginAction()` en el *bundle UsuarioBundle*. Utiliza 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 Symfony2 muestra un error. El motivo es que los usuarios de prueba guardan su contraseña en claro en la base de datos y Symfony2 codifica la contraseña del formulario de *login* mediante SHA-512.

Hasta que no se actualicen los archivos de datos o *fixtures* más adelante, la única solución posible para poder probar el *login* consiste en modificar la configuración de seguridad de la aplicación para que las contraseñas de los usuarios no se codifiquen:

```
# app/config/security.yml
security:
    # ...
    encoders:
        Cupon\UsuarioBundle\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.

NOTA Aunque cambies la codificación a `plaintext`, las contraseñas que se crearon con los *fixtures* de los capítulos anteriores no van a funcionar. El motivo es que, además de la contraseña, a cada usuario se le generó un valor aleatorio para su propiedad `salt`. Así que para poder acceder a la aplicación con alguno de los usuarios de prueba, debes modificar primero el valor de su `salt` sustituyéndolo por `null` o una cadena de texto vacía.

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 Symfony2 incluye soporte para esta opción, por lo que puedes incorporarla fácilmente a tus aplicaciones. En primer lugar, añade la opción `remember_me` al `firewall` que utilice esta característica:

```
# app/config/security.yml
firewalls:
    frontend:
        # ...
        remember_me:
            key:      cupon1234
            lifetime: 3600
```

La opción `key` es un valor aleatorio y secreto utilizado para codificar el contenido de la *cookie* que se envía al usuario para que permanezca conectado. La opción `lifetime` indica, en segundos, el tiempo que el usuario permanece *logueado* sin que se le vuelva a pedir su contraseña.

Después de configurar la opción `remember_me`, modifica los formularios de *login* para añadir la opción que permite a los usuarios decidir si quieren permanecer conectados:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/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>
{# ... #}

{# src/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.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 Symfony2 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/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.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 Symfony2 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 581) dedicado a la inyección de dependencias. Si no lo haces, te resultará muy difícil entender el código de los siguientes ejemplos.

Symfony2 utiliza el patrón de diseño *Observer* ([http://es.wikipedia.org/wiki/Observer_\(patr%C3%B3n_de_dise%C3%BAo\)](http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%BAo))) para la gestión de sus eventos:

- Symfony2 notifica diversos eventos en los momentos más importantes que se producen durante la ejecución de la petición.
- Para ejecutar código cuando se produzca uno de esos eventos, se crea una clase llamada *listener* con el código a ejecutar y se indica al *central event dispatcher* que te interesa un determinado evento.
- Cada vez que se produzca ese evento, Symfony2 lo notifica al *dispatcher*.
- El *dispatcher* ejecuta el código de todas las clases que antes expresaron su interés por ese evento.

El evento relacionado con el *login* del usuario se llama `security.interactive_login` y se ejecuta justo después de que el usuario introduzca su usuario y contraseña y pulse el botón *Acceder*. Así que el primer paso consiste en indicar a Symfony2 nuestro interés por este evento.

La configuración de los eventos y *listeners* se realiza en el archivo `services.yml` de cada *bundle*. Como el evento que se quiere utilizar tiene relación con el usuario, se recomienda definirlo dentro del *bundle* `UsuarioBundle`:

```
# src/Cupon/UsuarioBundle/Resources/config/services.yml
parameters:
#    usuario.example.class: Cupon\UsuarioBundle\Example

services:
    login_listener:
        class: Cupon\UsuarioBundle\Listener\LoginListener
        tags:
            - { name: kernel.event_listener, event: security.interactive_login }
```

Por convención, las clases que actúan de *listeners* para ejecutar el código relacionado con el evento 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/Cupon/UsuarioBundle/Listener/LoginListener.php
namespace Cupon\UsuarioBundle\Listener;

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;

class LoginListener
{
    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)
```

```
{  
}  
}
```

El *dispatcher* entrega a cada *listener* un objeto llamado `$event` con información útil sobre el propio evento y acceso a varios objetos de la aplicación. El problema es que 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`.

Para conocer el tipo de objeto que se pasa, tendrás que acceder a la definición del evento dentro del código fuente de Symfony2. Igualmente, tendrás que ver el código de la clase del evento para descubrir los métodos que proporciona. Si observas el código de la clase `Symfony\Component\Security\Http\Event\InteractiveLoginEvent`, verás 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 fácilmente mediante el siguiente código (la entidad `Usuario` debería contener una propiedad llamada `ultimaConexion` para que el siguiente código funcione):

```
// src/Cupon/UsuarioBundle/Listener/LoginListener.php  
namespace Cupon\UsuarioBundle\Listener;  
  
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;  
  
class LoginListener  
{  
    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)  
    {  
        $usuario = $event->getAuthenticationToken()->getUser();  
        $usuario->setUltimaConexion(new \DateTime());  
    }  
}
```

El problema del evento `InteractiveLoginEvent` es que 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:

```
# src/Cupon/UsuarioBundle/Resources/config/services.yml  
services:  
    login_listener:  
        class: Cupon\UsuarioBundle\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/Cupon/UsuarioBundle/Listener/LoginListener.php
namespace Cupon\UsuarioBundle\Listener;

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\HttpFoundation\RedirectResponse;

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) {
            $portada = ...
            $event->setResponse(new RedirectResponse($portada));
        }
    }
}
```

El único problema del código anterior es que para generar la ruta de la portada es necesario acceder al sistema de enrutamiento de la aplicación. Así que al definir el servicio `login_listener`, debes injectar el servicio `router` como argumento:

```
# src/Cupon/UsuarioBundle/Resources/config/services.yml
services:
    login_listener:
        class:      Cupon\UsuarioBundle\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 Symfony2 al crear el servicio y genera la ruta de la portada de cada usuario:

```
// src/Cupon/UsuarioBundle/Listener/LoginListener.php
namespace Cupon\UsuarioBundle\Listener;
```

```

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Routing\Router;

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) {
            $portada = $this->router->generate('portada', array(
                'ciudad' => $this->ciudad
            ));
            $event->setResponse(new RedirectResponse($portada));
        }
    }
}

```

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 356) 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 *role*.

8.5.1 Obteniendo el usuario *logueado*

Dentro de un controlador, el usuario se obtiene a través del servicio `security.context`:

```

class DefaultController extends Controller
{

```

```

public function defaultAction()
{
    $usuario = $this->get('security.context')->getToken()->getUser();
    $nombre = $usuario->getNombre();

    // ...
}
}

```

El método `getToken()` devuelve el *token* que representa al usuario *logueado*. Utilizando el método `getUser()` del *token* se obtiene el objeto de la entidad `Usuario` que representa al usuario *logueado*.

NOTA A partir de la versión 2.1 de Symfony, si tu controlador extiende del controlador base `Controller`, dispones de un atajo para obtener el usuario *logueado* más fácilmente: `$this->getUser()`.

Dentro de una plantilla, el usuario *logueado* se obtiene a través de la propiedad `user` de la variable global `app` creada por Symfony2:

```

{% 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 *role*. Dentro de un controlador utiliza el método `isGranted()`:

```

class DefaultController extends Controller
{
    public function defaultAction()
    {
        if ($this->get('security.context')->isGranted('ROLE_USUARIO')) {
            // el usuario tiene el role 'ROLE_USUARIO'
        }
        elseif ($this->get('security.context')->isGranted('ROLE_ADMIN')) {
            // el usuario tiene el role 'ROLE_ADMIN'
        }

        // ...
    }
}

```

El método `isGranted()` es muy fácil de utilizar pero su lógica es bastante pesada, por lo que debes minimizar en lo posible su uso. 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->get('security.context')->isGranted('ROLE_ADMIN')) {
            throw new AccessDeniedException();
        }

        // ...
    }
}
```

Además de los *roles* que definas en tu aplicación, Symfony2 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()
    {
        $contexto = $this->get('security.context');

        if ($contexto->isGranted('IS_AUTHENTICATED_FULLY')) {
            // El usuario está autenticado y la razón es que acaba de
            // introducir su nombre de usuario y contraseña
        }
        elseif ($contexto->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 ($contexto->isGranted('IS_AUTHENTICATED_ANONYMOUSLY')) {
            // Se trata de un usuario anónimo. Técnicamente, en Symfony2 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 %}
```

El único inconveniente de la función `is_granted()` es que si la utilizas en una plantilla cuya URL no se encuentra bajo algún *firewall*, se produce un error y Twig muestra una excepción.

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 `cajaLogin.html.twig` del *bundle UsuarioBundle*. 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 Doctrine2, 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/Cupon/UsuarioBundle/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 archivo de enrutamiento del *bundle* `UsuarioBundle`, junto con las rutas `usuario_login` y `usuario_login_check`.

Al igual que sucede con `usuario_login_check`, no es necesario crear ninguna acción para el *logout*, ya que Symfony2 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
```

La configuración anterior hace que el usuario se desconecte automáticamente cada vez que visita la URL de la ruta `usuario_logout` (en este caso, `/usuario/logout`). Además, una vez desconectado se le redirige a la URL `/sitio/vuelve-pronto`, donde se podría mostrar por ejemplo un mensaje que anime al usuario a volver a conectarse pronto al sitio web.

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, Symfony2 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://cupon.local/app_dev.php/usuario/compras/?_switch_user=usuario3@localhost
http://cupon.local/app_dev.php/?_switch_user=usuario100@localhost
```

Para volver a utilizar el usuario original, indica el valor especial `_exit` (`http://cupon.local/app_dev.php/usuario/compras/?_switch_user=_exit`)

Por motivos de seguridad Symfony2 no deja impersonarse a cualquier usuario. Así, por defecto sólo pueden utilizar esta característica los usuarios que dispongan del *role* `ROLE_ALLOWED_TO_SWITCH`. Para restringir todavía más su uso, puedes indicar el *role* 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 `sha512`. 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 Cupon\UsuarioBundle\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()`. Si la entidad dispone de una propiedad `salt`, pásala como segundo argumento.

Codificar las contraseñas dentro de un archivo de datos es igual de sencillo, pero antes debes injectar el contenedor de servicios en la clase (tal como se explica en el apéndice B (página 581)):

```
// src/Cupon/UsuarioBundle/DataFixtures/ORM/Usuarios.php  
namespace Cupon\UsuarioBundle\DataFixtures\ORM;  
  
use Doctrine\Common\DataFixtures\FixtureInterface;  
use Doctrine\Common\Persistence\ObjectManager;  
use Symfony\Component\DependencyInjection\ContainerAwareInterface;  
use Symfony\Component\DependencyInjection\ContainerInterface;  
use Cupon\UsuarioBundle\Entity\Usuario;  
  
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();  
  
            $passwordEnClaro = 'usuario' . $i;  
            $salt = md5(time());
```

```

$encoder = $this->container->get('security.encoder_factory')
    ->getEncoder($usuario);
$password = $encoder->encodePassword($passwordEnClaro, $salt);

$usuario->setPassword($password);
$usuario->setSalt($salt);

// ...
}

}

}

```

Si en las pruebas anteriores habías modificado la configuración del archivo `security.yml`, vuelve a cambiarla para utilizar `sha512` en las contraseñas de los usuarios:

```

# app/config/security.yml
security:
    # ...
    encoders:
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512,
                                                iterations: 10 }

```

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.

Si conoces los formularios de Symfony 1, puedes olvidar todo lo aprendido, ya que los formularios de Symfony2 son completamente diferentes. Como sucede con la mayoría de componentes, resulta muy sencillo crear los formularios para los casos de uso habituales mientras que su arquitectura interna es lo bastante flexible como para adaptarse a los casos de uso más especiales.

Antes de empezar a crear el formulario, prepara la nueva ruta, acción y plantilla. Añade en primer lugar una ruta dentro del *bundle UsuarioBundle* para acceder al formulario de registro:

```

# src/Cupon/UsuarioBundle/Resources/config/routing.yml
usuario_registro:
    path:      /registro
    defaults: { _controller: UsuarioBundle:Default:registro }

```

Después, crea una acción llamada `registroAction()` dentro del controlador por defecto:

```
// src/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    public function registroAction()
    {
        return $this->render('UsuarioBundle:Default:registro.html.twig');
    }
}
```

Y por último, crea una plantilla llamada `registro.html.twig`:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/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/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Cupon\UsuarioBundle\Entity\Usuario;

class DefaultController extends Controller
{
    // ...

    public function registroAction()
    {
        $usuario = new Usuario();

        $formulario = $this->createFormBuilder($usuario)
            ->add('nombre')
            ->add('apellidos')
```

```
        ->add('direccion', 'text')
        ->add('fechaNacimiento', 'date')
    ->getForm();

    return $this->render(
        'UsuarioBundle:Default: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.

La clase *form builder* es de tipo "*fluent interface*", que permite encadenar todas las llamadas a sus métodos. Si observas el código anterior, verás que el resultado es un código mucho más conciso. Para conseguirlo, todos los métodos de la clase incluyen la siguiente instrucción al final: `return $this;`

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 cometan 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 Symfony2 es definir una clase por cada formulario. Estas clases se definen por defecto en el directorio `Form/` del *bundle* y su nombre acaba en `Type`. De esta for-

ma, la clase de un formulario para registrar usuarios podría llamarse `UsuarioType.php` y se guardaría en el directorio `src/Cupon/UsuarioBundle/Form/`.

Por otra parte, en las aplicaciones web suelen utilizarse formularios diferentes para gestionar los datos de una misma entidad. Así por ejemplo, en la aplicación *Cupon* existe un formulario para que los usuarios del *frontend* se puedan registrar en el sitio web y existe otro formulario para que los administradores del *backend* puedan ver y modificar los datos de cualquier usuario.

En el formulario de registro no tiene sentido incluir campos como `salt` y `fecha_alta`, ya que son campos que sólo los administradores deben ver. Para distinguirlos puedes crear formularios con diferente nombre: `Form/UsuarioRegistroType.php` y `Form/UsuarioCompletoType.php`. Otra alternativa consiste en utilizar subdirectorios: `/Form/Frontend/UsuarioType.php`, `Form/Backend/UsuarioType.php`, etc. A lo largo de los próximos capítulos se utilizará esta última variante.

Crea el directorio `src/Cupon/UsuarioBundle/Form/Frontend/` y añade en su interior una clase llamada `UsuarioType.php` con el siguiente código:

```
// src/Cupon/UsuarioBundle/Form/Frontend/UsuarioType.php
namespace Cupon\UsuarioBundle\Form\Frontend;

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('permite_email')
            ->add('fecha_nacimiento')
            ->add('dni')
            ->add('numero_tarjeta')
            ->add('ciudad')
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Cupon\UsuarioBundle\Entity\Usuario'
        ));
    }
}
```

```

public function getName()
{
    return 'cupon_usuariobundle_usuariotype';
}
}

```

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 `setDefaultOptions()` 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 `getName()`. Este nombre normalmente sigue la nomenclatura `{nombre-proyecto}_{nombre-bundle}_{nombre-clase}`.

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 Symfony2 el que decida cuál es el tipo de campo más adecuado para cada propiedad.

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/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Cupon\UsuarioBundle\Entity\Usuario;
use Cupon\UsuarioBundle\Form\Frontend\UsuarioType;

class DefaultController extends Controller
{
    // ...

    public function registroAction()
    {
        $usuario = new Usuario();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        return $this->render(
            'UsuarioBundle:Default: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 una instancia de la clase del formulario (no olvides importarla previamente con la instrucción `use`).

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 hacer que la casilla que permite el envío de emails publicitarios esté marcada y que la fecha de nacimiento sea la que corresponde a alguien que tiene 18 años, la edad mínima para poder apuntarse al sitio web:

```
// src/Cupon/UsuarioBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function registroAction()
    {
        $usuario = new Usuario();
        $usuario->setPermiteEmail(true);
        $usuario->setFechaNacimiento(new \DateTime('today - 18 years'));

        $formulario = $this->createForm(new UsuarioType(), $usuario);

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

Independientemente de si creas el formulario en el controlador o mediante una clase, la forma de mostrarlo en la plantilla siempre es la misma:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/registro.html.twig #}
{% extends '::frontend.html.twig' %}

{% block id 'usuario' %}
{% block title %}Regístrate gratis como usuario{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<form action="{{ path('usuario_registro') }}" method="post" {{ form_enctype(formulario) }}>
    {{ form_widget(formulario) }}
    <input class="boton" type="submit" value="Registrarme" />
</form>
{% endblock %}

{% block aside %}{% endblock %}
```

La función `form_widget()` 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.

Lo único que debes añadir en la plantilla es la etiqueta `<form>` y el botón de envío del formulario. El atributo `action` de la etiqueta `<form>` se obtiene mediante la ruta definida previamente. La función `form_enctype()` de Twig determina si es necesario añadir el atributo `enctype="multipart/form-data"` porque el formulario incluye campos para subir archivos.

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 `cajaLogin.html.twig` y añade lo siguiente:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.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, Symfony2 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/Cupon/UsuarioBundle/Form/Frontend/UsuarioType.php
namespace Cupon\UsuarioBundle\Form\Frontend;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('apellidos')
            ->add('email', 'email')
            ->add('password')
            ->add('direccion')
            ->add('permite_email')
```

```

        ->add('fecha_nacimiento', 'birthday')
        ->add('dni')
        ->add('numero_tarjeta')
        ->add('ciudad')
    ;
}

// ...
}

```

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. Symfony2 dispone de un campo especial llamado `repeated` que muestra un campo dos veces y obliga a escribir el mismo valor en los dos:

```

// src/Cupon/UsuarioBundle/Form/Frontend/UsuarioType.php
namespace Cupon\UsuarioBundle\Form\Frontend;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('apellidos')
            ->add('email', 'email')

            ->add('password', 'repeated', array(
                'type' => 'password',
                'invalid_message' => 'Las dos contraseñas deben coincidir',
                'options' => array('label' => 'Contraseña')
            ))

            ->add('direccion')
            ->add('permite_email', 'checkbox', array('required' => false))
            ->add('fecha_nacimiento', 'birthday')
            ->add('dni')
            ->add('numero_tarjeta')
            ->add('ciudad')
    }
}

```

```

    ;
}

// ...
}

```

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.
- `options`, este array permite definir otras opciones de los campos de formulario 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.

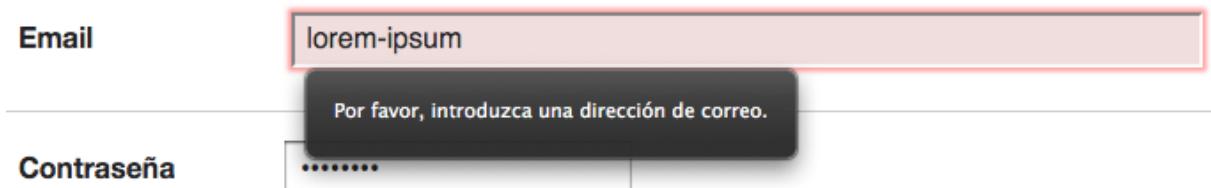


Figura 8.2 Mensaje de error mostrado por Google Chrome cuando no escribes un email válido

Por otra parte, Symfony2 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()`:

```

->add('permite_email', '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 Symfony2 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 expresión `{{ form_widget(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_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_rest(formulario)`, muestra todos los campos del formulario que no se hayan mostrado individualmente con la función `form_row()`. Resulta muy recomendable utilizar esta función en todos los formularios, para no tener que lidiar con campos especiales como los campos ocultos y los que se emplean para la protección frente a los ataques CSRF.

Utilizando las funciones anteriores puedes reordenar y agrupar los campos del formulario como quieras:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/registro.html.twig #}

{# ... #-}

<form action="{{ path('usuario_registro') }}" method="post" {{ form_enctype(formulario) }}>
    {{ form_errors(formulario) }}

    <fieldset>
        <legend>Datos personales</legend>

        {{ form_row(formulario.nombre) }}
        {{ form_row(formulario.apellidos) }}
        {{ form_row(formulario.fecha_nacimiento) }}
        {{ 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.permite_email) }}

{{ form_rest(formulario) }}

<input class="boton" type="submit" value="Registrarme" />
</form>

{# ... #}
```

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:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/registro.html.twig #-}

{# ... #}

<form action="{{ path('usuario_registro') }}" method="post" {{ form_enctype(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.fecha_nacimiento) }}
{{ form_row(formulario.dni) }}
{{ form_row(formulario.direccion) }}
</fieldset>
```

```
{# ... #}
```

Symfony2 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 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

Symfony2 utiliza la misma estrategia que Symfony 1 para procesar los formularios. Un único controlador se encarga de mostrar el formulario inicial vacío y de 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/Cupon/UsuarioBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function registroAction()
    {
        $peticion = $this->getRequest();

        $usuario = new Usuario();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            // Validar los datos enviados y guardarlos en la base de datos
        }

        return $this->render(
```

```
        'UsuarioBundle:Default:registro.html.twig',
        array('formulario' => $formulario->createView())
    );
}
}
```

Symfony2 simplifica el procesado de los formularios a solamente dos líneas de código:

```
$formulario->handleRequest($peticion);
if ($formulario->isValid()) {
    // ...
}
```

El método `handleRequest()` asocia el objeto `$formulario` creado anteriormente con los datos enviados por el usuario en la `$peticion`. Después de ejecutar esta instrucción, el objeto asociado con el formulario (en el ejemplo anterior, el objeto `$usuario`) está lleno 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 Symfony2 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/Cupon/UsuarioBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function registroAction()
    {
        $peticion = $this->getRequest();

        $usuario = new Usuario();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            $encoder = $this->get('security.encoder_factory')
                ->getEncoder($usuario);
            $usuario->setSalt(md5(time()));
            $passwordCodificado = $encoder->encodePassword(
                $usuario->getPassword(),
                $usuario->getSalt()
            );
            $usuario->setPassword($passwordCodificado);

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

```

```

        $em->persist($usuario);
        $em->flush();

        return $this->redirect($this->generateUrl('portada'));
    }

    return $this->render(
        'UsuarioBundle:Default: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: se establece un valor a la propiedad `salt` (que es un campo que no existe en el formulario de registro) y se localiza el *encoder* apropiado para la entidad `Usuario`.

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 Doctrine2.

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/Cupon/UsuarioBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function registroAction()
    {
        $peticion = $this->getRequest();

        $usuario = new Usuario();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            // ...

            $this->get('session')->getFlashBag()->add('info',
                ' Enhorabuena! Te has registrado correctamente en Cupon'
            );
        }

        return $this->redirect($this->generateUrl('portada', array(
            'ciudad' => $usuario->getCiudad()->getSlug()
        )));
    }
}

```

```

        ))));
}

// ...

```

Los mensajes *flash* se crean a través de la sesión del usuario. Utiliza el método `getFlashBag()` para acceder al objeto que almacena todos los mensajes *flash* y después, utiliza el método `add()` para añadir un nuevo mensaje.

El primer argumento del método `add()` es el nombre o clave que identifica al tipo de mensaje y puedes elegirlo libremente (a diferencia de Symfony 2.0, en Symfony 2.3 dos o más mensajes *flash* sí que pueden compartir el mismo tipo). El 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:

```

{# src/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.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/Cupon/UsuarioBundle/Controller/DefaultController.php
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;

class DefaultController extends Controller
{
    public function registroAction()
    {
        $peticion = $this->getRequest();

        $usuario = new Usuario();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            // ...

            $this->get('session')->getFlashBag()->add('info',
                'Enhorabuena! Te has registrado correctamente en Cupon'
            );

            $token = new UsernamePasswordToken(
                $usuario,
                $usuario->getPassword(),
                'frontend',
                $usuario->getRoles()
            );
            $this->container->get('security.context')->setToken($token);

            return $this->redirect($this->generateUrl('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/Cupon/UsuarioBundle/Entity/Usuario.php
namespace Cupon\UsuarioBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @ORM\Entity
 */
class Usuario implements UserInterface
{
    // ...

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

    // ...
}
```

Symfony2 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\`. Symfony2 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 Symfony2 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/Cupon/UsuarioBundle/Entity/Usuario.php
namespace Cupon\UsuarioBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @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 incluídas en Symfony2

Symfony2 incluye 43 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 quieras 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 quieras 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. La opción `version` permite restringir la validez a sólo direcciones IPv4, sólo IPv6, las dos versiones del protocolo, todas las direcciones salvo los rangos privados, etc. Consulta el código de la clase `IpValidator.php` para descubrir todos los valores posibles.

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 = {"nacional", "extranjero"})`. Si se permite seleccionar varios valores, añade la opción `multiple = true`. Para limitar el mínimo o máximo número de valores elegidos, utiliza respectivamente las opciones `min` y `max`. La opción `callback` permite indicar un *callback* de PHP encargado de generar las opciones disponibles.
- `@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. Ej: si la propiedad se define como `protected $cuenta = array('numero', 'titular')` la validación podría ser `@Assert\Collection(fields = {"numero" = @Assert\Regex(path="/\d{20}/"), "titular" = @Assert\NotBlank()})`
- `@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\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class Usuario
{
    // ...
}
```

- Para hacer que varios campos sean únicos, añade una validación `@UniqueEntity` para cada campo:

```
/**
 * @ORM\Entity
 * @UniqueEntity("email")
 * @UniqueEntity("dni")
 */

```

```
class Usuario
{
    // ...
}
```

- Por último, para hacer que una combinación de campos sea única, sin importar si el valor individual de cada uno de ellos se repite, utiliza la opción `fields` de la siguiente manera: `@UniqueEntity(fields = {"email", "dni"})`

Validaciones para **archivos**:

- `@Assert\File()`, el valor es un objeto de tipo `File`, `UploadedFile` o una cadena de texto (un objeto convertible a cadena de texto) que corresponde a la ruta de un archivo existente. Dispone de la opción `maxSize` para limitar su tamaño en bytes (o kilobytes con el sufijo `k` o megas con el sufijo `M`). También se puede limitar el tipo de archivo subido con la opción `mimeTypes` indicando un tipo o una lista de tipos permitidos. 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, Symfony2 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/Cupon/UsuarioBundle/Entity/Usuario.php
namespace Cupon\UsuarioBundle\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;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @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 Symfony2 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 Symfony2, el método que valida el DNI se llama [esDniValido\(\)](#):

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php

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;
use Symfony\Component\Validator\ExecutionContextInterface;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @ORM\Entity()
 * @DoctrineAssert\UniqueEntity("email")
 * @Assert\Callback(methods={"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 único propósito es añadir *violaciones* de validación cuando se produzca algún error. 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 Symfony2 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/Cupon/UsuarioBundle/Entity/Usuario.php
```

```
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;
use Symfony\Component\Validator\ExecutionContext;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @ORM\Entity()
 * @DoctrineAssert\UniqueEntity("email")
 * @Assert\Callback(methods={"esDniValido"})
 */
class Usuario implements UserInterface
{
    // ...

    public function esDniValido(ExecutionContext $context)
    {
        $dni = $this->getDni();

        // Comprobar que el formato sea correcto
        if (0 === preg_match("/\d{1,8}[a-z]/i", $dni)) {
            $context->addViolationAt('dni', 'El DNI introducido no tiene el
formato correcto (entre 1 y 8 números seguidos de una letra, sin guiones y sin
dejar ningún espacio en blanco)', array(), null);

            return;
        }

        // Comprobar que la letra cumple con el algoritmo
        $numero = substr($dni, 0, -1);
        $letra = strtoupper(substr($dni, -1));
        if ($letra != substr("TRWAGMYFPDXBNJZSQVHLCKE", strr($numero, "XYZ",
"012")%23, 1)) {
            $context->addViolationAt('dni', 'La letra no coincide con el número
del DNI. Comprueba que has escrito bien tanto el número como la letra',
array(), null);
        }
    }
}
```

Las *violaciones* se añaden con el método `addViolationAt()`, que espera como primer parámetro el nombre de la propiedad cuyo valor es erróneo y como segundo parámetro el mensaje de error personalizado.

8.7.4.3 Métodos de validación *ad-hoc*

Symfony2 incluye otra forma más sencilla de crear validadores propios que puedes utilizar cuando no necesites que los mensajes de error se muestren al lado del campo erróneo. Su funcionamiento se basa en crear un método llamado `isXXX()` dentro de la entidad y asociarle una regla de validación de tipo `@Assert\True`. Así por ejemplo podrías utilizar el siguiente código para asegurar que todos los usuarios que se registran sean mayores de edad:

```
// src/Cupon/UsuarioBundle/Entity/Usuario.php
namespace Cupon\UsuarioBundle\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;
use Symfony\Component\Validator\ExecutionContextInterface;

/**
 * Cupon\UsuarioBundle\Entity\Usuario
 *
 * @ORM\Entity()
 * ...
 */
class Usuario implements UserInterface
{
    // ...

    /**
     * @Assert\True(message = "Debes tener al menos 18 años")
     */
    public function isMayorDeEdad()
    {
        return $this->fecha_nacimiento <= 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()`. 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.

8.8 Visualizando el perfil del usuario

Los usuarios registrados y *logueados* en el sitio web pueden visualizar toda la información de su perfil para actualizar cualquier información que deseen. Para añadir esta nueva funcionalidad en la aplicación, define en primer lugar una ruta llamada `usuario_perfil` en el *bundle* `UsuarioBundle`:

```
# src/Cupon/UsuarioBundle/Resources/config/routing.yml
usuario_perfil:
    path:      /perfil
    defaults: { _controller: UsuarioBundle:Default:perfil }
```

Después, actualiza el enlace de la opción *Ver mi perfil* en la plantilla `cajaLogin.html.twig`:

```
{# src/Cupon/UsuarioBundle/Resources/views/Default/cajaLogin.html.twig #}
{% if is_granted('ROLE_USUARIO') %}
    {% include ':includes: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 añade la acción `perfilAction()` en el controlador por defecto del *bundle*:

```
// src/Cupon/UsuarioBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    // ...

    public function perfilAction()
    {
        // Obtener los datos del usuario logueado y utilizarlos para
        // rellenar un formulario de registro.
        //
        // Si la petición es GET, mostrar el formulario
        // Si la petición es POST, actualizar la información del usuario con
        // los nuevos datos obtenidos del formulario
    }
}
```

El usuario *logueado* se obtiene directamente a través del *token* que crea el componente de seguridad para representar al usuario:

```
public function perfilAction()
{
    $usuario = $this->get('security.context')->getToken()->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:

```
use Cupon\UsuarioBundle\Form\Frontend\UsuarioType;
// ...
public function perfilAction()
{
    $usuario = $this->get('security.context')->getToken()->getUser();
    $formulario = $this->createForm(new UsuarioType(), $usuario);
}
```

Después utiliza la lógica habitual de Symfony2 para mostrar el formulario o actualizar la información en función del tipo de petición:

```
use Cupon\UsuarioBundle\Form\Frontend\UsuarioType;
// ...
public function perfilAction()
{
    $usuario = $this->get('security.context')->getToken()->getUser();
    $formulario = $this->createForm(new UsuarioType(), $usuario);

    $peticion = $this->getRequest();

    $formulario->handleRequest($peticion);

    if ($formulario->isValid()) {
        // actualizar el perfil del usuario
    }

    return $this->render('UsuarioBundle:Default: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`:

```
{# #}
{% extends '::frontend.html.twig' %}

{% block id 'usuario' %}
{% block title %}Ver / Modificar mis datos{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<form action="{{ path('usuario_perfil') }}" method="post"
{{ form_enctype(formulario) }}>
<div class="errors">
    {{ form_errors(formulario) }}
</div>
```

```

<div>
    {{ form_row(formulario.nombre) }}
</div>

<div>
    {{ form_row(formulario.apellidos) }}
</div>

{# ... #}

<div>
    {{ form_errors(formulario.permite_email) }}
    {{ form_widget(formulario.permite_email) }}
    <span>Me gustaría recibir el boletín de ofertas de Cupon</span>
</div>

{{ form_rest(formulario) }}

<input class="boton" type="submit" value="Guardar cambios" />
</form>
{% endblock %}

```

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:

```

use Cupon\UsuarioBundle\Form\Frontend\UsuarioType;
// ...
public function perfilAction()
{
    $usuario = $this->get('security.context')->getToken()->getUser();
    $formulario = $this->createForm(new UsuarioType(), $usuario);

    $peticion = $this->getRequest();

    $formulario->handleRequest($peticion);

    if ($formulario->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($usuario);
        $em->flush();

        $this->get('session')->setFlashBag()->add('info',
            'Los datos de tu perfil se han actualizado correctamente'
        );
        return $this->redirect($this->generateUrl('usuario_perfil'));
    }

    return $this->render('UsuarioBundle:Default: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 Doctrine2. 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 Modificando la 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.

Symfony2 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, cuando el usuario no quiere cambiar su contraseña y deja los dos campos vacíos, su valor es `null`.

Si recuerdas la explicación de las secciones anteriores, después de ejecutar la instrucción `$formulario->handleRequest($peticion)`, el objeto del formulario se rellena con la información enviada por el usuario, sea esta válida o no. Así que cuando el usuario no cambia su contraseña, después de hacer el `handleRequest()` la contraseña vale `null`.

Para solucionarlo, basta con guardar el valor original de la contraseña del usuario antes de la llamada al método `handleRequest()`. Los formularios de Symfony2 permiten recuperar la entidad original utilizada para crear el formulario mediante el método `getData()`, por lo que la contraseña original se encuentra en:

```

$passwordOriginal = $formulario->getData()->getPassword();

```

Si la nueva contraseña del usuario es `null`, eso significa que no la ha cambiado y se utiliza directamente su contraseña original. Si la nueva contraseña no es `null`, quiere decir que el usuario desea cambiarla, por lo que se codifica su nuevo valor antes de guardar la entidad. El código completo resultante de la acción `perfilAction()` es por tanto el siguiente:

```
// src/Cupon/UsuarioBundle/Controller/DefaultController.php
namespace Cupon\UsuarioBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;
use Cupon\UsuarioBundle\Form\Frontend\UsuarioType;

class DefaultController extends Controller
{
    // ...

    public function perfilAction()
    {
        $peticion = $this->getRequest();
        $usuario = $this->get('security.context')->getToken()->getUser();
        $formulario = $this->createForm(new UsuarioType(), $usuario);

        $passwordOriginal = $formulario->getData()->getPassword();
        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            if (null == $usuario->getPassword()) {
                $usuario->setPassword($passwordOriginal);
            } else {
                $encoder = $this->get('security.encoder_factory')
                    ->getEncoder($usuario);
                $passwordCodificado = $encoder->encodePassword(
                    $usuario->getPassword(),
                    $usuario->getSalt()
                );
                $usuario->setPassword($passwordCodificado);
            }
        }

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

        $this->get('session')->setFlashBag()->add('info',
            'Los datos de tu perfil se han actualizado correctamente'
        );

        return $this->redirect($this->generateUrl('usuario_perfil'));
    }

    return $this->render('UsuarioBundle:Default:perfil.html.twig', array(
        'usuario'    => $usuario,
        'formulario' => $formulario->createView()
    ));
}
```

```
|    }  
|}  
|
```

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 Symfony2, 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`, Symfony2 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/Cupon/CiudadBundle/Resources/config/routing.yml
ciudad_recientes:
    path:      /{ciudad}/recientes.{_format}
    defaults: { _controller: CiudadBundle:Default:recientes }
```

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/Cupon/CiudadBundle/Resources/config/routing.yml
ciudad_recientes:
    path:      /{ciudad}/recientes.{_format}
    defaults: { _controller: CiudadBundle:Default:recientes, _format: html }
```

Ahora que el parámetro `_format` tiene un valor por defecto, al generar la ruta con la función `path()` puedes omitirlo, aunque si lo haces, 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/Cupon/CiudadBundle/Resources/config/routing.yml
ciudad_recientes:
    path:      /{ciudad}/recientes.{_format}
    defaults: { _controller: CiudadBundle:Default:recientes, _format: html }
    requirements:
        _format: html|rss
```

Con la configuración anterior, la instrucción `{{ path('ciudad_recientes', { 'ciudad': 'nombre-ciudad', '_format': 'pdf' }) }}` generaría un error, ya que el formato `pdf` no es un formato permitido.

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 (`CiudadBundle:Default:recientes.html.twig`) y añade el siguiente bloque llamado `rss` para incluir el enlace:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/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 Symfony2 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/Cupon/CiudadBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    // ...

    public function recientesAction($ciudad)
    {
        // ...
    }
}
```

```

    $formato = $this->get('request')->getRequestFormat();

    return $this->render(
        'CiudadBundle:Default:recientes.'.$formato.'.twig',
        array(...))
    );
}
}

```

El controlador obtiene en primer lugar el formato de la petición mediante el método `getRequestFormat()` del objeto de la petición. Después, utiliza el formato para determinar la plantilla que muestra la información: `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:

```

<!-- src/Cupon/CiudadBundle/Resources/views/Default/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>Symfony2</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.

```
{# src/Cupon/OfertaBundle/Resources/views/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>Symfony2</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:

```
{# src/Cupon/OfertaBundle/Resources/views/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[
                
                {{ 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>
```

```

    {% 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 ícono, 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 más sencilla consiste en construir manualmente la URL absoluta de la foto, para lo cual hay que obtener el nombre del servidor a través del parámetro `$_SERVER['SERVER_NAME']` de la petición:

```

{% set urlAbsolutaFoto = 'http://'
    ~ app.request.server.get('SERVER_NAME')
    ~ asset(directorio_imagenes ~ oferta.rutaFoto) %}
```

Juntando todo lo anterior, la plantilla `recientes.rss.twig` definitiva tiene el siguiente aspecto:

```

{{-- src/Cupon/OfertaBundle/Resources/views/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>Symfony2</generator>
        <atom:link rel="self" type="application/rss+xml"
            href="{{ url('ciudad_recientes', {
                'ciudad': ciudad.slug,
                '_format': 'rss'
            }) }}>
    
```

```

    {% for oferta in ofertas %}
        {% set urlAbsolutaFoto = 'http://'
            ~ app.request.server.get('SERVER_NAME')
            ~ asset(directorio_imagenes ~ oferta.rutaFoto)
        %}
        <item>
            <title>{{ oferta.nombre }}</title>
            <link>{{ url('oferta', { 'ciudad': oferta.ciudad.slug,
                'slug': oferta.slug }) }}</link>
            <description><![CDATA[
                
```

```

        {{ 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>Symfony2</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:

```

{# src/Cupon/OfertaBundle/Resources/views/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 %}
        {% set urlAbsolutaFoto = 'http://'
            ~ app.request.server.get('SERVER_NAME')
            ~ asset(directorio_imagenes ~ oferta.rutaFoto) %}

        <item>
            <title>{{ oferta.nombre }}</title>
            <link>{{ url('oferta', { 'ciudad': oferta.ciudad.slug,
                'slug': oferta.slug }) }}</link>
            <description><![CDATA[
                
                {{ 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 `TiendaBundle:Default:portada` y añade lo siguiente:

```

// src/Cupon/TiendaBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function portadaAction($ciudad, $tienda)
    {
        // ...

        $formato = $this->get('request')->getRequestFormat();
        return $this->render(
            'TiendaBundle:Default:portada.'.$formato.'.twig',

```

```

        array(...)

    );
}

}

```

Después, añade un enlace al canal RSS en la plantilla `portada.html.twig`:

```

{% src/Cupon/TiendaBundle/Resources/views/Default/portada.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': tienda.slug, '_format': 'rss' }) }}"
{% endblock %}

{# ... #}

```

Y por último, crea la plantilla `portada.rss.twig`:

```

{% src/Cupon/TiendaBundle/Resources/views/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 %}
        {% set urlAbsolutaFoto = 'http://'
            ~ app.request.server.get('SERVER_NAME')

```

```

        ~ asset(directorio_imagenes ~ oferta.rutaFoto) %}

<item>
    <title>{{ oferta.nombre }}</title>
    <link>{{ url('oferta', { 'ciudad': oferta.ciudad.slug,
                                'slug': oferta.slug }) }}</link>
    <description><![CDATA[
        
        {{ 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 %}

```

9.4 Registrando nuevos formatos

Symfony2 soporta por defecto nueve formatos, que a su vez se corresponden con 14 tipos MIME diferentes. Todos ellos se definen en la clase `Request`:

```

// vendor/symfony/src/Symfony/Component/HttpFoundation/Request.php
protected static function initializeFormats()
{
    static::$formats = array(
        'html' => array('text/html', 'application/xhtml+xml'),
        'txt'  => array('text/plain'),
        'js'   => array('application/javascript', 'application/x-javascript',
                        'text/javascript'),
        'css'  => array('text/css'),
        'json' => array('application/json', 'application/x-json'),
        'xml'  => array('text/xml', 'application/xml', 'application/x-xml'),
        'rdf'  => array('application/rdf+xml'),
        'atom' => array('application/atom+xml'),
        'rss'  => array('application/rss+xml'),
    );
}

```

La ventaja de utilizar formatos y tipos MIME conocidos es que Symfony2 sabe qué tipo de contenido devuelve el objeto `Response` y por tanto, puede establecer el valor más adecuado en la cabecera `Content-Type` de la respuesta.

Si tu aplicación genera por ejemplo respuestas en formato PDF, es aconsejable que lo añadas como formato. Los nuevos formatos se añaden fácilmente utilizando el sistema de eventos de Symfony2.

Para ello, crea un *listener* asociado al evento `kernel.request`. Symfony2 notifica este evento cuando empieza a procesar la petición y por tanto, es el evento ideal para modificar el propio objeto de la petición.

Como se explicó en los capítulos anteriores, por convención las clases *listener* se guardan en el directorio `Listener/` de algún *bundle*. Como en la aplicación `Cupon` se utiliza el *bundle* `OfertaBundle` para guardar todos los elementos comunes que no encajan en ningún *bundle* específico, crea el directorio `src/Cupon/OfertaBundle/Listener/`. En su interior, añade un archivo llamado `RequestListener.php` con el siguiente código:

```
// src/Cupon/OfertaBundle/Listener/RequestListener.php
namespace Cupon\OfertaBundle\Listener;
use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $event->getRequest()->setFormat('pdf', 'application/pdf');
    }
}
```

El código del *listener* es muy sencillo, porque solamente hay que utilizar el método `setFormat()` del objeto de la petición para añadir el nuevo formato y su tipo MIME asociado.

Finalmente, configura el nuevo *listener* para que Symfony2 sepa que debe ejecutarlo cada vez que se produzca el evento `kernel.request`. Esta configuración puedes incluirla por ejemplo en el archivo `app/config/services.yml` creado para los servicios de los capítulos anteriores:

```
# app/config/services.yml
services:
    cupon.ofertabundle.listener.request:
        class: Cupon\OfertaBundle\Listener\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request }
```

Los formatos o tipos MIME soportados por defecto por la clase `Request` no deben confundirse con los tipos MIME soportados por Symfony2. Cuando subes por ejemplo un archivo mediante un formulario, Symfony2 es capaz de detectar decenas de tipos de archivos (tipos MIME y extensiones). La lista completa de extensiones soportadas la puedes encontrar en la clase `MimeTypeExtensionGuesser` del componente `HttpFoundation`.

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 Symfony2 utilizan la nomenclatura habitual de concatenar 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) como por ejemplo:

- `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

Symfony2 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()`:

```
class DefaultController extends Controller
{
    public function defaultAction()
    {
        // ...

        $locale = $this->getRequest()->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 debe 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 Symfony2 incluye una variable especial llamada `_locale`. Si la añades al patrón de la ruta, Symfony2 se encargará de asignarle el valor adecuado para cada usuario:

```

contacto:
    path:      /{_locale}/contacto
    defaults: { _controller: OfertaBundle:Sitio:contacto }

```

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.

Gracias a la variable `_locale`, puedes internacionalizar todas las rutas de la aplicación casi sin esfuerzo. Abre el archivo `app/config/routing.yml` y comienza modificando las rutas sueltas:

```

# app/config/routing.yml

# ...
# Ruta de las páginas estáticas
pagina_estatica:
    path:      /{_locale}/sitio/{pagina}
    defaults: { _controller: OfertaBundle:Sitio:estatica }

# Ruta simple de la portada
_portada:
    path:      /
    defaults: { _controller: FrameworkBundle:Redirect:redirect,
                route: portada, ciudad: '%cupon.ciudad_por_defecto%' }

# Ruta completa de la portada (con el slug de la ciudad)
portada:
    path:      /{_locale}/{ciudad}
    defaults: { _controller: OfertaBundle:Default:portada, _locale: es }

```

La primera ruta modificada es `pagina_estatica`, que simplemente añade la variable `_locale` al principio de la URL, una práctica habitual en las aplicaciones internacionalizadas.

La ruta `portada` también añade la variable `_locale` en la misma posición, pero además, establece su valor por defecto a `es` mediante la opción `defaults`. Así no es necesario actualizar el código de la aplicación. Cuando un controlador redirija al usuario a esta ruta, si no indica el valor del `_locale` no se producirá ningún error, ya que simplemente se utilizará su valor por defecto `es`.

Por último, no es necesario añadir la variable `_locale` en la ruta `_portada`. Recuerda que esta ruta sólo redirige a la ruta `portada` y se define para que los usuarios puedan acceder al sitio web escribiendo simplemente `http://cupon.local/`, sin tener que añadir también el nombre de una ciudad.

El resto de las rutas de la aplicación se definen en cada uno de los *bundles* y se importan desde el archivo `app/config/routing.yml`. Haciendo uso de la opción `prefix`, resulta muy sencillo añadir la variable especial `_locale` a todas las rutas de la aplicación:

```
# app/config/routing.yml
CiudadBundle:
    resource: "@CiudadBundle/Resources/config/routing.yml"
    prefix:  /{_locale}

OfertaBundle:
    resource: "@OfertaBundle/Resources/config/routing.yml"
    prefix:  /{_locale}

TiendaBundle:
    resource: "@TiendaBundle/Resources/config/routing.yml"
    prefix:  /{_locale}

UsuarioBundle:
    resource: "@UsuarioBundle/Resources/config/routing.yml"
    prefix:  /{_locale}/usuario

# ...
```

Si ahora pruebas a navegar por el sitio web, verás que todas las URL de la aplicación incluyen al principio el valor del `_locale`. Si has utilizado la misma configuración que la explicada anteriormente, todas las rutas empezarán por `/es/...`

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/config.yml

CiudadBundle:
    resource: "@CiudadBundle/Resources/config/routing.yml"
    prefix:  /{_locale}
    requirements:
        _locale: en|es

# ...

pagina_estatica:
    path:      /{_locale}/sitio/{pagina}
    defaults: { _controller: OfertaBundle:Sitio:estatica }
    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 Symfony2 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, Symfony2 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}`.

Como este requerimiento es tan habitual en las aplicaciones internacionalizadas, existen varios bundles desarrollados por terceros que añaden esta funcionalidad. El más popular es BeSimpleI18nRoutingBundle (<http://www.symfony.es/bundles/besimple/besimplei18nroutingbundle/>) que permite definir las rutas de la siguiente manera:

```
portada:
    locales: { en: /welcome, fr: /bienvenue, de: /willkommen, es: /bienvenido }
    defaults: { _controller: MiBundle:Default:portada }
```

Y en las plantillas se puede generar cada ruta en función del *locale*:

```
{{ path('portada.en') }}
{{ path('portada', { 'locale': 'en' }) }}

{{ path('portada') }} {# toma el locale de la petición #}
```

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 Symfony2 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:

```
// Traducción al inglés en formato XLIFF  
messages.en.xliff  
  
// Traducción al español en formato YAML  
messages.es.yml  
  
// Traducción al francés en formato PHP  
messages.fr.php
```

Por convención, los catálogos se guardan en el directorio `Resources/translations/` del *bundle*. Este directorio no existe a menos que al generar el *bundle* indicaras que querías crear la estructura completa de directorios. Por otra parte, si quieras redefinir la traducción de algún *bundle* desarrollado por terceros, puedes incluir la nueva traducción en el directorio `app/Resources/translations/`.

El formato XLIFF es el recomendado por Symfony2 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. Como está relacionada con el *bundle* `OfertaBundle`, crea el archivo `src/Cupon/OfertaBundle/Resources/translations/messages.en.xliff` y añade lo siguiente:

```
<?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>  
    </body>  
  </file>  
</xliff>
```

```

        </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 quieras 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:

```

<!-- src/Cupon/OfertaBundle/Resources/translations/messages.en.xliff -->
<?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:

```
<!-- src/Cupon/OfertaBundle/Resources/translations.messages.es.xliff -->
<?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:

```
menu:
    dia:      Oferta del día
    recientes: Ofertas recientes
    mias:     Mis ofertas
```

Si la aplicación que se internacionaliza 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
src/Cupon/OfertaBundle/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 %}
```

Si quieres utilizar un dominio especial para la traducción, indícalo como segundo parámetro:

```

{# Utilizando el filtro trans() #}
{{ "<strong>Faltan</strong>: %fecha%" | trans(
    { '%fecha%': oferta.fechaExpiracion }, 'fechas'
) }}

{# Utilizando la etiqueta {% trans %} #}
{% trans with { '%fecha%': oferta.fechaExpiracion } from 'fechas' %}
    <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**:

```

<!-- src/Cupon/OfertaBundle/Resources/translations/messages.en.xliff -->
<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())
    );
}

```

Si quieres utilizar un dominio específico para traducir el texto, indícalo como tercer parámetro:

```

public function portadaAction($ciudad)
{
    // ...

    $cadena = $this->get('translator')->trans(
        'La oferta caduca el %fecha%',
        array('%fecha%' => $oferta->getFechaExpiracion(),
            'fechas'
    );
}

```

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*".

Symfony2 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 `|`. Symfony2 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:

```
<!-- src/Cupon/OfertaBundle/Resources/translations/messages.en.xliff -->
<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]
    Has 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) (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%
compras para activar la oferta!|]9,Inf] Faltan %faltan% compras para activar la
oferta
    {% endtranschoice %}

    {% else %}
        {# ... #}
    {% endif %}

```

Y la traducción de la cadena de texto anterior en el catálogo:

```

<!-- src/Cupon/OfertaBundle/Resources/translations/messages.en.xliff -->
<trans-unit id="17">
    <source>{1} Una sola compra más activa la oferta!|[1, 9] Sólo faltan
%faltan% compras para activar la oferta!|]9,Inf] Faltan %faltan% compras para
activar la oferta</source>
    <target>{1} Just one more purchase needed to get the deal!|[1, 9] Just
%faltan% more needed to get the deal!|]9,Inf] %faltan% more needed to get the
deal</target>
</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. Desafortunadamente, ni Symfony2 ni Doctrine2 incluyen esta característica tan importante.

Consciente de este problema, un programador llamado Gediminas Morkevicius (<https://github.com/l3pp4rd>) ha publicado varias extensiones para Doctrine2:

- [Tree](#), permite trabajar con las entidades en forma de árbol o *nested set*.
- [Translatable](#), gestiona la traducción de los contenidos de una entidad a diferentes idiomas.
- [Sluggable](#), crea el *slug* de todas las propiedades que quieras de la entidad.
- [Timestampable](#), añade propiedades para guardar la fecha de creación y/o modificación de la entidad.
- [Loggable](#), guarda todo el historial de cambios de la entidad y permite el uso de versiones.

- **Sortable**, permite ordenar las entidades según cualquiera de sus propiedades.

Para facilitar la instalación y uso de las extensiones en Symfony2, otro programador llamado Christophe Coevoet (<https://github.com/stof>) ha creado el *bundle* StofDoctrineExtensionsBundle (<http://www.symfony.es/bundles/stof/stofdoctrineextensionsbundle/>) . Así que para traducir los contenidos de la base de datos, primero debes instalar un *bundle* y varias extensiones de Doctrine2.

10.4.1 Instalando las extensiones de Doctrine2

Abre el archivo `composer.json` que se encuentra en el directorio raíz del proyecto y añade la siguiente nueva dependencia:

```
{  
    "require": {  
        "stof/doctrine-extensions-bundle": "dev-master"  
    }  
}
```

Instala los nuevos componentes ejecutando el comando habitual para actualizar los *vendors*:

```
$ composer update
```

NOTA Para que te funcione este comando, debes instalar Composer globalmente (página 46) en tu ordenador, tal y como se explicó en el capítulo 3.

Por último, activa el *bundle* dentro del *kernel* de la aplicación:

```
// app/AppKernel.php  
class AppKernel extends Kernel  
{  
    public function registerBundles()  
    {  
        $bundles = array(  
            // ...  
            new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),  
        );  
  
        // ...  
    }  
  
    // ...  
}
```

Las extensiones y el nuevo *bundle* ya están correctamente instalados y activados. Si al ejecutar ahora la aplicación se muestra algún error, borra la caché con el comando `php app/console cache:clear`.

Para no penalizar el rendimiento de la aplicación, por defecto las nuevas extensiones están desactivadas. Por tanto, abre el archivo de configuración general de la aplicación y añade lo siguiente:

```
# app/config/config.yml

# ...
stof_doctrine_extensions:
    default_locale:      es
    translationFallback: true
    orm:
        default:
            translatable: true
```

La configuración de las extensiones de Doctrine2 se realiza bajo la clave `stofDoctrineExtensions`. Las dos opciones principales para controlar el comportamiento de la extensión `Translatable` son `defaultLocale` y `translationFallback`.

El valor de la opción `defaultLocale` (cuyo valor por defecto es `en`) indica el *locale* que utiliza la extensión cuando un contenido no está disponible en el *locale* solicitado por el usuario. Por su parte, si la opción `translationFallback` es `false` (su valor por defecto) no se muestra nada cuando la traducción no está disponible en el idioma solicitado.

Por último, para activar la extensión `Translatable` se establece la opción `translatable` a `true`. Las opciones `orm` y `default` indican el *entity manager* sobre el que funciona la extensión. A menos que utilices una configuración muy avanzada, estos son los valores que debes utilizar siempre.

10.4.2 Actualizando las entidades y los archivos de datos

Las entidades internacionalizadas deben traducir los contenidos de algunas propiedades (nombres, descripciones, etc.) pero no es necesario que modifiquen los valores de muchas otras (id, fechas, número de compras, etc.)

Haciendo uso de las anotaciones, la extensión `Translatable` permite indicar las propiedades específicas que se traducen en cada entidad. El siguiente ejemplo muestra los cambios necesarios en la entidad `Oferta` para traducir el nombre y la descripción:

```
// src/Cupon/OfertaBundle/Entity/Oferta.php
namespace Cupon\OfertaBundle\Entity;

// ...

use Gedmo\Mapping\Annotation as Gedmo;
use Gedmo\Translatable\Translatable;

/**
 * @ORM\Entity(repositoryClass="Cupon\OfertaBundle\Entity\OfertaRepository")
 */
class Oferta
{
```

```
// ...

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

/**
 * @ORM\Column(type="text")
 * @Gedmo\Translatable
 * @Assert\NotBlank()
 * @Assert\MinLength(30)
 */
protected $descripcion;

// ...

/**
 * @Gedmo\Locale
 */
private $locale;

public function setTranslatableLocale($locale)
{
    $this->locale = $locale;
}
}
```

Primero se importan las clases necesarias mediante la instrucción `use` y se define el nuevo prefijo `@Gedmo\` para las anotaciones. Después sólo tienes que añadir la anotación `@Gedmo\Translatable` en todas las propiedades cuyos contenidos se traduzcan a los diferentes idiomas.

Añade también una propiedad con la anotación `@Gedmo\Locale`. Esta propiedad no se guarda en la base de datos, por lo que no es necesario que le añadas anotaciones de tipo `@ORM\`. El nombre de la propiedad también lo puedes elegir libremente, pero se recomienda utilizar el nombre `$locale` para indicar claramente que se trata de la variable que utiliza la extensión para gestionar el idioma de los contenidos.

Por último, añade un método llamado `setTranslatableLocale()` que permita modificar el valor de la propiedad que se acaba de explicar y por tanto, que permita modificar el `locale` de la entidad.

Para su correcto funcionamiento, la extensión `Translatable` necesita crear dos tablas nuevas en la base de datos, llamadas `ext_log_entries` y `ext_translations`. En el momento de escribir este libro, la estructura de cada tabla es la siguiente:

```

CREATE TABLE ext_log_entries (id INT AUTO_INCREMENT NOT NULL, action VARCHAR(8)
NOT NULL, logged_at DATETIME NOT NULL, object_id VARCHAR(32) DEFAULT NULL,
object_class VARCHAR(255) NOT NULL, version INT NOT NULL, DATA LONGTEXT DEFAULT
NULL COMMENT '(DC2Type:array)', username VARCHAR(255) DEFAULT NULL, INDEX
log_class_lookup_idx (object_class), INDEX log_date_lookup_idx (logged_at),
INDEX log_user_lookup_idx (username), PRIMARY KEY(id)) ENGINE = InnoDB;

CREATE TABLE ext_translations (id INT AUTO_INCREMENT NOT NULL, locale
VARCHAR(8) NOT NULL, object_class VARCHAR(255) NOT NULL, FIELD VARCHAR(32) NOT
NULL, foreign_key VARCHAR(64) NOT NULL, content LONGTEXT DEFAULT NULL, INDEX
translations_lookup_idx (locale, object_class, foreign_key), UNIQUE INDEX
lookup_unique_idx (locale, object_class, foreign_key, FIELD), PRIMARY KEY(id))
ENGINE = InnoDB;

```

Si es posible, lo mejor es borrar la base de datos y volver a crearla:

```

// Borrar la base de datos ( cuidado con este comando!)
$ php app/console doctrine:database:drop --force

// Crear la base de datos
$ php app/console doctrine:database:create

// Crear la estructura de tablas
$ php app/console doctrine:schema:create

```

Antes de cargar los archivos de datos o *fixtures*, añade los siguientes cambios en el archivo de datos de ofertas para traducir sus contenidos al inglés:

```

// src/Cupon/OfertaBundle/DataFixtures/ORM/Ofertas.php
class Ofertas extends AbstractFixture implements OrderedFixtureInterface,
ContainerAwareInterface
{
    // ...

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

        for ($i=0; $i<count($ciudades); $i++) {
            for ($j=1; $j<=20; $j++) {
                $oferta = new Oferta();

                $oferta->setNombre(...);
                $oferta->setDescripcion(...);
                $oferta->setCondiciones(...);
                $oferta->setRutaFoto(...);

                // ...
            }
        }
    }
}

```

```
$oferta->setTienda($tienda);

$manager->persist($oferta);
$manager->flush();

// Traducir los contenidos de la oferta al inglés
$id = $oferta->getId();
$offer = $manager->find('OfertaBundle:Oferta', $id);
$offer->setNombre('ENGLISH ' . $oferta->getNombre());
$offer->setDescripcion('ENGLISH ' . $oferta->getDescripcion());
$offer->setTranslatableLocale('en');

$manager->persist($offer);
$manager->flush();

// ...
}

}

// ...
}
```

La traducción de una entidad siempre se realiza a partir de la entidad en el idioma original. Después de guardar la oferta original mediante `persist()` y `flush()`, se busca la misma oferta con el método `find()` del *entity manager*.

Para traducir las propiedades `nombre` y `descripcion` al inglés, simplemente establece sus nuevos valores mediante los *getters* de la entidad. Antes de guardar los cambios, no olvides indicar el idioma de la nueva entidad mediante el método `setTranslatableLocale()` que se añadió anteriormente. Cuando se realice el `flush()` del nuevo objeto, la extensión *Translatable* se encarga de guardar la traducción en sus tablas, por lo que la entidad original no se ve afectada. A continuación, carga los datos de prueba con el siguiente comando:

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

Si ahora pruebas a acceder a cualquier página del sitio web, no verás ningún cambio en la información de las ofertas. Si cambias el idioma del sitio al inglés, automáticamente cambiará el nombre y la descripción de todas las ofertas. Si has utilizado el mismo código mostrado anteriormente, el nombre y la descripción de las ofertas incluirán la palabra *ENGLISH* por delante, indicando que se está mostrando su traducción, no su contenido original.

Lo mejor de la extensión *Translatable* es que gestiona las traducciones de forma transparente para el programador. No obstante, si necesitas un control preciso del proceso de traducción, puedes utilizar el código de los siguientes ejemplos.

```
// Cargar la misma entidad pero en otro idioma
$oferta = $em->find('OfertaBundle:Oferta', $idOferta);
```

```

$oferta->setLocale('en');
$em->refresh($oferta);

// Obtener todas las traducciones de la entidad
$oferta = $em->find('OfertaBundle:Oferta', $idOferta);
$repository = $em->getRepository('Gedmo\Translatable\Entity\Translation');
$traducciones = $repository->findTranslations($oferta);

/* $traducciones es un array con la siguiente estructura:
Array (
    [es] => Array (
        [nombre]      => ...
        [descripcion] => ...
    )
    [en] => Array (
        [nombre]      => ...
        [descripcion] => ...
    )
) */

```

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. Además, gracias a la flexibilidad de los *bundles* de Symfony2, puedes crear una carpeta para guardar las páginas de cada *locale* o idioma.

A continuación se muestra cómo traducir las páginas estáticas creadas en el capítulo 4 (página 69). Crea un directorio llamado `es/` dentro del directorio `Sitio/` del *bundle OfertaBundle* y copia en su interior todas las páginas estáticas:

```

src/
└ Cupon/
  └ OfertaBundle/
    └ 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):

```

src/
└ Cupon/

```

```

└─ OfertaBundle/
    └─ 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/Cupon/OfertaBundle/Controller/DefaultController.php
namespace Cupon\OfertaBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller;

class SitioController extends Controller
{
    public function estaticaAction($pagina)
    {
        return $this->render(sprintf(
            'OfertaBundle:Sitio:%s/%s.html.twig',
            $this->getRequest()->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

Otra de las carencias más significativas de Symfony2 es que la versión que incluye de Twig no soporta la internacionalización de las fechas. Así que los usuarios que acceden al sitio en inglés, verán todos los contenidos traducidos salvo las fechas, como por ejemplo la fecha de expiración de una oferta.

Afortunadamente, las versiones más recientes de Twig incluyen una extensión para traducir fechas, aunque por defecto está desactivada. Añade en primer lugar 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 2013* 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: `9/21/12` en inglés y `21/09/12` 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, 2012` en inglés y `21/09/2012` en español.
 - Hora: `11:59:59 PM GMT+02:00` en inglés y `23:59:59 GMT+02:00` en español.
- `IntlDateFormatter::LONG`:
 - Fecha: `September 21, 2012` en inglés y `21 de septiembre de 2012` en español.
 - Hora: `11:59:59 PM GMT+02:00` en inglés y `23:59:59 GMT+02:00` en español.
- `IntlDateFormatter::FULL`:
 - Fecha: `Saturday, September 21, 2012` en inglés y `sábado 21 de septiembre de 2012` 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 quieras 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)
        );

        // ...
    }
}
```

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*").

Los tests se dividen 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 Symfony2 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 1 disponía de una herramienta propia para crear y ejecutar los tests. Symfony2 ha optado por utilizar la librería PHPUnit (<http://www.phpunit.de>), que prácticamente se ha convertido en un estándar en el mundo PHP. De esta forma, los tests unitarios y funcionales de Symfony2 combinan la potencia de PHPUnit con las utilidades y facilidades proporcionadas por Symfony2.

Antes de crear los primeros tests, asegúrate de tener instalada como mínimo la versión 3.5.11 de PHPUnit. Para ello, ejecuta el comando `phpunit --version` en la consola de comandos:

```
$ phpunit --version  
PHPUnit 3.7.18 by Sebastian Bergmann.
```

Si se muestra un mensaje similar al anterior, ya puedes continuar. Si se muestra un mensaje de error, instala PHPUnit mediante PEAR con los siguientes comandos:

```
$ pear config-set auto_discover 1  
$ pear install pear.phpunit.de/PHPUnit
```

Si se produce algún error, actualiza primero PEAR con el comando `pear upgrade PEAR`. Si aún así se siguen produciendo errores, consulta la sección sobre instalación del manual de PHPUnit (<http://www.phpunit.de/manual/current/en/installation.html>) .

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 Symfony2 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 del *bundle* `Oferta`, su test debería crearse en `src/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php`.

Cuando se genera un *bundle*, Symfony2 crea automáticamente un pequeño test de ejemplo para el controlador por defecto. Así, por ejemplo, en el *bundle* `OfertaBundle` puedes encontrar el siguiente test llamado `DefaultControllerTest.php`:

```
// src/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php  
namespace Cupon\OfertaBundle\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")')->count() > 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/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php`. Para ello, crea el siguiente archivo y copia su contenido:

```
// src/Cupon/OfertaBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace Cupon\OfertaBundle\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.

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 3.7.18 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 3.7.18 by Sebastian Bergmann.

F

Time: 1 second, Memory: 1.50Mb
```

```
There was 1 failure:

1) Cupon\OfertaBundle\Tests\Twig\Extension\TwigExtensionTest::testDescuento
Probar que 1 es igual a 2
Failed asserting that <integer:2> matches expected <integer:1>.

/.../src/Cupon/OfertaBundle/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 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:

```
// src/Cupon/OfertaBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace Cupon\OfertaBundle\Tests\Twig\Extension;

use Cupon\OfertaBundle\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($valor Esperado, $valor Obtenido, $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 3.7.18 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 (`` o ``). 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:

```
// src/Cupon/OfertaBundle/Tests/Twig/Extension/CuponExtensionTest.php
namespace Cupon\OfertaBundle\Tests\Twig\Extension;

use Cupon\OfertaBundle\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/Cupon/OfertaBundle/Tests/Twig/Extension/fixtures/lista/original.txt
Primer elemento
Segundo elemento
Tercer elemento

// src/Cupon/OfertaBundle/Tests/Twig/Extension/fixtures/lista/esperado-ul.txt
<ul>
    <li>Primer elemento</li>
    <li>Segundo elemento</li>
    <li>Tercer elemento</li>
</ul>

// src/Cupon/OfertaBundle/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 Symfony2, para lo cual puedes utilizar el siguiente código:

```
// src/Cupon/OfertaBundle/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/Cupon/OfertaBundle/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/Cupon/OfertaBundle/Tests/Entity/OfertaTest.php
namespace Cupon\OfertaBundle\Tests\Entity;

use Symfony\Component\Validator\Validation;
use Cupon\OfertaBundle\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->getMessage());
    $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
publicación'
```

```

    );

        $error = $listaErrores[0];
        $this->assertEquals('La fecha de expiración debe ser posterior a la
fecha 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->getMessageTemplate());
        $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 crearán 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/Cupon/OfertaBundle/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/Cupon/OfertaBundle/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 OfertaBundle*. Como se explicó anteriormente, cuando se genera un *bundle*, Symfony2 crea un test de prueba para su controlador por defecto. Así que abre el archivo `src/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php` existente y sustituye su contenido por lo siguiente:

```
// src/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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/Cupon/OfertaBundle/

# Ejecuta sólo los test del controlador del bundle Oferta
$ phpunit -c app src/Cupon/OfertaBundle/Tests/Controller/

# Ejecuta sólo los test de la extensión de Twig
$ phpunit -c app src/Cupon/OfertaBundle/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:

```
$peticion = $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 Symfony2, 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 quieras 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/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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. Symfony2 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/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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(
            'cupon.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 `cupon.ciudad_por_defecto`:

```
$ciudadPorDefecto = $client->getContainer()->getParameter(
    'cupon.ciudad_por_defecto'
);
```

NOTA A través del contenedor también puedes acceder al servicio `router`, de forma que puedas generar las URL del método `request()` utilizando las rutas de la aplicación:

```
$client = static::createClient();
$url = $client->getContainer()->get('router')->generate('oferta',
array(
    '_locale' => 'es',
    'ciudad'   => 'madrid',
    'slug'     => 'lorem-ipsum-dolor-sit-amet'
));
$client->request('GET', $url);
```

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/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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. Symfony2 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 Symfony2 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/Cupon/OfertaBundle/Tests/Controller/DefaultControllerTest.php
namespace Cupon\OfertaBundle\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 llenar formularios y enviarlos. De hecho, Symfony2 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-space(string(@value)), ' '), ' Registrarme ')) or (@type="image" and contains(concat(' ', normalize-space(string(@alt)), ' '), ' Registrarme ')) or @id="Registrarme" or @name="Registrarme"] | //button[contains(concat(' ', normalize-space(string(.)), ' '), ' 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 Symfony2, el atributo `name` de cada campo del formulario sigue la nomenclatura `nombre-formulario[nombre-propiedad]`.

El nombre del formulario se establece con el método `getName()` de su clase. Para simplificar el código de los tests, es recomendable indicar un nombre corto y significativo. Para el formulario de registro de usuarios que se muestra en el *frontend*, un nombre adecuado podría ser `frontend_usuario`:

```

// src/Cupon/UsuarioBundle/Form/Frontend/UsuarioType.php
namespace Cupon\UsuarioBundle\Form\Frontend;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        // ...
    }

    public function getName()
    {
        return 'frontend_usuario';
    }
}

```

Si utilizas el código anterior, el nombre de los campos del formulario de registro de usuarios sería:

```

frontend_usuario[nombre]
frontend_usuario[apellidos]
frontend_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:

```
frontend_usuario[fecha_nacimiento][day]
frontend_usuario[fecha_nacimiento][month]
frontend_usuario[fecha_nacimiento][year]
```

Por su parte, a los campos de tipo *repeated* (como por ejemplo la contraseña) se les asigna los siguientes nombres:

```
frontend_usuario[password][first]
frontend_usuario[password][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/Cupon/OfertaBundle/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/Cupon/UsuarioBundle/Tests/Controller/DefaultControllerTest;
namespace Cupon\UsuarioBundle\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/Cupon/UsuarioBundle/Tests/Controller/DefaultControllerTest;
namespace Cupon\UsuarioBundle\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generalUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...
    }

    public function generalUsuarios()
    {
        return array(
            array(
                array(
                    'frontend_usuario[nombre]' => 'Anónimo',
                    'frontend_usuario[apellidos]' => 'Apellido1 Apellido2',
                    'frontend_usuario[email]' =>
                        'anonimo'.uniqid().'@localhost.localedomain',
                    'frontend_usuario[password][first]' => 'anonimo1234',
                    'frontend_usuario[password][second]' => 'anonimo1234',
                    'frontend_usuario[direccion]' => 'Calle ...',
                    'frontend_usuario[dni]' => '11111111H',
                    'frontend_usuario[numero_tarjeta]' => '123456789012345',
                    'frontend_usuario[ciudad]' => '1',
                    'frontend_usuario[permite_email]' => '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.localedomain'
```

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/Cupon/UsuarioBundle/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/Cupon/UsuarioBundle/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')->getValue(),
        '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 llenar 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 quieras 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/Cupon/UsuarioBundle/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/Cupon/UsuarioBundle/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['frontend_usuario[email]'],
        $crawler->filter(
            'form input[name="frontend_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/Cupon/UsuarioBundle/Tests/Controller/DefaultControllerTest;
class DefaultControllerTest extends WebTestCase
{
    /**
     * @dataProvider generaUsuarios
     */
    public function testRegistro($usuario)
    {
        // ...

        $usuario = $this->em->getRepository('UsuarioBundle:Usuario')
            ->findOneByEmail($usuario['frontend_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 Symfony2

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 Symfony2, 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`. Symfony2 ya incluye un archivo de configuración adecuado en `app/phpunit.xml.dist` con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- http://www.phpunit.de/manual/current/en/appendices.configuration.html -->
<phpunit
    backupGlobals          = "false"
    backupStaticAttributes = "false"
    colors                = "true"
    convertErrorsToExceptions = "true"
    convertNoticesToExceptions = "true"
    convertWarningsToExceptions = "true"
    processIsolation      = "false"
    stopOnFailure          = "false"
    syntaxCheck            = "false"
    bootstrap              = "bootstrap.php.cache" >

    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/*/*Bundle/*Bundle/Tests</directory>
        </testsuite>
    </testsuites>

    <!--
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    -->

    <filter>
        <whitelist>
            <directory>../src</directory>
            <exclude>
                <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 Symfony2 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 a dos únicos *bundles* (`OfertaBundle` y `UsuarioBundle`):

```
<!-- app/phpunit.xml -->
...
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/Cupon/OfertaBundle/Tests</directory>
        <directory>../src/Cupon/UsuarioBundle/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 3.7.18 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 *assertión*, 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 - Cupon\OfertaBundle\Tests\Controller\DefaultControllerTest::testPortada
ok 2 - Cupon\OfertaBundle\Tests\Entity\OfertaTest::testValidacion
ok 3 - Cupon\OfertaBundle\Tests\Twig\Extension\TwigExtensionTest::testDescuento
ok 4 - Cupon\OfertaBundle\Tests\Twig\Extension\
```

```
TwigExtensionTest::testMostrarComoLista
ok 5 - Cupon\UsuarioBundle\Tests\Controller\DefaultControllerTest::testRegistro
with data set #0 (array('Anónimo', 'Apellido1 Apellido2',
'anonimo4e672de965aff@localhost.localdomain', 'anonymo1234', 'anonymo1234', 'Mi
calle, Mi ciudad, 01001', '01', '01', '1970', '11111111H', '123456789012345',
'1', '1'))
1..5
```

También puedes hacer uso de la opción `--testdox`:

```
$ phpunit -c app --testdox

PHPUnit 3.7.18 by Sebastian Bergmann.

Cupon\OfertaBundle\Tests\Controller\DefaultController
 [x] Portada

Cupon\OfertaBundle\Tests\Entity\Oferta
 [x] Validacion

Cupon\OfertaBundle\Tests\Twig\Extension\TwigExtension
 [x] Descuento
 [x] Mostrar como lista

Cupon\UsuarioBundle\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 <http://www.phpunit.de/manual/current/en/>

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 Symfony2 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* [TiendaBundle](#), 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* [TiendaBundle](#).
- Crear un nuevo directorio llamado [Extranet/](#) dentro del directorio [Resources/views/](#) del *bundle* [TiendaBundle](#).
- 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 nuevo archivo de enrutamiento llamado `routing.yml` en el directorio `Resources/config/extranet/` del *bundle* `TiendaBundle` e importarlo desde el archivo `app/config/routing.yml`.
- 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.

Crea el directorio `Resources/config/extranet/` dentro del *bundle* `TiendaBundle` y añade un archivo llamado `routing.yml` con el siguiente contenido:

```
# src/Cupon/TiendaBundle/Resources/config/extranet/routing.yml

extranet_portada:
    path:      /
    defaults: { _controller: TiendaBundle:Extranet:portada }

extranet_oferta_nueva:
    path:      /oferta/nueva
    defaults: { _controller: TiendaBundle:Extranet:ofertaNueva }

extranet_oferta_editar:
    path:      /oferta/editar/{id}
    defaults: { _controller: TiendaBundle:Extranet:ofertaEditar }

extranet_oferta_ventas:
    path:      /oferta/ventas/{id}
    defaults: { _controller: TiendaBundle:Extranet:ofertaVentas }

extranet_perfil:
    path:      /perfil
    defaults: { _controller: TiendaBundle:Extranet:perfil }
```

Observa cómo el patrón de las rutas no incluye el valor `/extranet`. Cuando todas las rutas comienzan de la misma manera, es mejor utilizar la opción `prefix` al importar las rutas del *bundle*.

Abre el archivo de enrutamiento general de Symfony2 y añade lo siguiente para importar todas las rutas de la *extranet* y prefijarles el valor `/extranet`:

```
# app/config/routing.yml
#
# ...
#
Extranet:
    resource: '@TiendaBundle/Resources/config/extranet/routing.yml'
    prefix: /extranet
```

Si la *extranet* estuviera disponible en varios idiomas, también habría que añadir la variable especial `{_locale}` en el prefijo de las rutas.

12.3 Layout

Si recuerdas la sección *herencia de plantillas a tres niveles* (página 165) 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('bundles/oferta/css/normalizar.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/oferta/css/comun.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/tienda/css/extranet.css') }}"
          rel="stylesheet" type="text/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
oferta</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`. Como no se indica ni el *bundle* ni el directorio, Symfony2 la busca en el directorio `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 llenar 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('bundles/oferta/css/normalizar.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/oferta/css/comun.css') }}"
          rel="stylesheet" type="text/css" />
    <link href="{{ asset('bundles/tienda/css/extranet.css') }}"
          rel="stylesheet" type="text/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 `Resources/public/css/` del *bundle* `TiendaBundle`.

NOTA Para ver las páginas de la *extranet* con un mejor aspecto, puedes copiar los contenidos de la hoja de estilos `extranet.css` que se encuentra en el repositorio: <https://github.com/javiereguiluz/Cupon/blob/2.3/src/Cupon/TiendaBundle/Resources/public/css/extranet.css>

No olvides que para que las hojas de estilo y el resto de archivos web (JavaScript e imágenes) se vean en el sitio web, antes debes *instalarlas* con el comando:

```
$ php app/console assets:install
```

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  
oferta</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 Symfony2 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:
                key:         cupon1234
                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: Cupon\UsuarioBundle\Entity\Usuario,
               property: email }

encoders:
    Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512,
                                           iterations: 10 }

```

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: Cupon\UsuarioBundle\Entity\Usuario,
                  property: email }
    tiendas:
        entity: { class: Cupon\TiendaBundle\Entity\Tienda,
                  property: login }

# ...

```

El proveedor `tiendas` crea los usuarios a partir de la entidad `Tienda` del *bundle TiendaBundle* 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` la forma en la que se codifican las contraseñas de las tiendas:

```
# 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: Cupon\TiendaBundle\Entity\Tienda,
                      property: login }

    encoders:
        Cupon\UsuarioBundle\Entity\Usuario: { algorithm: sha512,
                                              iterations: 10 }
        Cupon\TiendaBundle\Entity\Tienda:   { algorithm: sha512,
                                              iterations: 10 }
```

Recuerda que si no defines un valor para la opción `iterations`, Symfony2 codifica cada contraseña 5.000 veces consecutivas utilizando el algoritmo SHA 512. Como este proceso consume un tiempo no despreciable, se reduce a 10 las veces que se codifica cada contraseña.

13.2 Preparando el proveedor de usuarios de las tiendas

Si recuerdas la sección *Creando proveedores de usuarios* (página 217) del capítulo 8, convertir una entidad de Doctrine2 en un proveedor de usuarios es tan sencillo como implementar la interfaz `UserInterface`.

Abre el archivo de la entidad `Tienda` del *bundle TiendaBundle* y añade los siguientes cambios:

```
// src/Cupon/TiendaBundle/Entity/Tienda.php
use Symfony\Component\Security\Core\User\UserInterface;

class Tienda implements UserInterface
{
    function eraseCredentials()
    {
    }

    function getRoles()
    {
        return array('ROLE_TIENDA');
```

```

    }

    function getUsername()
    {
        return $this->getLogin();
    }

    // Los métodos getSalt() y getPassword() ya existían 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 los métodos `getSalt()` y `getPassword()` porque ya existían, ya que la entidad `Tienda` dispone de las propiedades `salt` y `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/Cupon/TiendaBundle/DataFixtures/ORM/Tiendas.php
namespace Cupon\TiendaBundle\DataFixtures\ORM;

// ...
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class Tiendas extends AbstractFixture implements OrderedFixtureInterface,
ContainerAwareInterface
{
    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);
    $tienda->setSalt(md5(time()));

    $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 la autenticación mediante un formulario de login.

Symfony2 se encarga de gestionar internamente la lógica que comprueba el usuario y contraseña introducidos ([login_check](#)) y la desconexión del usuario ([logout](#)). Así que sólo es necesario crear la acción y el formulario en el que el usuario introduce sus credenciales.

Añade en primer lugar las siguientes tres rutas en el archivo de enrutamiento de la *extranet*:

```
# src/Cupon/TiendaBundle/Resources/config/extranet/routing.yml
#
# ...
```

```

extranet_login:
    pattern: /login
    defaults: { _controller: TiendaBundle:Extranet:login }

extranet_login_check:
    pattern: /login_check

extranet_logout:
    pattern: /logout

```

La ruta `extranet_login` está asociada con la acción `loginAction()` del controlador `ExtranetController` de `TiendaBundle`. Su código es el habitual de los formularios de login:

```

// src/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class ExtranetController extends Controller
{
    public function loginAction()
    {
        $peticion = $this->getRequest();
        $sesion = $peticion->getSession();

        $error = $peticion->attributes->get(
            SecurityContext::AUTHENTICATION_ERROR,
            $sesion->get(SecurityContext::AUTHENTICATION_ERROR)
        );

        return $this->render('TiendaBundle:Extranet:login.html.twig', array(
            'error' => $error
        ));
    }
}

```

Por último, crea el formulario de login en la plantilla `login.html.twig` del directorio `Resources/views/Extranet/` del bundle `TiendaBundle`:

```

{% # src/Cupon/TiendaBundle/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://cupon.local/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 225) del capítulo 8 utiliza el evento `security.interactive_login` para redirigir a cada usuario a la portada de su ciudad después del login:

```

// src/Cupon/UsuarioBundle/Listener/LoginListener.php
namespace Cupon\UsuarioBundle\Listener;

use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Routing\Router;

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) {
        $portada = $this->router->generate('portada', array(
            'ciudad' => $this->ciudad
        ));
        $event->setResponse(new RedirectResponse($portada));
    }
}
}

```

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 *role* `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 Symfony2:

```

if ($this->get('security.context')->isGranted('ROLE_TIENDA')) {
    // El usuario es de tipo Tienda
}
elseif ($this->get('security.context')->isGranted('ROLE_USUARIO')) {
    // El usuario es de tipo Usuario
}

```

Como el método `isGranted()` requiere acceder al contexto de seguridad, primero debes modificar la definición del servicio `login_listener` del *bundle* `UsuarioBundle` para inyectar el servicio `@security.context` como argumento:

```

# src/Cupon/UsuarioBundle/Resources/config/services.yml
services:
    login_listener:
        class: Cupon\UsuarioBundle\Listener\LoginListener
        arguments: [@security.context, @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 `SecurityContext` y añade la lógica necesaria para determinar la página a la que se redirecciona al usuario en función de su tipo:

```
// src/Cupon/UsuarioBundle/Listener/LoginListener.php  
namespace Cupon\UsuarioBundle\Listener;  
  
use Symfony\Component\EventDispatcher\Event;  
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;  
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;  
use Symfony\Component\Security\Core\SecurityContext;  
use Symfony\Component\Routing\Router;  
use Symfony\Component\HttpFoundation\RedirectResponse;  
  
class LoginListener  
{  
    private $contexto, $router, $ciudad = null;  
  
    public function __construct(SecurityContext $context, Router $router)  
    {  
        $this->contexto = $context;  
        $this->router = $router;  
    }  
  
    public function onSecurityInteractiveLogin(InteractiveLoginEvent $event)  
    {  
        $token = $event->getAuthenticationToken();  
        $this->ciudad = $token->getUser()->getCiudad()->getSlug();  
    }  
  
    public function onKernelResponse(FilterResponseEvent $event)  
    {  
        if (null != $this->ciudad) {  
            if($this->contexto->isGranted('ROLE_TIENDA')) {  
                $portada = $this->router->generate('extranet_portada');  
            }  
            else {  
                $portada = $this->router->generate('portada', array(  
                    'ciudad' => $this->ciudad  
                ));  
            }  
  
            $event->setResponse(new RedirectResponse($portada));  
            $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 Symfony2 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* propios son clases PHP que implementan la interfaz `VoterInterface`. Esta interfaz obliga a definir los siguientes tres métodos:

```
interface VoterInterface
{
    public function supportsAttribute($attribute);
    public function supportsClass($class);
    public function vote(TokenInterface $token, $object, array $attributes);
}
```

El método `supportsAttribute()` es el encargado de restringir el alcance de cada *voter*. El argumento que se le pasa es el nombre del permiso que está solicitando el usuario (por ejemplo, `ROLE_EDITAR_OFERTA`). Si el permiso solicitado no tiene nada que ver con el *voter*, su respuesta será una abstención, ya que el *voter* no tiene información suficiente para decidir si el usuario tiene o no permiso.

El método `supportsClass()` simplemente comprueba si el *voter* soporta la clase que representa al usuario que solicita el permiso. Como casi siempre se utiliza la clase de usuario que proporciona Symfony2, el contenido de este método suele ser `return true;`.

El método `vote()` es el más importante, ya que incluye toda la lógica propia del *voter* que le permitirá tomar la decisión. 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 `vote()` es el `$token` que permite acceder directamente a los datos del usuario que solicita el permiso. El segundo parámetro es el objeto sobre el que se solicita el permiso (por ejemplo, la oferta que se quiere modificar). El tercer parámetro es un array con los permisos solicitados por el usuario.

La respuesta devuelta por el método `vote()` debe ser uno de los siguientes tres valores definidos como constantes:

- `VoterInterface::ACCESS_GRANTED`, se autoriza el acceso de este usuario al recurso solicitado.
- `VoterInterface::ACCESS_ABSTAIN`: el *voter* no puede tomar una decisión y por tanto, se abstiene.
- `VoterInterface::ACCESS_DENIED`: no se autoriza el acceso del usuario al recurso solicitado.

13.4.2 Creando un *security voter* propio

Las buenas prácticas en el desarrollo de aplicaciones Symfony2 recomiendan definir los *voters* propios en el directorio `Security` de algún *bundle*. Como el *voter* comprueba si una tienda puede modificar una oferta, es mejor crearlo en el *bundle* `TiendaBundle`. Además, resulta común denominar `OwnerVoter` a este tipo de *voters* que comprueban si un usuario puede realizar alguna operación sobre un determinado objeto.

Con todo lo anterior, crea un nuevo archivo llamado `src/Cupon/TiendaBundle/Security/OwnerVoter.php` con el siguiente contenido inicial:

```
// src/Cupon/TiendaBundle/Security/OwnerVoter.php
namespace Cupon\TiendaBundle\Security;

use Symfony\Component\Security\Core\Authorization\Voter\VoterInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

class OwnerVoter implements VoterInterface
{
    public function supportsAttribute($attribute)
    {

    }

    public function supportsClass($class)
    {
    }
}
```

```

    public function vote(TokenInterface $token, $object, array $attributes)
    {
    }
}

```

El contenido del método `supportsAttribute()` es muy sencillo para este *voter*, ya que sólo se ocupa de un permiso llamado `ROLE_EDITAR_OFERTA`:

```

// src/Cupon/TiendaBundle/Security/OwnerVoter.php

class OwnerVoter implements VoterInterface
{
    public function supportsAttribute($attribute)
    {
        return 'ROLE_EDITAR_OFERTA' == $attribute;
    }

    // ...
}

```

Si observas por ejemplo el código del *voter* `RoleVoter` definido por Symfony2, verás una lógica muy similar aunque algo menos restrictiva, ya que ese *voter* funciona para cualquier permiso que empiece por `ROLE_`:

```

// Symfony\Component\Security\Core\Authorization\Voter\RoleVoter.php;

class RoleVoter implements VoterInterface
{
    public function supportsAttribute($attribute)
    {
        return 0 === strpos($attribute, $this->prefix);
    }

    // ...
}

```

El código del método `supportsClass()` es trivial ya que, como se explicó anteriormente, este *voter* hace uso de la clase de usuario por defecto de Symfony2 y por tanto sólo tiene que devolver `true`:

```

// src/Cupon/TiendaBundle/Security/OwnerVoter.php

class OwnerVoter implements VoterInterface
{
    public function supportsClass($class)
    {
        return true;
    }
}

```

```
// ...  
}
```

El código del método `vote()` es el más cambiante de un *voter* a otro, ya que implementa la lógica específica del *voter*. Aún así, la estructura de su código suele ser similar:

1. Recorrer el array de los permisos solicitados (mediante el tercer atributo del método, `$attributes`).
2. Comprobar para cada permiso solicitado si este es el *voter* adecuado para tomar la decisión (mediante el método `supportsAttribute()`).
3. Si este *voter* no puede decidir sobre ninguno de los permisos, abstenerse devolviendo la respuesta `ACCESS_ABSTAIN`.
4. Si el *voter* puede decidir sobre un determinado permiso, ejecutar la lógica necesaria para permitir el acceso (devolviendo `ACCESS_GRANTED`) o para denegarlo (devolviendo `ACCESS_DENIED`).

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/Cupon/TiendaBundle/Security/OwnerVoter.php  
  
class OwnerVoter implements VoterInterface  
{  
    public function vote(TokenInterface $token, $object, array $attributes)  
    {  
        // ...  
  
        if ($object->getTienda()->getId() === $user->getId()) {  
            return VoterInterface::ACCESS_GRANTED;  
        } else {  
            return VoterInterface::ACCESS_DENIED;  
        }  
  
        // ...  
    }  
}
```

A continuación se muestra el código completo del *voter* resultante:

```
// src/Cupon/TiendaBundle/Security/OwnerVoter.php  
namespace Cupon\TiendaBundle\Security;  
  
use Symfony\Component\Security\Core\Authorization\Voter\VoterInterface;  
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;  
  
class OwnerVoter implements VoterInterface  
{
```

```

public function supportsAttribute($attribute)
{
    return 'ROLE_EDITAR_OFERTA' == $attribute;
}

public function supportsClass($class)
{
    return true;
}

public function vote(TokenInterface $token, $object, array $attributes)
{
    foreach ($attributes as $attribute) {
        if (false === $this->supportsAttribute($attribute)) {
            continue;
        }

        $user = $token->getUser();

        // comprobar que la oferta que se edita fue publicada
        // por esta misma tienda
        if ($object->getTienda()->getId() === $user->getId()) {
            return VoterInterface::ACCESS_GRANTED;
        } else {
            return VoterInterface::ACCESS_DENIED;
        }
    }

    return VoterInterface::ACCESS_ABSTAIN;
}
}

```

El último paso para activar este *voter* propio consiste en definir un nuevo servicio y etiquetarlo con la etiqueta especial `security.voter`. En nuestro caso, la forma más sencilla de activar este servicio es añadirlo en el archivo `app/config/services.yml` creado en los capítulos anteriores:

```

# app/config/services.yml
services:
    # ...

    cupon.tiendabundle.security.ownervoter:
        class: Cupon\TiendaBundle\Security\OwnerVoter
        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:
    # ...

    cupon.tiendabundle.security.ownervoter:
        class: Cupon\TiendaBundle\Security\OwnerVoter
        arguments: ['@doctrine.orm.entity_manager', '@service_container']
        tags:
            - { name: security.voter }
```

13.4.3 Comprobando los permisos

Una vez creado y activado el *security voter* propio, resulta 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\SecurityContext;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// ...

public function ofertaEditarAction($id)
{
    $oferta = $em->getRepository('OfertaBundle:Oferta')->find($id);

    if (false === $this->get('security.context')->isGranted(
        'ROLE_EDITAR_OFERTA', $oferta
    )) {
        throw new AccessDeniedException();
    }

    // ...
}
```

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/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExtranetController extends Controller
{
    // ...

    public function portadaAction()
    {
        $em = $this->getDoctrine()->getManager();

        $tienda = $this->get('security.context')->getToken()->getUser();
        $ofertas = $em->getRepository('TiendaBundle:Tienda')
            ->findOfertasRecientes($tienda->getId());

        return $this->render('TiendaBundle:Extranet:portada.html.twig', array(
            'ofertas' => $ofertas
        ));
    }
}
```

```

        '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/Cupon/TiendaBundle/Entity/TiendaRepository.php
namespace Cupon\TiendaBundle\Entity;

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 OfertaBundle:Oferta o JOIN o.tienda t
             WHERE o.tienda = :id
             ORDER BY o.fecha_expiracion 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:

```

{# src/Cupon/TiendaBundle/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') }}</td>
            {% 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 %}
                
            
        {% endfor %}
    </tbody>

```

```

        </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 ? '✔' : '✘' }}</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/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExtranetController extends Controller
{
    // ...

    public function ofertaVentasAction($id)
    {
        $em = $this->getDoctrine()->getManager();

        $ventas = $em->getRepository('OfertaBundle:Oferta')
            ->findVentasByOferta($id);

        return $this->render('TiendaBundle: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/Cupon/OfertaBundle/Entity/OfertaRepository.php
namespace Cupon\OfertaBundle\Entity;
use Doctrine\ORM\EntityRepository;

class OfertaRepository extends EntityRepository

```

```
{
    // ...

    public function findVentasByOferta($oferta)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery(
            "SELECT v, o, u
             FROM OfertaBundle: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:

```
{# src/Cupon/TiendaBundle/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>



| DNI                     | Nombre y apellidos                                         | Fecha venta                                         |
|-------------------------|------------------------------------------------------------|-----------------------------------------------------|
| {{ venta.usuario.dni }} | {{ venta.usuario.nombre ~ ' ' ~ venta.usuario.apellidos }} | {{ venta.fecha   localizeddate('long', 'medium') }} |


```

```

<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, Symfony2 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 TiendaBundle:Tienda

The new TiendaType.php class file has been created under .../cupon.local/src/
Cupon/TiendaBundle/Form/TiendaType.php.
```

Si observas el formulario `TiendaType` generado automáticamente, verás que su código es muy sencillo:

```

// src/Cupon/TiendaBundle/Form/TiendaType.php
namespace Cupon\TiendaBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

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 setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Cupon\TiendaBundle\Entity\Tienda'
    ));
}

public function getName()
{
    return 'cupon_tiendabundle_tienda';
}
}

```

Como es posible que en la aplicación existan varios formularios diferentes para modificar los datos de las tiendas, es mejor mover este formulario a un directorio llamado [Extranet/](#) dentro del directorio [Form/](#) del bundle [TiendaBundle](#). Así será más fácil distinguir entre el formulario que modifica tiendas en la *extranet* y el que las modifica en el *backend*.

Crea el directorio [Extranet/](#), mueve el formulario [TiendaType.php](#) y actualiza su *namespace*:

```

// Antes
namespace Cupon\TiendaBundle\Form;

// Ahora
namespace Cupon\TiendaBundle\Form\Extranet;

```

A continuación, ajusta los tipos y opciones de los campos del formulario:

```

// src/Cupon/TiendaBundle/Form/Extranet/TiendaType.php
namespace Cupon\TiendaBundle\Form\Extranet;

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')
    ;
}

// ...
}

```

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.

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/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Cupon\TiendaBundle\Form\Extranet\TiendaType;

class ExtranetController extends Controller
{
    // ...

    public function perfilAction()

```

```

{
    $peticion = $this->getRequest();

    $tienda = $this->get('security.context')->getToken()->getUser();
    $formulario = $this->createForm(new TiendaType(), $tienda);

    $formulario->handleRequest($peticion);

    if ($formulario->isValid()) {
        // procesar formulario
    }

    return $this->render('TiendaBundle:Extranet:perfil.html.twig', array(
        'tienda'      => $tienda,
        'formulario'  => $formulario->createView()
    ));
}
}

```

La tienda actualmente *logueada* se obtiene a través del método `getUser()` del *token* creado por el componente de seguridad cuando la tienda accede a la *extranet*. Después, se crea un formulario de tipo `TiendaType` y se rellena con los datos de la tienda:

```

$tienda = $this->get('security.context')->getToken()->getUser();
$formulario = $this->createForm(new 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('TiendaBundle: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 Symfony2:

```

// src/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function perfilAction()
    {
        $peticion = $this->getRequest();

        $tienda = $this->get('security.context')->getToken()->getUser();
        $formulario = $this->createForm(new TiendaType(), $tienda);
    }
}

```

```

$formulario->handleRequest($peticion);

if ($formulario->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($tienda);
    $em->flush();

    $this->get('session')->setFlashBag()->add('info',
        'Los datos de tu perfil se han actualizado correctamente'
    );
}

return $this->redirect(
    $this->generateUrl('extranet_portada')
);
}

return $this->render('TiendaBundle:Extranet:perfil.html.twig', array(
    'tienda'      => $tienda,
    'formulario'  => $formulario->createView()
));
}
}

```

El único problema del código anterior es que cuando la tienda envía el formulario sin cambiar su contraseña, el valor de `password` es `null` y así se guardará en la base de datos. Para evitarlo, se guarda la contraseña original antes de llamar al método `handleRequest()`. Si la tienda no cambia la contraseña, se utiliza la original. Si la cambia, se codifica su valor antes de guardarla en la base de datos:

```

// src/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function perfilAction()
    {
        // ...

        $passwordOriginal = $formulario->getData()->getPassword();

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            if (null == $tienda->getPassword()) {
                // La tienda no cambia su contraseña, utilizar la original
                $tienda->setPassword($passwordOriginal);
            } else {
                // La tienda cambia su contraseña, codificar su valor
            }
        }
    }
}

```

```

        $encoder = $this->get('security.encoder_factory')
            ->getEncoder($tienda);
        $passwordCodificado = $encoder->encodePassword(
            $tienda->getPassword(),
            $tienda->getSalt()
        );
        $tienda->setPassword($passwordCodificado);
    }

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

    // ...
}

// ...
}
}

```

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 action="{{ path('extranet_perfil') }}" method="post">
    {{ form_widget(formulario) }}
    <input type="submit" value="Guardar cambios" />
</form>
{% 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 los métodos `form_row`, `form_widget` y `form_rest`:

```

{{-- src/Cupon/TiendaBundle/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 action="{{ path('extranet_perfil') }}" method="post">

```

```
<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>

    {{ form_rest(formulario) }}

    <input class="boton" type="submit" value="Guardar cambios" />
</div>
</form>
{% endblock %}
```

Esta página se ha dejado vacía a propósito

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

Siguiendo el mismo razonamiento explicado en los capítulos anteriores, el formulario con el que se modifican las ofertas en la *extranet* se crea en el directorio **Extranet/** del directorio **Form/** del *bundle* **OfertaBundle**. Crea ese directorio y añade dentro un archivo llamado **OfertaType.php** con el siguiente contenido:

```
// src/Cupon/OfertaBundle/Form/Extranet/OfertaType.php
namespace Cupon\OfertaBundle\Form\Extranet;

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('ruta_foto', 'file', array('required' => false))
            ->add('precio', 'money')
            ->add('descuento', 'money')
            ->add('umbral')
        ;
    }
}
```

```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Cupon\OfertaBundle\Entity\Oferta'
    ));
}

public function getName()
{
    return 'oferta_tienda';
}
```

Como Symfony2 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. Symfony2 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`:

```
// Cupon/OfertaBundle/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 $ruta_foto;  
  
/**  
 * @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 $fecha_publicacion;  
  
/**  
 * @ORM\Column(type="datetime", nullable=true)  
 *  
 * @Assert\DateTime  
 */  
protected $fecha_expiracion;  
  
/**  
 * @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, Symfony2 permite añadir validaciones dinámicas, tal como se explicó en la sección [Métodos de validación ad-hoc](#) (página 263) 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:

```
// Cupon/OfertaBundle/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->fecha_publicacion == null
            ||
            $this->fecha_expiracion == null) {
            return true;
        }

        return $this->fecha_expiracion > $this->fecha_publicacion;
    }
}
```

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/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Cupon\OfertaBundle\Entity\Oferta;
use Cupon\OfertaBundle\Form\Extranet\OfertaType;

class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction()
    {
        $peticion = $this->getRequest();

        $oferta = new Oferta();
        $formulario = $this->createForm(new OfertaType(), $oferta);

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($oferta);
            $em->flush();

            return $this->redirect($this->generateUrl('extranet_portada'));
        }

        return $this->render('TiendaBundle: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/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction()
    {
        $oferta = new Oferta();

        // ...
    }
}
```

```

$formulario->handleRequest($peticion);

if ($formulario->isValid()) {
    $tienda = $this->get('security.context')->getToken()->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 `ruta_foto` 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/Cupon/OfertaBundle/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="Cupon\OfertaBundle\Entity\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/Cupon/OfertaBundle/Form/Extranet/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 `ruta_foto`, 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/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    // ...

    public function ofertaNuevaAction()
    {
        $oferta = new Oferta();

        // ...

        $formulario->handleRequest($peticion);

        if ($formulario->isValid()) {
            $tienda = $this->get('security.context')->getToken()->getUser();

            $oferta->setCompras(0);
            $oferta->setTienda($tienda);
            $oferta->setCiudad($tienda->getCiudad());

            $oferta->subirFoto();

            // ...
        }
        // ...
    }
}
```

La gran ventaja de las entidades de Doctrine2 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/Cupon/OfertaBundle/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 `ruta_foto` 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 Symfony2 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 Doctrine2 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/Cupon/TiendaBundle/Controller/ExtranetController.php
public function ofertaNuevaAction()
{
    // ...

    $oferta->subirFoto(
        $this->container->getParameter('cupon.directorio.imagenes')
    );

    // ...
}

// Entidad: src/Cupon/OfertaBundle/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:

```
{# src/Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #}
{% extends '::extranet.html.twig' %}

{% block id 'oferta' %}
{% block title %}Añadir una nueva oferta{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<form action="{{ path('extranet_oferta_nueva') }}" method="post"
{{ form_enctype(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') }}
    </div>
```

```
    {{ form_errors(formulario.umbral) }}
    {{ form_widget(formulario.umbral) }}

```

```
</div>
```

```
    {{ form_rest(formulario) }}
```

```

<input class="boton" type="submit" value="Añadir oferta" />

```

```
</div>
```

```
</form>
```

```
{% endblock %}
```

```

{% block aside %}
    {# ... #}

```

```
{% endblock %}
```

No olvides añadir la instrucción `{{ form_enctype(formulario) }}` en la etiqueta `<form>`, ya que este formulario permite subir archivos.

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, también resulta común que el formulario deba incluir algún campo 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 que declaren que los datos de la oferta son correctos, que se comprometen a mantenerlos y que tienen capacidad para hacerlo. Estas condiciones legales se pueden mostrar en forma de *checkbox* que sea obligatorio activar al crear una oferta:

- Declaro que toda la información de esta oferta es correcta, que soy consciente de la obligación de cumplir las condiciones prometidas y que dispongo de los medios necesarios para hacerlo.

Añadir 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 más sencilla sería añadir una nueva propiedad en la entidad. El problema es que, además de *ensuciar* la base de datos con información irrelevante, habría que hacer lo mismo para cada campo adicional que se quiera añadir.

La mejor solución consiste en aprovechar la flexibilidad del *form builder* de Symfony2 para añadir campos de formulario que no se corresponden con ninguna propiedad de la entidad. El siguiente código muestra cómo añadir en el formulario `OfertaType` un *checkbox* adicional llamado `acepto` y que representa a las condiciones legales que deben aceptar las tiendas:

```
// src/Cupon/OfertaBundle/Form/Extranet/OfertaType.php
class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('descripcion')
            ->add('condiciones')
            ->add('foto', 'file', array('required' => false))
            ->add('precio', 'money')
            ->add('descuento', 'money')
            ->add('umbral')
        ;

        $builder->add('acepto', 'checkbox', array(
            'mapped' => false,
            'required' => false
        ));
    }

    // ...
}
```

La clave del código anterior es la opción `mapped` con un valor `false`, que indica a Symfony2 que este campo de formulario no corresponde a ninguna propiedad de la entidad. A continuación, añade el nuevo campo en la plantilla del formulario:

```
{# src/Cupon/TiendaBundle/Resources/views/Extranet/formulario.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>
```

Ahora el formulario ya muestra el `checkbox` con las condiciones legales que se deben aceptar. Como no se ha añadido la opción `'required' => false`, este campo es obligatorio, lo que en la práctica significa que el usuario debe activar el `checkbox` para poder enviar el formulario.

No obstante, es posible que el campo de formulario añadido no sea tan sencillo y su valor requiera de algún tipo de validación. En Symfony 2.0, este validador se definía con el método `addValidator()` dentro de la misma clase que construye el formulario.

En Symfony 2.3 este método se ha eliminado, por lo que la nueva forma de añadir validadores a los campos no asociados al modelo consiste en crear un `listener` que responda al evento `FormEvents::PRE_SUBMIT` del formulario.

En primer lugar, crea un nuevo *listener* de tipo `OfertaTypeListener` (cuyo código se definirá a continuación) y utiliza el método `addEventSubscriber()` del objeto `$builder` para asociarlo al evento `PRE_SUBMIT` del formulario:

```
// src/Cupon/OfertaBundle/Form/Extranet/OfertaType.php

// ...
use Symfony\Component\Form\FormEvents;
use Cupon\OfertaBundle\Listener\OfertaTypeListener;

class OfertaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('descripcion')
            ->add('condiciones')
            ->add('foto', 'file', array('required' => false))
            ->add('precio', 'money')
            ->add('descuento', 'money')
            ->add('umbral')

        ;

        $builder->add('acepto', 'checkbox', array(
            'mapped' => false,
            'required' => false
        ));

        $listener = new OfertaTypeListener();
        $builder->addEventSubscriber(FormEvents::PRE_SUBMIT, array($listener,
            'preSubmit'));
    }

    // ...
}
```

El primer argumento de `addEventSubscriber()` es el nombre del evento para el que se quiere ejecutar el *listener*. Los eventos se definen mediante constantes de la clase `FormEvents` y actualmente son los siguientes: `PRE_SUBMIT`, `SUBMIT`, `POST_SUBMIT`, `PRE_SET_DATA` y `POST_SET_DATA`. El segundo parámetro de `addEventSubscriber()` es el *listener* a ejecutar, que en este caso se indica como un método (`preSubmit`) de una clase (`OfertaTypeListener`). El tercer parámetro opcional de `addEventSubscriber()` es un número entero que indica la prioridad del *listener* (cuanto más alto, más prioridad).

El código del *listener* también es bastante conciso y sencillo:

```

namespace Cupon\OfertaBundle\Listener;

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormError;

class OfertaTypeListener
{
    public function preSubmit(FormEvent $event)
    {
        $formulario = $event->getForm();

        if (false == $formulario->get('acepto')->getData()) {
            $formulario->get('acepto')->addError(new FormError(
                'Debes aceptar las condiciones legales.'
            ));
        }
    }
}

```

El formulario que provocó que se ejecutara este *listener* se puede obtener a través del objeto `$event` que Symfony pasa automáticamente al método del *listener*. A partir del formulario, se puede obtener cualquier campo (`get('nombre_campo')`), sus valores (`getData()`) y también se pueden añadir mensajes de error (`addError()`) en caso de que los valores no sean los esperados. Los métodos que puedes utilizar sobre el formulario se definen en la interfaz [Symfony\Component\Form\FormInterface](#).

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/Cupon/TiendaBundle/Controller/ExtranetController.php
namespace Cupon\TiendaBundle\Controller;

// ...
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Cupon\OfertaBundle\Form\Extranet\OfertaType;

class ExtranetController extends Controller
{

```

```

public function ofertaEditarAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $oferta = $em->getRepository('OfertaBundle:Oferta')->find($id);

    if (!$oferta) {
        throw $this->createNotFoundException('La oferta no existe');
    }

    $contexto = $this->get('security.context');
    if (false === $contexto->isGranted('ROLE_EDITAR_OFERTA', $oferta)) {
        throw new AccessDeniedException();
    }

    if ($oferta->getRevisada()) {
        $this->get('session')->setFlashBag()->add('error',
            'La oferta no se puede modificar porque ya ha sido revisada'
        );
    }

    return $this->redirect($this->generateUrl('extranet_portada'));
}

$peticion = $this->getRequest();
$formulario = $this->createForm(new OfertaType(), $oferta);

$formulario->handleRequest($peticion);

if ($formulario->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($oferta);
    $em->flush();

    return $this->redirect(
        $this->generateUrl('extranet_portada')
    );
}

return $this->render('TiendaBundle: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 = $em->getRepository('OfertaBundle:Oferta')->find($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->get('security.context')->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->get('session')->getFlashBag()->add('error',
        'La oferta no se puede modificar porque ya ha sido revisada');

    return $this->redirect($this->generateUrl('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/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    public function ofertaEditarAction($id)
    {
        // ...

        $peticion = $this->getRequest();
```

```

$formulario = $this->createForm(new OfertaType(), $oferta);

$rutaFotoOriginal = $formulario->getData()->getRutaFoto();

$formulario->handleRequest($peticion);

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();
$formulario->handleRequest($peticion);
// ...

```

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 quieras encargarte de la gestión de las fotos subidas (copiarlas en el directorio destino, borrar las fotos que se han modificado, etc.) puedes hacer que Doctrine2 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) (http://symfony.com/doc/current/cookbook/doctrine/file_uploads.html#using-lifecycle-callbacks) de la documentación oficial de Symfony2 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/Cupon/TiendaBundle/Controller/ExtranetController.php
class ExtranetController extends Controller
{
    public function ofertaNuevaAction($id)
    {
        // ...

        return $this->render('TiendaBundle:Extranet:formulario.html.twig',
            array(
                'accion'      => 'crear',
                'formulario'  => $formulario->createView()
            )
        );
    }
}
```

```

public function ofertaEditarAction($id)
{
    // ...

    return $this->render('TiendaBundle: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:

```

{# Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #}
{# ... #-}

{% block title %}{{ accion == 'crear'
    ? 'Añadir una nueva oferta'
    : 'Modificar la oferta ' ~ oferta.nombre
}}{% endblock %}

{# ... #-}

```

Después, actualiza el atributo `action` de la etiqueta `<form>`, ya que cada acción corresponde a una ruta diferente:

```

{# Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #}
{# ... #-}

<form action="{{ accion == 'crear' ? path('extranet_oferta_nueva') :
path('extranet_oferta_editar', { 'id': oferta.id }) }}" method="post" {{ form_enctype(formulario) }}>

{# ... #-}

```

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:

```

{# Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #}
{# ... #-}

<div>
    {{ form_label(formulario.foto, 'Fotografía') }}
    {{ form_errors(formulario.foto) }}

    {% if accion == 'editar' %}

```

```



{{ 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:

```

{# Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #-}

{# ... #}

{% if accion == 'crear' %}
<div>
    {{ form_errors(formulario.acepto) }}
    {{ form_widget(formulario.acepto) }} <span>Declaro que toda la información de esta oferta es correcta, que soy consciente de la obligación de cumplir las condiciones prometidas y que dispongo de los medios necesarios para hacerlo.</span>
</div>
{% endif %}

{# ... #}

```

El último cambio consiste en ajustar el texto que muestra el botón del formulario:

```

{# Cupon/TiendaBundle/Resources/views/Extranet/formulario.html.twig #-}

{# ... #}

<input class="boton" type="submit" value="{{ accion == 'crear' ? 'Añadir oferta' : 'Guardar cambios' }}" />

{# ... #}

```

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/Cupon/OfertaBundle/Form/Extranet/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 Symfony2 es muy flexible, lo que significa que suele haber varias soluciones a una misma necesidad. En symfony 1 el *backend* siempre se desarrolla en una aplicación separada del *frontend*. Sin embargo, en Symfony es posible:

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 [OfertaBundle](#) se puede añadir el controlador [Controller/BackendController.php](#), el archivo de rutas [Resources/config/routing/backend.yml](#) y la carpeta de plantillas [Resources/views/Backend/](#).

La segunda opción se puede descartar en esta aplicación porque el tamaño esperado de los *bundles* de administración no justifica la necesidad de crear tantos *bundles*. Así que el próximo capítulo desarrolla la parte de administración utilizando la primera opción, un gran *bundle* llamado

`BackendBundle` con todos los elementos de administración de todas las entidades. Igualmente se muestra un ejemplo sencillo que cumple con la tercera alternativa.

Con todo lo anterior, ejecuta el siguiente comando para crear el *bundle* `BackendBundle` que se utiliza en los siguientes capítulos:

```
$ php app/console generate:bundle --namespace=Cupon/BackendBundle  
--bundle-name=BackendBundle --dir=src/ --format=yml --structure=no  
--no-interaction
```

16.2 Layout

El *backend* o parte de administración es una de las zonas más especiales del sitio web. Por eso se define un nuevo *layout* para todas sus páginas, aprovechando la herencia de plantillas a tres niveles (página 165) explicada en el capítulo 7.

Crea una nueva plantilla en `app/Resources/views/backend.html.twig` y copia en su interior el siguiente código Twig:

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

{% block stylesheets %}
<link href="{{ asset('bundles/oferta/css/normalizar.css') }}"
      rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/oferta/css/comun.css') }}"
      rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/backend/css/backend.css') }}"
      rel="stylesheet" type="text/css" />
{% endblock %}

{% block body %}
<header>
    <h1><a href="{{ path('backend_portada') }}">CUPON ADMIN</a></h1>
    <nav>
        <ul>
            <li><a href="{{ path('backend_oferta') }}">Ofertas</a></li>
            <li><a href="{{ path('backend_tienda') }}">Tiendas</a></li>
            <li><a href="{{ path('backend_usuario') }}">Usuarios</a></li>
            <li><a href="{{ path('backend_ciudad') }}">Ciudades</a></li>

            <li>{{ render(controller('CiudadBundle:Default:listaCiudades', {
                'ciudad': app.session.get('ciudad')
            })) }}</li>
        </ul>
    </nav>
</header>

<article>
    {% block article %}{% endblock %}
```

```
</article>
{%
    endblock %}
```

El bloque `stylesheets` enlaza las tres hojas de estilo CSS que definen el aspecto de las páginas del *backend*. Además de las habituales `normalizar.css` y `comun.css`, se crea una nueva CSS llamada `backend.css` que sólo incluye los estilos propios del *backend*:

```
{%
    block stylesheets %}
<link href="{{ asset('bundles/oferta/css/normalizar.css') }}"
      rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/oferta/css/comun.css') }}"
      rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/backend/css/backend.css') }}"
      rel="stylesheet" type="text/css" />
{%
    endblock %}
```

Después, en el bloque `body` se define la estructura de contenidos de todas las páginas, que está formada por una cabecera con el menú principal de navegación y un elemento `<article>` para mostrar los contenidos propios de la página. La cabecera incluye las rutas que se definirán en los próximos capítulos, así que si pruebas ahora el *layout* Twig mostrará un error indicando que las rutas no existen:

```
{%
    block body %}
<header>
    <h1><a href="{{ path('backend_portada') }}">CUPON ADMIN</a></h1>
    <nav>
        <ul>
            <li><a href="{{ path('backend_oferta') }}">Ofertas</a></li>
            <li><a href="{{ path('backend_tienda') }}">Tiendas</a></li>
            <li><a href="{{ path('backend_usuario') }}">Usuarios</a></li>
            <li><a href="{{ path('backend_ciudad') }}">Ciudades</a></li>

            {# ... #}
        {%
            endblock %}
```

NOTA Para ver las páginas del *backend* con un mejor aspecto, puedes copiar los contenidos de la hoja de estilos `backend.css` que se encuentra en el repositorio: <https://github.com/javierguiluz/Cupon/blob/2.3/src/Cupon/BackendBundle/Resources/public/css/backend.css>

Recuerda que para ver las nuevas hojas de estilo y el resto de archivos web (JavaScript e imágenes) antes debes *instalarlas* con el comando:

```
$ php app/console assets:install
```

16.2.1 Seleccionando la ciudad activa

El último elemento del menú principal de navegación es el más interesante, ya que es el que muestra la lista desplegable con las ciudades disponibles en la aplicación:

```
<li>{{ render(controller('CiudadBundle:Default:listaCiudades', {
    'ciudad': app.session.get('ciudad')
})) }}</li>
```

Para facilitar el trabajo de los administradores, los listados del *backend* sólo muestran la información relacionada con la ciudad actualmente seleccionada. Así por ejemplo, el listado de ofertas sólo muestra las ofertas publicadas en la ciudad que aparece seleccionada en esta lista desplegable.

Aunque en el *frontend* la ciudad activa forma parte de la URL de la página, en el *backend* la ciudad activa se almacena directamente en la sesión del usuario. Por tanto, para obtener su valor en la plantilla, se utiliza la función `app.session.get()` de Twig.

La lista de ciudades se muestra con la misma acción `listaCiudades()` del *bundle CiudadBundle* que se creó para el *frontend*:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/listaCiudades.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="selected"' : '' }}>
            {{ 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>
```

Debido a las diferencias entre el *backend* y el *frontend*, es necesario refactorizar ligeramente este código por lo siguiente:

```
{# src/Cupon/CiudadBundle/Resources/views/Default/listaCiudades.html.twig #-}
<select id="ciudadseleccionada">
    {% for ciudad in ciudades %}
        <option value="{{ ciudad.slug }}"
            data-url="{{ is_granted('ROLE_ADMIN')
                ? url('backend_ciudad_cambiar', { 'ciudad': ciudad.slug })
                : url('ciudad_cambiar', { 'ciudad': ciudad.slug }) }}
```

```

        }}"
        {{ ciudadActual == ciudad.slug ? 'selected="selected"' : '' }}>
        {{ 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>

```

Si el usuario que está viendo la página es de tipo administrador (`is_granted('ROLE_ADMIN')`), al seleccionar una ciudad se le redirige a la ruta `backend_ciudad_cambiar` (que todavía no está definida). Para el resto de usuarios el código sigue funcionando igual y son redirigidos a la ruta `ciudad_cambiar`.

Añade la nueva ruta `backend_ciudad_cambiar` en el archivo de rutas del *bundle BackendBundle*:

```
# src/Cupon/BackendBundle/Resources/config/routing.yml
backend_ciudad_cambiar:
    path:      /cambiar-a-{ciudad}
    defaults: { _controller: BackendBundle:Default:ciudadCambiar }
```

Y ahora define la acción `ciudadCambiar()` en el controlador `DefaultController`:

```
// src/Cupon/BackendBundle/Controller/DefaultController.php
namespace Cupon\BackendBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse;

class DefaultController extends Controller
{
    public function ciudadCambiarAction($ciudad)
    {
        $this->getRequest()->getSession()->set('ciudad', $ciudad);

        $dondeEstaba = $this->getRequest()->server->get('HTTP_REFERER');
        return new RedirectResponse($dondeEstaba, 302);
    }
}
```

Lo primero que hace la acción es guardar en la sesión del usuario la nueva ciudad seleccionada. A continuación, trata de determinar la página en la que se encontraba el usuario mediante el pa-

rámetro `HTTP_REFERER` del servidor. Por último, se redirige al usuario a la misma página en la que estaba (aunque como la ciudad activa ha cambiado, los contenidos de la página también habrán cambiado).

16.3 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 *role* llamado `ROLE_ADMIN` para distinguir a estos usuarios de los del *role* `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
            provider:    tiendas
            # ...

        frontend:
            pattern:      ^/*
            provider:    usuarios
            # ...

    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 Symfony2 para crear los administradores directamente en el archivo `security.yml`, sin crear ni modificar ninguna entidad de Doctrine2:

```
# 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*:

```
providers:
    # ...
    administradores:
        memory:
            users:
```

```

    admin: { password: 1234, roles: ROLE_ADMIN }
    jose: { password: secreto, roles: ['ROLE_ADMIN', 'ROLE_MANAGER'] }
}
editor: { password: s4jdi8Sp, roles: ['ROLE_ADMIN',
'ROLE_EDITOR'] }

```

La clave asociada a cada usuario (`admin`, `jose`, `editor`) es el nombre de usuario. Si alguno de estos nombres contiene guiones o empieza por un número, debes usar la notación alternativa:

```

providers:
# ...
administradores:
    memory:
        users:
            - { name: admin, password: 1234, roles: ROLE_ADMIN }
            - { name: jose, password: secreto,
                roles: ['ROLE_ADMIN', 'ROLE_MANAGER'] }
            - { name: editor, password: s4jdi8Sp,
                roles: ['ROLE_ADMIN', 'ROLE_EDITOR'] }

```

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 Symfony2 se guardan en claro (opción `plaintext`):

```

# app/config/security.yml

security:
# ...

encoders:
# ...
    Symfony\Component\Security\Core\User\User: plaintext

```

Guardar en claro las contraseñas de los administradores no es una buena práctica de seguridad. Por eso también puedes codificar estas contraseñas con el algoritmo SHA-512:

```

# app/config/security.yml

security:
# ...

providers:
# ...
administradores:
    memory:
        users:
            admin: { password: Eti36Ru/
pWG6WfoIPiDFUBxUuyvgMA4L8+LLuGbGyqV9ATuT9brCWPchBqX5vFTF+DgntacecW+sSGD+GZts2A==,
roles: ROLE_ADMIN }

```

```
encoders:  
    # ...  
    Symfony\Component\Security\Core\User\User: sha512
```

La contraseña del administrador sigue siendo `1234`, pero ahora es *imposible* de adivinar ni aún accediendo a los contenidos del archivo `security.yml`. Recuerda que la opción `sha512` codifica 5.000 veces seguidas la contraseña utilizando el algoritmo SHA de 512 bits y después codifica el resultado en base64. Si consideras suficiente con codificar la contraseña una sola vez, añade la siguiente configuración:

```
# app/config/security.yml  
  
security:  
    # ...  
  
    encoders:  
        # ...  
        Symfony\Component\Security\Core\User\User: { algorithm: sha512,  
                                                    iterations: 1 }
```

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. Symfony 1 cuenta con un completo *admin generator* que permite crear con un solo comando una parte de administración casi completa.

A pesar de que el *admin generator* de symfony 1 es su característica más destacada, Symfony2 no incluye todavía un *admin generator* completo. En su lugar dispone de un generador de código básico que permite crear un prototipo de la parte de administración, pero que es muy inferior al de symfony 1.

Para disponer de una mejor perspectiva de las opciones disponibles, en este capítulo se crea la parte de administración del sitio web de tres formas diferentes:

- **Admin generator manual**, haciendo uso de todo lo aprendido en los capítulos anteriores.
- **Generador de código** de Symfony2.
- **Admin generator desarrollado por terceros**, como por ejemplo el *admin bundle* del proyecto Sonata.

17.1 Admin generator manual

Por razones de espacio, el *admin generator* manual se desarrolla para la entidad más sencilla: **Ciudad**. Cuando un administrador accede a `/backend/ciudad` se le muestra la lista de ciudades de la aplicación (acción `index`) y varios enlaces con las cuatro operaciones *CRUD* habituales: crear nuevas ciudades (acción `crear`) y ver, modificar o borrar las ciudades existentes (acciones `ver`, `actualizar` y `borrar` respectivamente).

17.1.1 Enrutamiento

Las cinco acciones anteriores (`index` + las cuatro operaciones *CRUD*) requieren la definición de las siguientes cinco rutas:

```
# src/cupon/BackendBundle/Resources/config/routing/ciudad.yml
backend_ciudad:
    path:    /
    defaults: { _controller: "BackendBundle:Ciudad:index" }

backend_ciudad_crear:
    path:    /crear
    defaults: { _controller: "BackendBundle:Ciudad:crear" }

backend_ciudad_ver:
```

```

path:      /ver/{id}
defaults: { _controller: "BackendBundle:Ciudad:ver" }

backend_ciudad_actualizar:
path:      /actualizar/{id}
defaults: { _controller: "BackendBundle:Ciudad:actualizar" }

backend_ciudad_borrar:
path:      /borrar/{id}
defaults: { _controller: "BackendBundle:Ciudad:borrar" }

```

No es necesario que añadas `/ciudad` en el patrón de todas las rutas anteriores, ya que es más fácil añadir este prefijo con la opción `prefix` al importar todas las rutas:

```

# src/cupon/BackendBundle/Resources/config/routing.yml

# ...

BackendCiudad:
resource: "@BackendBundle/Resources/config/routing/ciudad.yml"
prefix:  /ciudad

```

17.1.2 CRUD básico de una entidad

La manipulación de los datos de las entidades se realiza mediante formularios, por lo que en primer lugar se crea el formulario `CiudadType` en el directorio `Form/` del *bundle* `BackendBundle`:

```

// src/Cupon/BackendBundle/Form/CiudadType.php
namespace Cupon\BackendBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CiudadType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nombre')
            ->add('slug')
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Cupon\CiudadBundle\Entity\Ciudad',
        ));
    }
}

```

```

public function getName()
{
    return 'cupon_backend_ciudad';
}
}

```

A continuación, crea un nuevo controlador llamado `CiudadController` para definir todas las acciones de la parte de administración de las ciudades:

```

// src/Cupon/BackendBundle/Controller/CiudadController.php
namespace Cupon\BackendBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Cupon\CiudadBundle\Entity\Ciudad;
use Cupon\BackendBundle\Form\CiudadType;

class CiudadController extends Controller
{
    public function indexAction()
    {
        $em = $this->getDoctrine()->getManager();
        $ciudades = $em->getRepository('CiudadBundle:Ciudad')->findAll();

        return $this->render('BackendBundle:Ciudad:index.html.twig', array(
            'ciudades' => $ciudades
        ));
    }

    public function verAction($id)
    {
        $em = $this->getDoctrine()->getManager();
        $ciudad = $em->getRepository('CiudadBundle:Ciudad')->find($id);

        if (!$ciudad) {
            throw $this->createNotFoundException('No existe esa ciudad');
        }

        return $this->render('BackendBundle:Ciudad:ver.html.twig', array(
            'ciudad' => $ciudad,
        ));
    }

    public function crearAction()
    {
        $peticion = $this->getRequest();

        $ciudad = new Ciudad();
        $formulario = $this->createForm(new CiudadType(), $ciudad);
    }
}

```

```
$form->handleRequest($request);

if ($formulario->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($ciudad);
    $em->flush();

    return $this->redirect($this->generateUrl('backend_ciudad'));
}

return $this->render('BackendBundle:Ciudad:crear.html.twig', array(
    'formulario' => $formulario->createView()
));
}

public function actualizarAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $ciudad = $em->getRepository('CiudadBundle:Ciudad')->find($id);

    if (!$ciudad) {
        throw $this->createNotFoundException('No existe esa ciudad');
    }

    $formulario = $this->createForm(new CiudadType(), $ciudad);

    $peticion = $this->getRequest();
    $form->handleRequest($request);

    if ($formulario->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($ciudad);
        $em->flush();

        return $this->redirect($this->generateUrl('backend_ciudad'));
    }

    return $this->render('BackendBundle:Ciudad:actualizar.html.twig', array(
        'formulario' => $formulario->createView(),
        'ciudad'      => $ciudad
    ));
}

public function borrarAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $ciudad = $em->getRepository('CiudadBundle:Ciudad')->find($id);
```

```

    if (!$ciudad) {
        throw $this->createNotFoundException('No existe esa ciudad');
    }

    $em->remove($ciudad);
    $em->flush();

    return $this->redirect($this->generateUrl('backend_ciudad'));
}
}

```

En la mayoría de sitios web, desarrollar su parte de administración resulta sencillo porque se basan en mostrar listados de datos o formularios sin apenas necesidad de ser modificados. El código del controlador anterior es un buen ejemplo de ello.

Las plantillas asociadas a las acciones anteriores también son muy sencillas:

```

{# src/Cupon/BackendBundle/Resources/views/Ciudad/index.html.twig #-}
{% extends '::backend.html.twig' %}

{% block id 'ciudad' %}
{% block title %}Listado de ciudades{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>



| NOMBRE              | SLUG              | ACCIONES                                                                             |
|---------------------|-------------------|--------------------------------------------------------------------------------------|
| {{ ciudad.nombre }} | {{ ciudad.slug }} | <a href="{{ path('backend_ciudad_actualizar', { 'id': ciudad.id }) }}">MODIFICAR</a> |


```

```
<a href="{{ path('backend_ciudad_nueva') }}">Crear una nueva ciudad</a>
{% endblock %}

{# src/Cupon/BackendBundle/Resources/views/Ciudad/ver.html.twig #-}
{% extends '::backend.html.twig' %}

{% block ciudad %}
{% block title %}Datos de la ciudad {{ ciudad.nombre }}{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>



| ID     | {{ ciudad.id }}     |
|--------|---------------------|
| Nombre | {{ ciudad.nombre }} |
| Slug   | {{ ciudad.slug }}   |



- Volver al listado
- Modificar
- Borrar


{% endblock %}

{# src/Cupon/BackendBundle/Resources/views/Ciudad/crear.html.twig #-}
{% extends '::backend.html.twig' %}
```

```

{% block id 'ciudad' %}
{% block title %}Añadir una nueva ciudad{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<form action="{{ path('backend_ciudad_crear') }}" method="post">
    {{ form_widget(formulario) }}

    <input type="submit" value="Añadir ciudad" />
</form>

<a href="{{ path('backend_ciudad') }}>Volver al listado</a>
{% endblock %}

{# src/Cupon/BackendBundle/Resources/views/Ciudad/actualizar.html.twig #-}
{% extends '::backend.html.twig' %}

{% block id 'ciudad' %}
{% block title %}Modificar la ciudad {{ ciudad.nombre }}{% endblock %}

{% block article %}
<h1>{{ block('title') }}</h1>

<form action="{{ path('backend_ciudad_actualizar', { 'id': ciudad.id }) }}" method="post">
    {{ form_widget(formulario) }}

    <input type="submit" value="Guardar cambios" />
</form>

<ul>
    <li>
        <a href="{{ path('backend_ciudad') }}>Volver al listado</a>
    </li>
    <li>
        <a href="{{ path('backend_ciudad_borrar', { 'id': ciudad.id }) }}>Borrar</a>
    </li>
</ul>
{% endblock %}

```

17.1.3 Ventajas y desventajas

La gran ventaja de crear la parte de administración manualmente es que dispones de la máxima flexibilidad, ya que el único límite son tus conocimientos sobre Symfony2 o el tiempo disponible para su desarrollo.

Por contra, la gran desventaja es que supone un gran esfuerzo de desarrollo y que puedes acabar con mucho código repetido si la parte de administración es grande.

Por todo ello, el único escenario en el que puede ser útil crear la parte de administración a mano son los micro-proyectos o aquellos proyectos con unos requerimientos tan especiales que requieren disponer de toda la flexibilidad posible.

17.2 Generador de código de Symfony2

Symfony2 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 las siguientes tareas para generar código:

- `generate:bundle`, genera toda la estructura inicial de archivos y directorios de un *bundle*.
- `generate:doctrine:form` (o `doctrine:generate:form`), genera un formulario con todas las propiedades de la entidad de Doctrine2 indicada.
- `generate:doctrine:entity` (o `doctrine:generate:entity`), genera todo el código de una entidad de Doctrine2 interactivamente, en base a preguntas y respuestas sobre sus propiedades y características.
- `generate:doctrine:entities` (o `doctrine:generate:entities`), genera el código de una o más entidades de Doctrine2 en función de la configuración XML o YAML disponible en el *bundle*.

Además de estas tareas, Symfony2 incluye otra que genera código y que está relacionada con los *admin generators*:

- `generate:doctrine:crud` (o `doctrine:generate:crud`), genera todo el código necesario para administrar los datos de una entidad de Doctrine2 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=OfertaBundle: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* `OfertaBundle`.
- `--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, Symfony2 genera los siguientes archivos y realiza los siguientes cambios:

- Crea el controlador `OfertaBundle\Controller\OfertaController.php` con todas las acciones del *CRUD* de las ofertas.
- Crea el formulario `OfertaBundle\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 `OfertaBundle/Resources/views/Oferta/`.
- Crea el archivo de enrutamiento `oferta.yml` en el directorio `OfertaBundle/Resources/config/routing/`.
- Importa el anterior archivo `oferta.yml` desde el archivo `OfertaBundle/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=OfertaBundle: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/Cupon/OfertaBundle/Resources/config/routing.yml).
```

```
OfertaBundle_oferta:  
    resource: "@OfertaBundle/Resources/config/routing/oferta.yml"  
    prefix:  /oferta
```

17.2.1 Reordenando los archivos generados

Symfony2 genera todos los archivos en el mismo *bundle* en el que se encuentra la entidad. Esta es la opción que mezcla el *frontend* y el *backend* de la entidad en un mismo *bundle*. Si optas por centralizar la parte de administración de todas las entidades en un único *bundle*, debes modificar todos los archivos anteriores. Suponiendo que el nuevo *bundle* sea `BackendBundle`:

1. Mueve el controlador `OfertaBundle/Controller/OfertaController.php` al directorio `BackendBundle/Controller/` y actualiza su *namespace* y las rutas de las instrucciones `use`.
2. Mueve el formulario `OfertaBundle/Form/OfertaType.php` al directorio `BackendBundle/Form/`. Actualiza su *namespace* y el valor que devuelve el método `getName()`.
3. Mueve el directorio de plantillas `OfertaBundle/Resources/views/Oferta/` al directorio `BackendBundle/Resources/views/`. Actualiza en el controlador `OfertaController` las rutas de las plantillas en las instrucciones `$this->render('...')`.
4. Mueve el archivo de rutas `OfertaBundle/Resources/config/routing/oferta.yml` al directorio `BakcendBundle/Resources/config/routing/`. Elimina en el archivo `OfertaBundle/Resources/config/routing.yml` la importación de este archivo e impórtalo en el archivo `BackendBundle/Resources/config/routing.yml`:

```
# src/BackendBundle/Resources/config/routing.yml

# ...
BackendOferta:
    resource: "@BackendBundle/Resources/config/routing/oferta.yml"
    prefix: /oferta
```

Para hacer que el listado de ofertas sea también la portada del *backend*, añade la siguiente ruta en el mismo archivo:

```
# src/BackendBundle/Resources/config/routing.yml

# ...
backend_portada:
    path: /
    defaults: { _controller: BackendBundle:Oferta:index }
```

17.2.2 Refactorizando los archivos generados

Después de los cambios de las secciones anteriores, ya puedes probar la parte de administración generada automáticamente por Symfony2 para las ofertas. Accede a la URL `http://cupon.local/app_dev.php/backend` o `http://cupon.local/app_dev.php/backend/oferta` para ver el listado de todas las ofertas:

Oferta list

ID	Nombre	Slug	Descripción	Condiciones	Foto	Precio	Descuento	Fecha_publicacion	Fecha_expiracion	Compras	Umbrales	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 imperdier ut imperdier aut	oferta-imperdier-ut-imperdier-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 imperdier lacinia sitamet enimsit facilisis	oferta-imperdier-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 Symfony2

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 la plantilla `backend.html.twig`:

```
{# src/Cupon/BackendBundle/Resources/views/Oferta/index.html.twig #}
{% extends '::backend.html.twig' %}

{% block article %}
<h1>Oferta list</h1>

{# ... #}
{% endblock %}
```

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/Cupon/BackendBundle/Resources/views/Oferta/index.html.twig #}
{% extends '::backend.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.

17.2.3 Añadiendo un paginador

Las plantillas generadas por Symfony2 no incluyen un paginador para mostrar de forma cómoda los listados largos. Doctrine2 tampoco dispone de un paginador completamente funcional, ni siquiera a través de sus extensiones oficiales. Afortunadamente, varias empresas externas han desarrollado y publicado sus propios *bundles* de paginación para Symfony2.

Uno de los más sencillos es IdeupSimplePaginatorBundle (<https://github.com/javiacei/Ideup-SimplePaginatorBundle>) desarrollado por la empresa ideup! (<http://www.ideup.com/>). Internamente este paginador utiliza la extensión *Paginate* de las extensiones de Doctrine2 desarrolladas por Benjamin Eberlei (<https://github.com/beberlei/DoctrineExtensions>).

17.2.3.1 Instalación y configuración

Para instalar el paginador, añade en primer lugar la siguiente dependencia en el archivo `composer.json` del proyecto:

```
{
  "require": {
    "ideup/simple-paginator-bundle": "dev-master"
  }
}
```

Guarda los cambios y descarga e instala las nuevas dependencias ejecutando el siguiente comando:

```
$ composer update
```

NOTA Para que te funcione este comando, debes instalar Composer globalmente (página 46) en tu ordenador, tal y como se explicó en el capítulo 3.

Por último, activa en el archivo `AppKernel.php` el nuevo *bundle* del paginador para que esté disponible en la aplicación:

```
// app/AppKernel.php
use Symfony\Component\HttpKernel\Kernel;
```

```

use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),

            // ...

            new Ideup\SimplePaginatorBundle\IdeupSimplePaginatorBundle(),
        );

        // ...
    }
}

```

17.2.3.2 Creando el paginador

En las secciones anteriores se ha desarrollado la parte de administración de las ofertas. La portada del *backend* muestra el listado de todas las ofertas de la ciudad activa. Para facilitar el trabajo de los administradores, a continuación se añade un paginador para mostrar las ofertas de diez en diez.

Al igual que sucede con la mayoría de paginadores, el paginador *IdeupSimplePaginatorBundle* no trabaja ni con entidades, ni con objetos ni con colecciones de objetos. El paginador requiere como parámetro la consulta DQL con la que se buscan las entidades.

El listado de ofertas se obtiene a través de la consulta `findTodasLasOfertas()` del repositorio `CiudadRepository.php`, que devuelve una colección con todas las ofertas de la ciudad indicada:

```

// src/Cupon/CiudadBundle/Entity/CiudadRepository.php
class CiudadRepository extends EntityRepository
{
    // ...

    public function findTodasLasOfertas($ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t FROM OfertaBundle:Oferta o
                JOIN o.tienda t JOIN o.ciudad c
                WHERE c.slug = :ciudad
            ORDER BY o.fecha_publicacion DESC');
        $consulta->setParameter('ciudad', $ciudad);
    }
}

```

```

        return $consulta->getResult();
    }
}

```

Para poder paginar esta consulta, su código se divide en dos métodos: el primero prepara la consulta y el segundo la ejecuta y devuelve el resultado. Resulta muy recomendable utilizar una nomenclatura estandarizada para distinguir fácilmente qué hace cada método. Así por ejemplo, los métodos cuyo nombre sea `queryXXX()` sólo devuelven la consulta y los métodos `findXXX()` ejecutan las consultas:

```

class CiudadRepository extends EntityRepository
{
    // ...

    public function queryTodasLasOfertas($ciudad)
    {
        // devuelve la consulta para obtener todas las ofertas de la ciudad
    }

    public function findTodasLasOfertas($ciudad)
    {
        // devuelve todas las ofertas de la ciudad ejecutando la consulta que
        // obtiene con el método anterior
    }
}

```

Siguiendo las indicaciones anteriores, el método `findTodasLasOfertas()` se divide en los dos siguientes métodos:

```

// src/Cupon/CiudadBundle/Entity/CiudadRepository.php
class CiudadRepository extends EntityRepository
{
    // ...

    public function queryTodasLasOfertas($ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery('
            SELECT o, t FROM OfertaBundle:Oferta o
                JOIN o.tienda t JOIN o.ciudad c
                WHERE c.slug = :ciudad
                ORDER BY o.fecha_publicacion DESC');
        $consulta->setParameter('ciudad', $ciudad);

        return $consulta;
    }

    public function findTodasLasOfertas($ciudad)

```

```
{  
    return $this->queryTodasLasOfertas($ciudad)->getResult();  
}  
}
```

Dividir las consultas en dos métodos tiene la ventaja de que el código de las consultas no se duplica y que el resto de la aplicación no se ve afectado por los cambios requeridos por el paginador.

Después de estos cambios, ya puedes añadir el paginador en la acción que muestra el listado de ofertas. El código original de la acción era el siguiente:

```
// src/Cupon/BackendBundle/Controller/OfertaController.php  
class OfertaController extends Controller  
{  
    public function indexAction()  
    {  
        // ...  
  
        $entities = $em->getRepository('CiudadBundle:Ciudad')  
            ->findTodasLasOfertas($slug);  
  
        return $this->render('BackendBundle:Oferta:index.html.twig', array(  
            'entities' => $entities  
        ));  
    }  
  
    // ...  
}
```

Para añadir el paginador, refactoriza el código anterior por lo siguiente:

```
// src/Cupon/BackendBundle/Controller/OfertaController.php  
class OfertaController extends Controller  
{  
    public function indexAction()  
    {  
        // ...  
  
        $paginador = $this->get('ideup.simple_paginator');  
  
        $entities = $paginador->paginate(  
            $em->getRepository('CiudadBundle:Ciudad')->queryTodasLasOfertas($slug)  
        )->getResult();  
  
        return $this->render('BackendBundle:Oferta:index.html.twig', array(  
            'entities' => $entities  
        ));  
    }  
}
```

```
// ...  
}
```

En primer lugar, el controlador obtiene el objeto del paginador a través del servicio `ideup.simplePaginator`. A continuación se invoca el método `paginate()` al que se le pasa la consulta con la que se obtienen todas las entidades. Por último, se invoca el método `getResult()` para obtener la colección de entidades correspondientes a la página en la que actualmente se encuentra el paginador.

Para mostrar el paginador en la plantilla, puedes hacer uso de una función de Twig definida por el propio paginador:

```
{# ... #}  
  
{{ simplePaginator_render('backend_oferta') }}
```

La función `simplePaginator_render()` requiere un único argumento: el nombre de la ruta asociada a la página en la que se incluye el paginador. No es necesario pasarle también el número de página, ya que el propio paginador se encarga de gestionarlo de forma transparente.

Para modificar el aspecto y las características del paginador, puedes pasar varias opciones como tercer argumento de la función:

- `routeParams`, parámetros que se pasan a la ruta indicada como primer argumento de la función.
- `container_class`, clase CSS que se aplica al elemento `` que encierra al paginador (por defecto, `simplePaginator`).
- `currentClass`, clase CSS que se aplica al elemento `` que encierra el enlace de la página actual (por defecto, `current`).
- `previousPageText` y `nextPageText`, texto que se muestra para hacer referencia a la página anterior/siguiente (por defecto, `Previous` y `Next` respectivamente).
- `previousEnabledClass` y `nextEnabledClass`, clase CSS que se aplica al texto de la página anterior/siguiente cuando existe la página anterior/siguiente (por defecto `left` y `right` respectivamente).
- `previousDisabledClass` y `nextDisabledClass`, clase CSS que se aplica al texto de la página anterior/siguiente cuando no existe la página anterior/siguiente (por defecto `left_disabled` y `right_disabled` respectivamente).
- `firstPageText` y `lastPageText`, texto que se muestra para hacer referencia a la primera/última página (por defecto, `First` y `Last` respectivamente),
- `firstEnabledClass` y `lastEnabledClass`, clase CSS que se aplica al texto de la primera/última página cuando existe la primera/última página (por defecto `first` y `last` respectivamente).

- `firstDisabledClass` y `lastDisabledClass`, clase CSS que se aplica al texto de la primera/última página cuando no existe la primera/última página (por defecto `first_disabled` y `last_disabled` respectivamente).

```
{# ... #}

{{ simplePaginator_render('backend_oferta', null, {
    'container_class': 'paginador',
    'previousPageText': 'Anterior',
    'nextPageText': 'Siguiente',
    'currentClass': 'actual',
    'firstPageText': 'Primera',
    'lastPageText': 'Última'
}) }}
```

Si necesitas un control todavía más preciso de la forma en la que se muestran los diferentes elementos del paginador, puedes pasar el objeto `$paginador` del controlador a la plantilla:

```
// src/Cupon/BackendBundle/Controller/OfertaController.php
class OfertaController extends Controller
{
    public function indexAction()
    {
        // ...

        $paginador = $this->get('ideup.simple_paginator');

        $entities = $paginador->paginate(
            $em->getRepository('CiudadBundle:Ciudad')->queryTodasLasOfertas($slug)
        )->getResult();

        return $this->render('BackendBundle:Oferta:index.html.twig', array(
            'entities' => $entities,
            'paginador' => $paginador
        ));
    }

    // ...
}
```

Ahora ya puedes adaptar el aspecto del paginador a tus necesidades accediendo a las propiedades del objeto paginador:

```
<ul class="paginador">
    {% if paginador.currentPage > 1 %}
        <li><a href="{{ path('backend_oferta', { 'page': paginador.previousPage
    }) }}>Anterior</a></li>
    {% else %}
        <li>Anterior</li>
```

```

        {% endif %}

        {% for page in paginador.minPageInRange..paginador.maxPageInRange %}
            {% if page == paginador.currentPage %}
                <li class="actual">{{ page }}</li>
            {% else %}
                <li><a href="{{ path('backend_oferta', { 'page': page }) }}">{{ page
}}</a></li>
            {% endif %}
        {% endfor %}

        {% if paginador.currentPage < paginador.lastPage %}
            <li><a href="{{ path('backend_oferta', { 'page': paginador.nextPage }) }}"
}>Siguiente</a></li>
        {% else %}
            <li>Siguiente</li>
        {% endif %}
    </ul>

```

Por defecto el paginador muestra 10 entidades en cada página y permite seleccionar un máximo de 10 páginas. De esta forma, si la consulta produce 100 páginas de resultados, sólo se ven cada vez 10 números de página diferentes, que se van desplazando a medida que se selecciona una u otra página. Puedes modificar estos valores con los métodos que incluye el objeto paginador:

```

$paginator = $this->get('ideup.simple_paginator');

// Ahora cada página muestra 20 entidades
$paginator->setItemsPerPage(20);

// Ahora sólo se muestran 5 números de página en el paginador
$paginator->setMaxPageItems(5);

```

17.2.4 Completando el resto del *backend*

El *admin generator* de la entidad `Oferta` que se acaba de desarrollar es sólo una de las secciones que forman el *backend* de la aplicación. Para completarla, es necesario crear el *admin generator* de las entidades `Ciudad`, `Tienda`, `Usuario` y `Venta`:

```

$ php app/console doctrine:generate:crud --entity=CiudadBundle:Ciudad
--route-prefix=backend/ciudad --with-write --format=yml --no-interaction

$ php app/console doctrine:generate:crud --entity=TiendaBundle:Tienda
--route-prefix=backend/tienda --with-write --format=yml --no-interaction

$ php app/console doctrine:generate:crud --entity=UsuarioBundle:Usuario
--route-prefix=backend/usuario --with-write --format=yml --no-interaction

$ php app/console doctrine:generate:crud --entity=OfertaBundle:Venta
--route-prefix=backend/venta --with-write --format=yml --no-interaction

```

Al ejecutar este último comando se produce el siguiente error:

```
[RuntimeException]
The CRUD generator does not support entity classes with multiple primary keys.
```

El generador de código de Symfony2 es tan incompleto que no soporta por ejemplo entidades cuya clave primaria esté compuesta por dos o más propiedades, como sucede en la entidad `Venta`. La única solución por el momento consiste en crearlo manualmente adaptando el código de cualquier otro *admin generator*.

Una vez generada la estructura básica de la administración de cada entidad, debes seguir los mismos pasos de la sección anterior para mover los archivos generados al *bundle BackendBundle*:

1. Mover el controlador generado al directorio `Controller/` del *bundle BackendBundle*, actualizando las rutas de su `namespace` y de las instrucciones `use` necesarias.
2. Mover el formulario generado al directorio `Form/` del *bundle BackendBundle*, actualizando su `namespace` y el valor devuelto por el método `getName()`.
3. Mover el directorio con las plantillas generadas `index`, `new`, `edit` y `show` al directorio `Resources/views/` del *bundle BackendBundle*, actualizando en el controlador las rutas de las instrucciones `$this->render('...')`.
4. Mover el archivo de rutas generado al directorio `Resources/config/routing/` del *bundle BackendBundle* e importarlas desde el archivo `Resources/config/routing.yml`.
5. Refactorizar el código del controlador, formulario y plantillas para adaptarse a las necesidades específicas del *admin generator* deseado.

17.2.5 Ventajas y desventajas

La ventaja de utilizar el generador de código de Symfony2 es que dispones de un *CRUD* completo casi sin esfuerzo y sin renunciar a la flexibilidad de poder añadir o modificar el código generado.

Su punto débil es que todavía no es un *admin generator* completo, ya que le faltan características tan esenciales como la paginación, los filtros de búsqueda y el reordenamiento de los listados por columnas.

Por tanto, el generador de código es una buena herramienta para prototipar rápidamente la parte de administración del sitio web. De hecho, el código generado es un excelente punto de partida para desarrollar tu propia parte de administración.

17.3 SonataAdminBundle

El proyecto Sonata (<http://sonata-project.org>) , fundado por Thomas Rabaix y esponsorizado por la empresa Ekino (<http://ekino.fr>) , tiene como objetivo crear una solución de comercio electrónico para Symfony2. El desarrollo se basa en componentes independientes, siendo los principales:

- *Admin bundle*, que es un *admin generator* mucho más completo que el que incluye Symfony2.
- *Media bundle*, que permite generar todo tipo de medios como archivos, imágenes y vídeos.

- *Page bundle*, que permite convertir cualquier acción de Symfony2 en un CMS o gestor de contenidos.

Como cada componente es independiente de los demás, puedes utilizar en tus proyectos solamente aquellos que necesites. En este ejemplo sólo se utiliza el *admin bundle* para crear la parte de administración del sitio web.

A pesar de sus buenas intenciones, el principal problema de Sonata es su mala documentación. El proyecto evoluciona constantemente, pero sus cambios no se documentan. Por eso la experiencia de usar Sonata para crear sitios medianamente complejos es desesperante. Así que este libro ya no explica cómo usar Sonata con Symfony 2.3. Consulta las ediciones anteriores para saber cómo utilizarlo con Symfony 2.0 y Symfony 2.1.

Esta página se ha dejado vacía a propósito

CAPÍTULO 18

Newsletters y comandos de consola

Los usuarios registrados en la aplicación *Cupon* reciben a diario un email con la información sobre la *oferta del día* de la ciudad a la que están asociados. Si no desean recibir el email, pueden indicarlo en su perfil gracias a la propiedad `permite_email` 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 Symfony2 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, Symfony2 incluye decenas de comandos que se pueden listar ejecutando `php app/console` desde el directorio raíz del proyecto. Los comandos propios de tu aplicación se crean con las mismas herramientas que los comandos estándar y Symfony2 los trata exactamente igual que a sus comandos.

Por convención, los comandos se guardan en el directorio `Command/` del *bundle*. Además, cada comando se define en una clase cuyo nombre siempre acaba en `Command`. Una buena práctica consiste en agrupar todos los comandos de administración en un mismo *bundle* (idealmente un *bundle* de tipo *backend* o administración).

Así que para desarrollar el comando que envía el email diario, crea el directorio `src/Cupon/BackendBundle/Command/` y añade en su interior un archivo llamado `EmailOfertaDelDiaCommand.php` con el siguiente contenido:

```
// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
namespace Cupon\BackendBundle\Command;

use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Output\Output;
use Symfony\Component\Console\Command\Command;
```

```

class EmailOfertaDelDiaCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('email:oferta-del-dia')
            ->setDefinition(array())
            ->setDescription('Genera y envía a cada usuario el email con la
                oferta diaria')
            ->setHelp(<<<EOT
El comando <info>email:oferta-del-dia</info> genera y envía un email con la
oferta del día de la ciudad en la que se ha apuntado el usuario.
EOT
        );
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $output->writeln('Generando emails...');

        // ...
    }
}

```

Los comandos de consola de Symfony2 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 email:oferta-del-dia`. No es obligatorio hacerlo, pero resulta habitual dividir el nombre en dos partes separadas por el carácter `:`. La primera parte actúa de *namespace* y sirve para agrupar todos los comandos relacionados (`doctrine:`, `generate:`, etc.) La segunda parte se considera el nombre específico del comando dentro de ese grupo. En este ejemplo, `email:` sería la parte común de todos los comandos que envían o generan emails).
- `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`

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 `email:oferta-del-dia` junto al resto de comandos internos de Symfony2:

```
$ php app/console

Available commands:
  help                               Displays help for a command
  list                               Lists commands
  assetic
    assetic:dump                    Dumps all assets to the filesystem

// ...

doctrine
  doctrine:cache:clear-metadata   Clear all metadata cache for a entity manager
  doctrine:cache:clear-query      Clear all query cache for a entity manager
  doctrine:cache:clear-result     Clear result cache for a entity manager
  doctrine:database:create        Create the configured databases
  doctrine:database:drop          Drop the configured databases

// ...

email
  email:oferta-del-dia           Genera y envía a cada usuario el email con la
                                 oferta diaria

generate
  generate:bundle                Generates a bundle
  generate:doctrine:crud         Generates a CRUD based on a Doctrine entity

// ...
```

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 email:oferta-del-dia sevilla
```

Este tipo de parámetros se llaman **argumentos** y cada comando puede definir tantos como desee. Los argumentos, que pueden ser obligatorios u opcionales, se definen mediante la clase `InputArgument` dentro del método `setDefinition()` del comando:

```
// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php

// ...
```

```
protected function configure()
{
    $this
        ->setName('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. No obstante, si el comando tiene dos o más argumentosopcionales, 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. Indica que el valor es de tipo array de PHP, lo que permite pasar información muy compleja al comando sin tener que definir multitud de argumentos. Se puede combinar con los anteriores: `InputArgument::OPTIONAL | InputArgument::IS_ARRAY`.
- `$descripcion`, es la descripción del argumento que se muestra cuando se consulta la ayuda del comando mediante `php app/console help nombre-del-comando`.
- `$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 admite los valores `generar` y `enviar`:

```
$ php app/console 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/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php

// ...

protected function configure()
{
    $this
        ->setName('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 Symfony2 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/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
// ...
```

```

protected function configure()
{
    $this
        ->setName('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',
                'enviar'),
        ))
        ->...
}

```

Como tanto el argumento como la opción son opcionales, puedes combinarlos como quieras al ejecutar el comando:

```

$ php app/console email:oferta-del-dia sevilla --accion=generar
$ php app/console email:oferta-del-dia --accion=enviar sevilla
$ php app/console email:oferta-del-dia --accion=generar
$ php app/console 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/Cupon/BackendBundle/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.

18.1.2.1 Mostrando mensajes informativos

Los mensajes informativos se muestran con los métodos `write()` y `writeln()` del objeto `Output` que Symfony2 pasa al método `execute()`:

```

// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command

```

```
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $output->writeln('Comienza el proceso de generación de emails...');

        // ...

        $output->write(array(
            'Generados 10 emails',
            'Comienza el envío de los mensajes',
            'Conectando con el servidor de correo...'
        ));

        // ...
    }
}
```

Los dos métodos admiten como primer argumento un cadena de texto o un array con el mensaje o mensajes que se quieren mostrar. La diferencia entre los dos métodos es que `writeln()` siempre añade un salto de línea `\n` después de cada mensaje, de ahí su nombre.

El método `write()` no añade el salto de línea automáticamente, pero puedes añadirlo si pasas el valor `true` como segundo argumento. Por tanto, estas dos instrucciones son equivalentes:

```
$output->writeln('Comienza el proceso de generación de emails...');
$output->write('Comienza el proceso de generación de emails...', true);
```

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 Symfony2:

```
$output->write(array(
    'Generados <info>10</info> emails',
    '<comment>Comienza el envío de los mensajes</comment>',
    '<info>Conectando</info> con el <comment>servidor de correo</comment>...'
));
```

Symfony2 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:

```
$output->write('Generados <fg=magenta;bg=yellow>10</> emails');
$output->write('Generados <fg=red;option=underscore>10</> emails');
$output->write('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), `underline` (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 Symfony2.

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 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 o para solicitar una cantidad variable de información que no se puede obtener mediante opciones y argumentos.

Los comandos de Symfony2 disponen de tres métodos para realizar preguntas: `ask()`, `askConfirmation()` y `askAndValidate()`. A todos ellos se accede a través de la clase `Dialog`.

El método más sencillo es `askConfirmation()`, que solicita la confirmación del usuario realizando una pregunta hasta que este contesta `y` o `n`:

```
// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        // ...

        $dialog = $this->getHelperSet()->get('dialog');
        $respuesta = $dialog->askConfirmation($output, '<question>¿Quieres
enviar ahora todos los emails?</question>', 'n');
```

```
// ...
}
}
```

El primer argumento de `askConfirmation()` es el objeto `$output` con el que se muestran los mensajes en la consola. El segundo argumento es la pregunta que se realiza al usuario, que también se puede formatear con cualquiera de los estilos explicados en la sección anterior. El último argumento, que por defecto vale `true`, es el valor booleano que se asigna como respuesta cuando el usuario no escribe nada y simplemente pulsa la tecla `<Enter>`.

Como las únicas respuestas válidas son `y` (de `yes`), `n` (de `no`) y `<Enter>`, puede resultar interesante mostrar las posibles respuestas y su valor por defecto junto a la pregunta. Esto facilita mucho el uso del comando a los usuarios que no están acostumbrados a utilizar el inglés para responder a las preguntas:

```
// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        // ...

        $dialog = $this->getHelperSet()->get('dialog');
        $respuesta = $dialog->askConfirmation($output, '<question>¿Quieres
enviar ahora todos los emails? (y, n)</question> [<comment>y</comment>]', 
false);

        // ...
    }
}
```

Mientras el usuario no escriba `y`, `n` o pulse `<Enter>` la pregunta se repite una y otra vez. El método `askConfirmation()` devuelve `true` cuando la respuesta a la pregunta es `y`, no importa si ha sido escrita por el usuario o es su valor por defecto. De la misma forma, devuelve `false` cuando la respuesta a la pregunta es `n`. Así, puedes utilizar el siguiente código para detener el comando cuando el usuario no da su conformidad:

```
if (!$dialog->askConfirmation($output, '¿Quieres enviar los emails?', false)) {
    return;
}
```

El método `ask()` es una generalización de `askConfirmation()`, ya que sus posibles respuestas no están limitadas a `y` o `n`, sino que se acepta cualquier respuesta:

```
// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command
{
```

```
// ...

protected function execute(InputInterface $input, OutputInterface $output)
{
    // ...

    $dialog = $this->getHelperSet()->get('dialog');
    $ciudad = $dialog->ask($output, '<question>¿Para qué ciudad quieres
generar los emails?</question>', 'sevilla');

    // ...
}
```

Los argumentos del método `ask()` son idénticos a los del método `askConfirmation()` anterior. En este caso la pregunta nunca se repite, ya que cualquier valor que introduzca el usuario se considera válido. El método `ask()` aplica el método `trim()` antes de devolver la respuesta del usuario, por lo que no debes preocuparte de los posibles espacios en blanco y del `<Enter>` del final.

Por último, el método `askAndValidate()` es el más avanzado y complejo, ya que además de realizar preguntas es capaz de validar las respuestas del usuario. El método `askAndValidate()` admite cinco parámetros con los siguientes valores por defecto:

```
askAndValidate(OutputInterface $output, $pregunta, $validador, $intentos =
false, $por_defecto = null)
```

- `$output`, es como siempre el objeto que permite mostrar mensajes en la consola.
- `$pregunta`, es la pregunta que se formula al usuario. En este caso también se pueden utilizar las etiquetas para formatear el estilo del texto.
- `$validador`, es el *callback* utilizado para validar la respuesta del usuario. Se puede utilizar cualquier *callback* de PHP, incluyendo las funciones anónimas.
- `$intentos`, es el máximo número de veces que se repite la pregunta mientras la respuesta no se considere válida. El valor por defecto es `false`, que significa infinitos reintentos.
- `$por_defecto`, es el valor por defecto de la pregunta y por tanto, el valor que se utiliza cuando el usuario simplemente responde pulsando la tecla `<Enter>`.

El siguiente ejemplo muestra cómo realizar una pregunta al usuario y validar que su respuesta sea un número:

```
$dialog = $this->getHelperSet()->get('dialog');
$numero = $dialog->askAndValidate(
    $output,
    '¿Cuántos emails quieres enviar? ',
    function($valor) {
        if (false == is_numeric($valor)) {
            throw new \InvalidArgumentException($valor.' no es un número');
    }
}
```

```

        return $valor;
    },
    2
);

```

El código anterior pregunta al usuario *¿Cuántos emails quieres enviar?* y valida su respuesta mediante la función `is_numeric()` de PHP dentro de una función anónima.

Si la primera vez el usuario no escribe un número, se vuelve a formular la pregunta. Si la segunda vez vuelve a responder con algo que no sea un número, se detiene la ejecución del comando, ya que el máximo número de intentos se ha establecido en `2`.

El *callback* que hace de validador recibe como primer parámetro la respuesta introducida por el usuario. El código del *callback* puede ser tan sencillo o tan complejo como desees, pero siempre debe cumplir dos condiciones:

- Cuando la respuesta del usuario no se considere válida, no se devuelve `false`, sino que se lanza una excepción de tipo `InvalidArgumentException`.
- El *callback* siempre devuelve la respuesta del usuario (`return $valor;`). No es necesario que sea exactamente igual que lo que introdujo el usuario, ya que el *callback* puede modificar o corregir su contenido.

Cuando la validación es compleja, no es recomendable *ensuciar* el código del comando utilizando una función anónima. Es esos casos es mejor crear una clase que agrupe todos los validadores de los comandos e invocarlos mediante la notación `array(namespace-de-la-clase, nombre-del-metodo)`, tal y como muestra el siguiente ejemplo del código fuente de Symfony2:

```

// vendor/bundles/Sensio/Bundle/GeneratorBundle/Command/
GenerateBundleCommand.php
// ...
$namespace = $dialog->askAndValidate(
    $output,
    $dialog->getQuestion('Bundle namespace', $input->getOption('namespace')),
    array('Sensio\Bundle\GeneratorBundle\Command\Validators',
        'validateBundleNamespace'),
    false,
    $input->getOption('namespace')
);

$input->setOption('namespace', $namespace);

// vendor/bundles/Sensio/Bundle/GeneratorBundle/Command/Validators.php
class Validators
{
    static public function validateBundleNamespace($namespace)
    {
        if (!preg_match('/Bundle$/i', $namespace)) {

```

```

        throw new \InvalidArgumentException(
            'The namespace must end with Bundle.'
        );
    }

    $namespace = strtr($namespace, '/', '\\');
    if (!preg_match(
        '/^(:[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*\\?)+$/',
        $namespace
    )) {
        throw new \InvalidArgumentException(
            'The namespace contains invalid characters.'
        );
    }

    // ...

    return $namespace;
}
}

```

Una buena práctica relacionada con las preguntas al usuario consiste en agruparlas todas bajo el método `interact()` del comando. Este método 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/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends Command
{
    // ...

    protected function interact(InputInterface $input, OutputInterface $output)
    {
        $output->writeln(array(
            'Bienvenido al generador de emails',
            '',
            'Para continuar, debes contestar a varias preguntas...'
        ));

        $dialog = $this->getHelperSet()->get('dialog');

        $ciudad = $dialog->ask($output,
            '¿Para qué ciudad quieres generar los emails? ',
            'sevilla'
        );
        $input->setArgument('ciudad', $ciudad);
    }
}

```

```

$accion = $dialog->askAndValidate($output,
    '¿Quéquieres hacer con los emails? (generar o enviar) ',
    function($valor) {
        if (!in_array($valor, array('generar', 'enviar'))) {
            throw new \InvalidArgumentException(
                'La acción sólo puede ser "generar" o "enviar"'
            );
        }
        return $valor;
    });
$input->setOption('accion', $accion);

if (!$dialog->askConfirmation($output,
    sprintf('¿Quieres %s ahora los emails de %s?', $accion, $ciudad),
    true
)) {
    // ...
}
}

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 581)):

```

// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
// ...
use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;

class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $contenedor = $this->getContainer();
        $em = $contenedor->get('doctrine')->getManager();

```

```

// Obtener el listado de usuarios que permiten el envío de email
$usuarios = $em->getRepository('UsuarioBundle:Usuario')
    ->findBy(array('permite_email' => true));

$output->writeln(sprintf(
    'Se van a enviar <info>%s</info> emails', count($usuarios)
));

// Buscar la 'oferta del día' en todas las ciudades de la aplicación
$ofertas = array();
$ciudades = $em->getRepository('CiudadBundle:Ciudad')->findAll();
foreach ($ciudades as $ciudad) {
    $id    = $ciudad->getId();
    $slug = $ciudad->getSlug();

    $ofertas[$id] = $em->getRepository('OfertaBundle:Oferta')
        ->findOfertaDelDiaSiguiente($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/Cupon/OfertaBundle/Entity/OfertaRepository.php
class OfertaRepository extends EntityRepository
{
    // ...

    public function findOfertaDelDiaSiguiente($ciudad)
    {
        $em = $this->getEntityManager();

        $consulta = $em->createQuery(
            'SELECT o, c, t
             FROM OfertaBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t
             WHERE o.revisada = true
             AND o.fecha_publicacion < :fecha
             AND c.slug = :ciudad
             ORDER BY o.fecha_publicacion 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, se utiliza el mismo método `render()` que en los controladores, pasando como primer argumento el nombre de la plantilla y opcionalmente un segundo argumento con las variables que se pasan a la plantilla. Puedes obtener el método `render()` a través del servicio `twig`:

```

// src/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    // ...

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $contenedor = $this->getContainer();

        $contenido = $contenedor->get('twig')->render(
            'BackendBundle:Oferta:email.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 `email.html.twig` en el *bundle BackendBundle* y copia el siguiente código:

```

#{ src/Cupon/BackendBundle/Resources/views/Oferta/email.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>
        
    </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, oferta.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
        estás suscrito al servicio de envío de <em>"La oferta del día"</em>. Para darte
        de 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/javierregiluz/Cupon/blob/2.3/src/Cupon/BackendBundle/Resources/views/Oferta/email.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/Cupon/BackendBundle/Command/EmailOfertaDelDiaCommand.php
class EmailOfertaDelDiaCommand extends ContainerAwareCommand
{
    protected function configure()

```

```
{  
    // ...  
}  
  
protected function execute(InputInterface $input, OutputInterface $output)  
{  
    $host = 'dev' == $input->getOption('env') ?  
        'http://cupon.local' :  
        'http://cupon.com';  
  
    $accion = $input->getOption('accion');  
  
    $contenedor = $this->getContainer();  
    $em = $contenedor->get('doctrine')->getManager();  
  
    $usuarios = $em->getRepository('UsuarioBundle:Usuario')->findBy(...);  
  
    // Buscar la 'oferta del día' en todas las ciudades de la aplicación  
    $ofertas = array();  
    $ciudades = $em->getRepository('CiudadBundle:Ciudad')->findAll();  
    foreach ($ciudades as $ciudad) {  
        $id = $ciudad->getId();  
        $slug = $ciudad->getSlug();  
  
        $ofertas[$id] = $em->getRepository('OfertaBundle:Oferta')  
            ->findOfertaDelDiaSiguiente($slug);  
    }  
  
    // Generar el email personalizado de cada usuario  
    foreach ($usuarios as $usuario) {  
        $ciudad = $usuario->getCiudad();  
        $oferta = $ofertas[$ciudad->getId()];  
  
        $contenido = $contenedor->get('twig')->render(  
            'BackendBundle:Oferta:email.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 Symfony2 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 Symfony2. 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/Cupon/BackendBundle/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(
    'BackendBundle:Oferta:email.txt.twig',
    array( ... )
);

$html = $contenedor->get('twig')->render(
    'BackendBundle:Oferta:email.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:   ~
```

Si no utilizas el configurador web de Symfony2 o no quieres hacer uso del archivo `parameters.yml`, puedes realizar toda la configuración directamente en el archivo `config.yml`.

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. Symfony2 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. Symfony2 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/`).

CAPÍTULO 19

Mejorando el rendimiento

Según sus propios creadores, en aplicaciones web reales Symfony2 es decenas de veces más rápido que symfony 1 y significativamente más rápido que la mayoría de frameworks PHP. Aún así, existen opciones de configuración, técnicas y trucos que pueden mejorar todavía más el rendimiento de tus aplicaciones.

19.1 Desactivando los elementos que no utilizas

Symfony2 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. Si por ejemplo no utilizas anotaciones para definir la seguridad de la aplicación, elimina el *bundle* `JMS\SecurityExtraBundle`.
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 PHP, desactiva Twig como motor de plantillas:

```
# app/config/config.yml
framework:
    templating: { engines: ['php'] }    # 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 }
```

19.2 Mejorando la carga de las clases

El primer concepto importante relacionado con la carga de clases es el archivo `bootstrap.php.cache` que se encuentra en el directorio `app/`. Se trata de un archivo gigantesco en el que se incluye el código de las clases más utilizadas por Symfony2.

Así cada vez que se recibe una petición, Symfony2 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\ClassLoader\ApcClassLoader;
use Symfony\Component\HttpFoundation\Request;

$loader = require_once __DIR__ . '/../app/bootstrap.php.cache';

// ...
```

Cada vez que instalas o actualizas los `vendors` Symfony2 regenera automáticamente este archivo. Si por cualquier circunstancia quieres regenerar el archivo manualmente, accede al directorio raíz del proyecto y ejecuta el siguiente comando:

```
$ 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`, Symfony2 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:

```
$ php composer.phar 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 Symfony2 utilice la cache APC de PHP. Si tienes instalada y configurada correctamente la extensión APC en tu servidor PHP, descomenta las dos siguientes líneas que se encuentran al principio del archivo `web/app.php`:

```
// web/app.php
// ...

$loader = new ApcClassLoader('sf2', $loader);
$loader->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. Para asegurarte de que este prefijo no entra en conflicto con el prefijo de ninguna otra aplicación del servidor, puedes cambiar su valor por algo único para tu aplicación:

```
// web/app.php
// ...

$loader = new ApcClassLoader('cupon_', $loader);
$loader->register(true);
```

19.3 Mejorando el rendimiento del enrutamiento

El sistema de enrutamiento es otra de las partes que más afectan al rendimiento, ya que en cada petición se compara la URL solicitada por el usuario con las expresiones regulares de todas las rutas de la aplicación.

Si accedes al directorio de la caché del entorno en el que estás ejecutando la aplicación (por ejemplo `/app/cache/dev/`) verás dos archivos relacionados con el enrutamiento. El primero se llama `appdevUrlMatcher.php` y se encarga de decidir cuál es la ruta que corresponde a la URL solicitada por el usuario. Si observas su código fuente, verás la gran cantidad de comparaciones con expresiones regulares que incluye:

```
use Symfony\Bundle\FrameworkBundle\Routing\RedirectableUrlMatcher;

class appdevUrlMatcher extends RedirectableUrlMatcher
{
    // ...

    public function match($pathinfo)
    {
        // ...

        // ciudad_recientes
        if (preg_match('#^/(?P<_locale>en|es)/(?P<ciudad>[^/]+)/recientes#$s', $pathinfo, $matches)) {
            return $this->mergeDefaults(array_replace(
                $matches,
                array('_route' => 'ciudad_recientes'),
                array('_controller' =>
'Cupon\\CiudadBundle\\Controller\\DefaultController::recientesAction',
'_locale' => 'es',
));
        }
    }
}
```

```
// ...
}
}
```

El segundo archivo se llama `appdevUrlGenerator.php` y se encarga de generar las URL a partir de las rutas y variables que se le indican, como por ejemplo cada vez que utilizas la función `path()` en una plantilla de Twig:

```
use Symfony\Component\Routing\Generator\UrlGenerator;

class appdevUrlGenerator extends UrlGenerator
{
    static private $declaredRouteNames = array(
        // ...
        'ciudad_recientes' => array (
            0 => array(0 => '_locale', 1 => 'ciudad',),
            1 => array('_controller' =>
'Cupon\\CiudadBundle\\Controller\\DefaultController::recientesAction',
'_locale' => 'es',),
            2 => array('_locale' => 'en|es',),
            3 => array(
                0 => array(0 => 'text', 1 => '/recientes',),
                1 => array(0 => 'variable', 1 => '/', 2 => '[^/]++', 3 =>
'ciudad',),
                2 => array(0 => 'variable', 1 => '/', 2 => 'en|es', 3 =>
'_locale',),
            ),
            4 => array(),
        ),
        // ...
    );

    public function generate($name, $parameters = array(), $referenceType =
self::ABSOLUTE_PATH)
    {
        if (!isset(self::$declaredRoutes[$name])) {
            throw new RouteNotFoundException(sprintf('Unable to generate a URL
for the named route "%s" as such route does not exist.', $name));
        }

        list($variables, $defaults, $requirements, $tokens, $hostTokens) =
self::$declaredRoutes[$name];

        return $this->doGenerate($variables, $defaults, $requirements, $tokens,
$parameters, $name, $referenceType, $hostTokens);
    }

    // ...
}
```

La clase que genera las rutas es difícil de optimizar, más allá de las pequeñas mejoras que se pueden incluir en su código PHP. Sin embargo, la primera clase encargada de determinar la ruta que corresponde a cada URL se puede optimizar con ayuda del servidor web.

En primer lugar, abre el archivo de configuración del entorno de producción y añade las dos siguientes opciones para hacer que tu aplicación utilice la clase `ApacheUrlMatcher` en vez de la clase `UrlMatcher` mostrada anteriormente:

```
# app/config/config_prod.yml
parameters:
    router.options.matcher.cache_class: ~
    router.options.matcher_class: Symfony\Component\Routing\Matcher\
ApacheUrlMatcher
```

A continuación, utiliza el comando `router:dump-apache` para convertir las rutas de tu aplicación en reglas interpretables por el módulo `mod_rewrite` de Apache:

```
$ php app/console router:dump-apache -e=prod --no-debug
```

El comando muestra por pantalla todo el volcado de las rutas de Symfony2 a reglas de Apache:

```
$ php app/console router:dump-apache

# skip "real" requests
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule .* - [QSA,L]

# ...

# ciudad_recientes
RewriteCond %{REQUEST_URI} ^/(en|es)/([^.]+)/recientes$
RewriteRule .* app.php
[QSA,L,E=_ROUTING_route:ciudad_recientes,E=_ROUTING_param__locale:%1,E=_ROUTING_pa

# ...

# 405 Method Not Allowed
RewriteCond %{_ROUTING__allow_POST} !-z
RewriteRule .* app.php [QSA,L]
```

Por último, copia todas estas reglas y pégalas en el archivo de configuración del `VirtualHost` de Apache correspondiente a la aplicación. Reinicia el servidor web para que se apliquen los cambios y la aplicación ya no utilizará la clase `appdevUrlMatcher` para traducir rutas en controladores, sino que será el propio módulo `mod_rewrite` de Apache el encargado de esa tarea.

Cada vez que añadas, elimines o modifiques una ruta de la aplicación, tendrás que volver a volcar las rutas con el comando `router:dump-apache` y copiarlas en el archivo de configuración de Apache.

Para mejorar todavía más el rendimiento de la aplicación, modifica el controlador frontal de producción para que utilice la clase `ApacheRequest` en vez de la tradicional clase `Request`:

```
// web/app.php

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
//require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\ApacheRequest;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
//$kernel = new AppCache($kernel);
$kernel->handle(ApacheRequest::createFromGlobals())->send();
```

Si en tu proyecto no puedes hacer uso del comando `router:dump-apache`, recuerda que Symfony2 dispone de un archivo `.htaccess` en el directorio `web/`. Para mejorar el rendimiento de la aplicación, pasa su contenido al archivo de configuración de Apache y deshabilita la búsqueda de archivos `.htaccess`:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ app.php [QSA,L]
</IfModule>
```

19.4 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 si después el usuario debe esperar 10 segundos a que se carguen las imágenes, hojas de estilos y archivos JavaScript. Mejorar el rendimiento de la parte del cliente debe ser una prioridad en todas las aplicaciones web, ya que el esfuerzo requerido es mínimo y la mejora conseguida es muy significativa.

Symfony2 incluye para ello la utilidad Assetic (<https://github.com/kriswallsmith/assetic>) , un gestor de *assets* o archivos web (CSS, JS, imágenes) muy avanzado creado por Kris Wallsmith y basado en la librería webassets (<http://elsdoerfer.name/files/docs/webassets/>) de Python.

19.4.1 Combinando y minimizando las hojas de estilo

Assetic es compatible con una gran variedad de utilidades y herramientas a través de sus filtros. Sin embargo, no incluye el código de ninguna librería o herramienta externa. Por eso, siempre que quieras utilizar un determinado filtro primero tienes que descargar e instalar su herramienta asociada.

Las hojas de estilo CSS se minimizan con un filtro llamado `yui_css` que utiliza la librería *YUI compressor* de Yahoo!. Así que antes de configurar Assetic, baja el código de *YUI compressor* que se encuentra en <http://yuilibrary.com/download/yuicompressor/>

La librería completa de *YUI compressor* se encuentra comprimida en un archivo `.jar` de Java. Para utilizarlo en tu proyecto Symfony2, crea el directorio `app/Resources/java/` y copia en su interior el archivo `build/yuicompressor-2.4.7.jar` que encontrarás al descomprimir el archivo descargado (el número de versión de *YUI compressor* puede variar).

NOTA Como *YUI compressor* es una aplicación Java, antes de utilizarlo asegúrate de que Java se encuentra instalado en el servidor en el que se ejecuta la aplicación.

Después de descargar e instalar *YUI compressor*, Assetic ya puede utilizar el filtro `yui_css` para minimizar hojas de estilos. Para ello, añade la siguiente configuración en el archivo `config.yml` de la aplicación:

```
# app/config/config.yml
assetic:
    debug:          %kernel.debug%
    use_controller: false
    filters:
        cssrewrite: ~
        yui_css:
            jar:    %kernel.root_dir%/Resources/java/yuicompressor-2.4.7.jar
```

Si no tienes instalado Java en `/usr/bin/java`, también debes indicar la ruta de su ejecutable:

```
# app/config/config.yml
assetic:
    debug:          %kernel.debug%
    use_controller: false
    java:           /usr/bin/java
    # ...
```

La configuración anterior activa el filtro `yui_css`, pero no comprime automáticamente las hojas de estilo. Para comprimirlas es necesario modificar ligeramente la plantilla que enlaza con los CSS. En el caso de las páginas del *frontend*, estos archivos se enlazan en la plantilla `frontend.html.twig`:

```
{# app/Resources/views/frontend.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
<link href="{{ asset('bundles/oferta/css/normalizar.css') }}" rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/oferta/css/comun.css') }}" rel="stylesheet" type="text/css" />
<link href="{{ asset('bundles/oferta/css/frontend.css') }}" />
```

```
    rel="stylesheet" type="text/css" />
{%- endblock %}

{# ... #}
```

Assetic define una etiqueta de Twig llamada `{% stylesheets %}` para indicar que las hojas de estilo se minimizan antes de servirlas:

```
{# app/Resources/views/frontend.html.twig #-}

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

{% block stylesheets %}
{% stylesheets '@OfertaBundle/Resources/public/css/*' filter='yui_css' %}
    <link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
{% endstylesheets %}
{% endblock %}

{# ... #}
```

El primer argumento de la etiqueta `{% stylesheets %}` es la ruta o colección de rutas de las hojas de estilo CSS que se minimizan. Los filtros que se aplican a los archivos CSS se indican mediante la opción `filter`. El código anterior hace que Assetic busque todos los archivos que se encuentran en el directorio `Resources/views/css/` del *bundle* `OfertaBundle` y les aplique el filtro `yui_css` configurado anteriormente.

Observa cómo la ruta de los archivos CSS se ha escrito en *notación bundle*. Si añades el carácter `@` por delante de la ruta, Symfony2 no lo interpreta como una ruta del sistema de archivos, sino que considera que la primera palabra es el nombre del *bundle* y el resto es la ruta relativa dentro del *bundle*.

En el interior de la etiqueta `{% stylesheets %}` se define el código HTML con el que se enlaza la hoja de estilo. La URL del archivo CSS creado por Assetic está disponible en una variable especial llamada `asset_url`. Si Assetic genera varios archivos CSS, se repite este código para cada archivo, cambiando cada vez el valor de la variable `asset_url`.

El problema del código anterior es que Assetic busca todos los archivos CSS del *bundle* `OfertaBundle` y después los incluye por orden alfabético. Esto es un grave error, ya que en las aplicaciones web profesionales el orden en el que se incluyen los archivos CSS es esencial para asegurar que la aplicación tiene el aspecto deseado.

Así que modifica el código anterior por lo siguiente para poder especificar uno por uno los archivos CSS incluidos y el orden deseado:

```
{# app/Resources/views/frontend.html.twig #-}

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

{% block stylesheets %}
{% stylesheets '@OfertaBundle/Resources/public/css/normalizar.css'
    '@OfertaBundle/Resources/public/css/comun.css'

```

```

'@OfertaBundle/Resources/public/css/frontend.css'
filter='yui_css' %}
<link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
{%- endstylesheets %}
{%- endblock %}

{# ... #}

```

Cuando indiques varias rutas en la etiqueta `{% stylesheets %}` puedes seguir utilizando la *notación bundle*, pero no olvides separar los archivos entre sí mediante un espacio en blanco.

Si cargas ahora cualquier página del *frontend* verás el siguiente código HTML en la parte de la cabecera:

```

<!-- ... -->
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, cada día ofertas increíbles en tu ciudad con descuentos de
hasta el 90% | Cupon</title>
    <link href="/app_dev.php/css/c3afcc6_part_1_comun_1.css" rel="stylesheet"
type="text/css" />
    <link href="/app_dev.php/css/c3afcc6_part_1_frontend_2.css"
rel="stylesheet" type="text/css" />
    <link href="/app_dev.php/css/c3afcc6_part_1_normalizar_3.css"
rel="stylesheet" type="text/css" />
    <link rel="shortcut icon" href="/favicon.ico" />
</head>
<!-- ... -->

```

Assetic crea por cada hoja de estilos original una nueva CSS con su contenido minimizado. Estas nuevas hojas de estilo se sirven dinámicamente a través de rutas de tipo `/app_dev.php/css/*`. Assetic crea estas rutas automáticamente, como puedes comprobar ejecutando el comando `route:debug` de Symfony2:

```
$ php app/console route:debug

[router] Current routes
Name           Method Pattern
_assetic_c3afcc6 ANY    /css/c3afcc6.css
_assetic_c3afcc6_0 ANY   /css/c3afcc6_part_1_comun_1.css
_assetic_c3afcc6_1 ANY   /css/c3afcc6_part_1_frontend_2.css
_assetic_c3afcc6_2 ANY   /css/c3afcc6_part_1_normalizar_3.css
...
```

Las reglas `_assetic_*` se definen en el archivo `app/config/routing_dev.yml`, por lo que es mejor que no toques esas reglas a menos que sepas bien lo que estás haciendo:

```
# app/config/routing_dev.yml
_assetic:
    resource: .
    type:     assetic

# ...
```

Si ejecutas el comando `route:debug` pasándole el nombre de una ruta, se muestra toda su información relacionada. Así se comprueba que estas rutas se procesan en un controlador especial proporcionado por Assetic:

```
$ php app/console route:debug _assetic_c3afcc6

[router] Route "_assetic_c3afcc6"
Name      _assetic_c3afcc6
Path      /css/c3afcc6.css
Host      ANY
Scheme    ANY
Method    ANY
Class     Symfony\Component\Routing\Route
Defaults  _controller: assetic.controller:render
          _format: css
          name: c3afcc6
          pos: NULL
...
...
```

El uso de un controlador para servir los *assets* se configura mediante la opción `use_controller` de Assetic. Por razones de rendimiento su valor es `false` en el archivo de configuración `app/config/config.yml`, pero vale `true` en el archivo `app/config/config_dev.yml`. Esto significa que en el entorno de desarrollo los archivos CSS se sirven a través de un controlador especial y por tanto, cualquier cambio que hagas en el CSS se aplica instantáneamente.

En cualquier caso, las hojas de estilo originales se han minimizado, pero no se han combinado, ya que sigue habiendo un archivo por cada CSS. Este comportamiento se debe a que en el entorno de desarrollo, la opción `debug` de Assetic es `true`. Si cargas cualquier página en el entorno de producción, el valor de `debug` es `false` y el resultado es el siguiente:

```
<!-- ... -->
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Cupon, cada día ofertas increíbles en tu ciudad con descuentos de
hasta el 90% | Cupon</title>
    <link href="/css/c3afcc6.css" rel="stylesheet" type="text/css" />
    <link rel="shortcut icon" href="/favicon.ico" />
</head>
<!-- ... -->
```

Si no ves un código similar al anterior, borra la caché del entorno de producción y vuelve a cargar la página:

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

Ahora las hojas de estilos están minimizadas y todas ellas se han combinado en un único archivo, ahorrando varias peticiones HTTP al cargar las páginas del sitio web. Además, la hoja de estilos ya no se carga a través del controlador especial de Assetic, sino que es un enlace a un archivo estático del directorio `web/`.

Sin embargo, si visualizas la página en un navegador, verás que la página no carga ningún estilo. El motivo es que no existe la hoja de estilos enlazada (`/css/c3afcc6.css` en el ejemplo anterior). Assetic dispone de un comando llamado `dump` para crear todos los archivos necesarios:

```
$ php app/console assetic:dump --env=prod

Dumping all prod assets.
Debug mode is on.

[file+] .../cupon.local/app/.../web/css/c3afcc6.css
[file+] .../cupon.local/app/.../web/css/c3afcc6_part_1_comun_1.css
[file+] .../cupon.local/app/.../web/css/c3afcc6_part_1_frontend_2.css
[file+] .../cupon.local/app/.../web/css/c3afcc6_part_1_normalizar_3.css
```

Este comando genera las hojas de estilo individuales y la completa. Para generar solamente la hoja de estilos completa que se enlaza en producción, añade la opción `--no-debug`:

```
$ php app/console assetic:dump --env=prod --no-debug

Dumping all prod assets.
Debug mode is off.

[file+] .../cupon.local/app/.../web/css/c3afcc6.css
```

Vuelve a cargar la página en producción. La página debería tener el mismo aspecto que antes, pero con la ventaja de que ahora sólo se enlaza una única hoja de estilos y que su contenido está minimizado.

¿Qué sucede cuando se modifica el contenido de las hojas de estilo originales? En el entorno de desarrollo los cambios se visualizan inmediatamente sin tener que hacer nada, ya que las hojas de estilos se sirven a través de un controlador especial que comprueba si se han producido cambios.

Sin embargo, en el entorno de producción no se aplican los cambios de las hojas de estilos. **Cada vez que modifiques los estilos, debes ejecutar el comando `assetic:dump` para regenerar los archivos:**

```
$ php app/console assetic:dump --env=prod --no-debug
```

A pesar de que los cambios en producción no son tan numerosos como en desarrollo, esta tarea puede convertirse rápidamente en algo tedioso. Por ello, dispones de una opción llamada `--watch` que obliga al comando a entrar en un bucle infinito en el que cada segundo se comprueba si se ha modificado alguna hoja de estilos. En tal caso, se vuelve a generar automáticamente la hoja de estilos completa:

```
$ php app/console assetic:dump --env=prod --watch
```

La opción `--watch` no se puede utilizar junto con la opción `--no-debug`. Si 1 segundo te parece un intervalo de comprobación demasiado corto, se puede definir otro valor (en segundos) mediante la opción `--period`:

```
$ php app/console assetic:dump --env=prod --watch --period=60
```

Assetic genera por defecto un nombre aleatorio para la hoja de estilos completa. Si prefieres utilizar tu propio nombre, puedes hacerlo mediante la opción `output` en la plantilla de Twig:

```
{# app/Resources/views/frontend.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
  {% stylesheets '@OfertaBundle/Resources/public/css/normalizar.css'
    '@OfertaBundle/Resources/public/css/comun.css'
    '@OfertaBundle/Resources/public/css/frontend.css'
    filter='yui_css'
    output='css/frontend.css' %}
    <link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
  {% endstylesheets %}
  {% endblock %}

{# ... #}
```

Si has ejecutado previamente el comando `assetic:dump` con la opción `--watch`, el cambio de nombre de la hoja de estilos se aplica automáticamente. Si no, ejecuta de nuevo el comando `php app/console assetic:dump --env=prod --no-debug` para generar la nueva hoja de estilos. Además, no olvides borrar la caché de producción para que el cambio en la plantilla surta efecto:

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

19.4.2 Combinando y minimizando los archivos JavaScript

El tratamiento de los archivos JavaScript es idéntico al que se acaba de explicar para las hojas de estilo. De hecho, se utiliza incluso la misma herramienta para minimizar el código, *YUI Compressor*, aunque se debe configurar un nuevo filtro llamado `yui_js`:

```
# app/config/config.yml
assetic:
    # ...
    filters:
        yui_js:
            jar: %kernel.root_dir%/Resources/java/yuicompressor-2.4.7.jar
```

Después, modifica la forma en la que se incluyen archivos JavaScript en las plantillas Twig. En la aplicación que se está desarrollando, solamente se utiliza JavaScript en las páginas del *frontend* o parte pública del sitio:

```
{% block javascripts %}
<script src="{{ asset('bundles/oferta/js/frontend.js') }}"
       type="text/javascript"></script>
{% endblock %}
```

Para minimizar el archivo `frontend.js` reemplaza el código anterior por lo siguiente:

```
{% block javascripts %}
{% javascripts '@OfertaBundle/Resources/public/js/*'
              filter='yui_js'
              output='js/frontend.js' %}
    <script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
{% endblock %}
```

La etiqueta `{% javascripts %}` está definida por Assetic y funciona exactamente igual que la etiqueta `{% stylesheets %}` utilizada en la sección anterior. La URL del archivo JavaScript generado por Assetic se encuentra disponible en la variable especial `asset_url`.

Por último, recuerda borrar la caché de la aplicación (tanto en el entorno de desarrollo como en el de producción) para ver los cambios en el código HTML de la página.

19.4.3 Configuración avanzada

19.4.3.1 Deshabilitando temporalmente un filtro

En el entorno de desarrollo, las hojas de estilo no se combinan en un único archivo pero su contenido sí que se minimiza. El código CSS minimizado es muy difícil de leer y casi imposible de depurar, por lo que puede ser útil deshabilitar el filtro `yui_css` en el entorno de desarrollo.

Para ello basta con escribir el carácter `?` delante del nombre del filtro. Assetic deshabilita los filtros que empiezan por `?` siempre que el valor de la opción de configuración `debug` sea `true`, tal y como sucede en el entorno de desarrollo:

```
{# Antes #}
{% stylesheets '@OfertaBundle/Resources/public/css/*'
              filter='yui_css'
              output='css/frontend.css' %}
<link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
```

```
{% endstylesheets %}

{# Ahora #}
{% stylesheets '@OfertaBundle/Resources/public/css/*'
    filter='?yui_css'
    output='css/frontend.css' %}
<link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
{% endstylesheets %}
```

19.4.3.2 Aplicando un filtro automáticamente

Normalmente, cuando se configura un filtro de Assetic, se utiliza en todos los archivos de un determinado tipo. En los ejemplos anteriores, el filtro `yui_css` se aplica a todas las hojas de estilo del sitio. En estos casos, no es necesario añadir la opción `filter` en todas las plantillas de Twig, sino que basta con añadir una opción de configuración.

Elimina primero la opción `filter` de todas las etiquetas `{% stylesheets %}` de las plantillas de la aplicación:

```
{# Antes #}
{% stylesheets '@OfertaBundle/Resources/public/css/*'
    filter='yui_css'
    output='css/frontend.css' %}
<link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
{% endstylesheets %}

{# Ahora #}
{% stylesheets '@OfertaBundle/Resources/public/css/*'
    output='css/frontend.css' %}
<link href="{{ asset_url }}" rel="stylesheet" type="text/css" />
{% endstylesheets %}
```

A continuación, añade la opción de configuración `apply_to` en el filtro `yui_css` para indicar que debe aplicarse a todos los archivos cuyo nombre acabe en la extensión `.css`:

```
# app/config/config.yml

#
# ...

assetic:
    debug:          %kernel.debug%
    use_controller: false
    filters:
        #
        yui_css:
            jar:      %kernel.root_dir%/Resources/java/yuicompressor-2.4.7.jar
            apply_to: \.css$
```

Igualmente, si quieras aplicar el filtro `yui_js` a todos los archivos JavaScript, añade la opción `apply_to`:

```
# app/config/config.yml
# ...
assetic:
    debug:                 %kernel.debug%
    use_controller: false
    filters:
        # ...
    yui_js:
        jar:      %kernel.root_dir%/Resources/java/yuicompressor-2.4.7.jar
        apply_to: \.js$
```

19.4.3.3 Otros filtros

Los ejemplos anteriores muestran el uso de los filtros `yui_css` y `yui_js`, pero Assetic incluye literalmente decenas de filtros muy útiles para aplicaciones web:

- `closure`, minimiza los archivos Javascript utilizando el Closure Compiler (<http://code.google.com/closure/compiler/>) de Google.
- `coffee`, compila los archivos de tipo CoffeeScript (<http://jashkenas.github.com/coffee-script/>) para convertirlos en archivos JavaScript. No minimiza sus contenidos.
- `compass`, compila los archivos creados con el framework Compass (<http://compass-style.org/>) en archivos CSS normales. No minimiza sus contenidos.
- `csseembed`, embebe las imágenes dentro de los archivos CSS utilizando la notación `background:url(data:image/...;base64,...)`.
- `cssimport`, incluye en el archivo CSS el contenido de todas las hojas de estilos importadas mediante la regla `@import`.
- `cssrewrite`, corrige las URL relativas incluidas en las hojas de estilo originales para que sigan funcionando en las nuevas hojas de estilo generadas.
- `jpegoptim`, optimiza las imágenes de tipo JPEG con la utilidad `Jpegoptim` (<http://www.kokkonen.net/tjko/projects.html>).
- `jpegtran`, optimiza las imágenes de tipo JPEG con la utilidad `jpegtran` (<http://jpegclub.org/>).
- `less`, compila los archivos creados con LESS (<http://lesscss.org/>) en archivos CSS normales. Utiliza las herramientas `less.js` y `node.js`. No minimiza sus contenidos.
- `lessphp`, compila los archivos creados con LESS (<http://lesscss.org/>) en archivos CSS normales. Utiliza la herramienta `lessphp`. No minimiza sus contenidos.
- `optipng`, optimiza las imágenes de tipo PNG con la utilidad OptiPNG (<http://optipng.sourceforge.net/>).
- `pngout`, optimiza varios tipos de imágenes (PNG, JPEG, etc.) con la utilidad `PNGOUT` (<http://advsys.net/ken/utils.htm>).
- `sass`, compila los archivos creados con SASS (<http://sass-lang.com/>) en archivos CSS normales. No minimiza sus contenidos.

- [scss](#), compila los archivos creados con SCSS (<http://sass-lang.com/>) en archivos CSS normales. No minimiza sus contenidos.
- [sprockets](#), gestiona las dependencias de la herramienta Sprockets (<https://github.com/stephenson/sprockets>) de JavaScript.
- [stylus](#), compila los archivos creados con Stylus (<http://learnboost.github.com/stylus/>) en archivos CSS normales. No minimiza sus contenidos.
- [yui_css](#), minimiza los archivos CSS utilizando la herramienta YUI Compressor (<http://developer.yahoo.com/yui/compressor/>).
- [yui_js](#), minimiza los archivos JavaScript utilizando la herramienta YUI Compressor (<http://developer.yahoo.com/yui/compressor/>).

Desafortunadamente, todavía no están documentadas todas las opciones de configuración disponibles en cada filtro. Por eso es recomendable echar un vistazo a los archivos XML que definen la configuración de cada filtro y que se encuentran en [vendor/symfony/assetic-bundle/Symfony/Bundle/AsseticBundle/Resources/config/filters/](#).

Assetic permite combinar, siempre que sean compatibles entre sí, todos los filtros anteriores. Tan sólo debes indicar en la opción `filter` el nombre de varios filtros separados por comas. Si creas por ejemplo tus CSS con la utilidad LESS, puedes compilar, minimizar y combinar todos los estilos con:

```
{% stylesheets '@OfertaBundle/Resources/public/css/normalizar.less'  
    '@OfertaBundle/Resources/public/css/comun.less'  
    '@TiendaBundle/Resources/public/css/extranet.less'  
    filter='yui_css, less'  
    output='css/extranet.css' %}  
    <link href="{{ asset_url }}" rel="stylesheet" type="text/css" />  
{% endstylesheets %}
```

Si todavía no conoces o no utilizas herramientas como Compass, LESS, SASS o Stylus, aprovecha su integración perfecta con Assetic para introducirte en este nuevo mundo del diseño web. Serás capaz de hacer cosas que ni imaginas y tu productividad mejorará enormemente.

19.4.4 Comprimir imágenes

Cuando una aplicación permite a sus usuarios subir imágenes, no se puede esperar que estas imágenes hayan sido optimizadas para reducir su tamaño lo máximo posible. Afortunadamente existen multitud de herramientas que reducen automáticamente el peso de las imágenes sin provocar una pérdida de calidad apreciable.

Assetic es compatible con cuatro de estas herramientas a través de los filtros `jpegoptim`, `jpegtran`, `optipng` y `pngout`. A continuación se muestra cómo utilizar OptiPNG para reducir el tamaño de las imágenes del sitio. Como siempre, primero debes descargar, instalar y configurar la herramienta asociada con el filtro. Suponiendo que OptiPNG haya sido instalado con éxito en la ruta `/usr/local/bin/optipng`, el siguiente paso consiste en configurar un nuevo filtro de Assetic:

```
# app/config/config.yml
assetic:
    # ...
    filters:
        # ...
        optipng:
            bin: /usr/local/bin/optipng
            level: 5
```

El parámetro `bin` es necesario porque por defecto Assetic busca OptiPNG en la ruta `/usr/bin/optipng`. El parámetro `level` indica el nivel de compresión deseado. El valor por defecto de OptiPNG es `2` y cuanto más alto sea su valor, más se comprimen las imágenes pero más lento es el proceso y mayor es la pérdida de calidad.

Una vez configurado el filtro, ya se puede aplicar en cualquier imagen del sitio utilizando la etiqueta especial de Twig `{% image %}` y los parámetros `filter` y `output` habituales:

```
{# Antes #}


{# Ahora #}
{% image '/uploads/images/' ~ oferta.rutaFoto filter='optipng'
output='/images/' ~ oferta.foto %}
    
{% endimage %}
```

Cambiar todas las imágenes del sitio utilizando esta notación es algo tedioso. Por eso Assetic incluye funciones especiales de Twig que actúan de atajos muy útiles para la compresión de imágenes. Para ello, añade la opción `twig` en la configuración de Assetic:

```
# app/config/config.yml
assetic:
    filters:
        # ...

    twig:
        functions:
            optipng: ~
```

Ahora puedes optimizar cualquier imagen utilizando la función `optipng()` de Twig:

```
{# Antes #}
{% image '/uploads/images/' ~ oferta.rutaFoto filter='optipng'
output='/images/' ~ oferta.foto %}
    
{% endimage %}

{# Ahora #}
```

```

```

Si quieras modificar también el directorio en el que se guardan las imágenes optimizadas, indícalo en la propia configuración de la función Twig:

```
# app/config/config.yml
assetic:
    filters:
        # ...

twig:
    functions:
        optipng: { output: imagenes/optimizadas/*.png }
```

Assetic incluye funciones Twig para todas las herramientas de compresión de imágenes: `jpegoptim()`, `jpegtran()`, `optipng()` y `pngout()`.

19.5 Mejorando el rendimiento de Doctrine2

Doctrine es un proyecto independiente de Symfony2, 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 Symfony2 hace que Doctrine2 no utilice ninguna caché. Esto significa que en cada petición Doctrine2 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 Doctrine2**.

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 Doctrine2 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`). Si en vez de APC utilizas Memcache (<http://memcached.org/>) , puedes configurarlo de la siguiente manera:

```
# app/config/config_prod.yml
doctrine:
    orm:
        metadata_cache_driver: apc
        query_cache_driver:
            type: memcache
            host: localhost
            port: 11211
            instance_class: Memcache
            class: Doctrine\Common\Cache\MemcacheCache
```

Las opciones `host`, `port`, `instance_class` y `class` anteriores muestran sus valores por defecto, por lo que sólo debes indicarlas si necesitas cambiar esos valores. Normalmente basta con indicar el `host` y el `port`. Todas las opciones anteriores también están disponibles para APC y el resto de tipos de caché soportadas.

Además de APC y Memcache, Doctrine2 también incluye soporte para Xcache (<http://xcache.lighttpd.net/>) y para 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?

Doctrine2 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/Cupon/CiudadBundle/Entity/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 Doctrine2 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 o Xcache). Si quieras 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/Cupon/OfertaBundle/Entity/OfertaRepository.php
public function findOfertaDelDia($ciudad)
{
    $em = $this->getEntityManager();
```

```

$consulta = $em->createQuery('
    SELECT o, c, t
    FROM OfertaBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t
    WHERE o.revisada = true
        AND o.fecha_publicacion < :fecha
        AND c.slug = :ciudad
    ORDER BY o.fecha_publicacion 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 Doctrine2, 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 OfertaBundle: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. Doctrine2 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 OfertaBundle: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? Doctrine2 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 OfertaBundle: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 OfertaBundle: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 OfertaBundle:Oferta o JOIN o.ciudad c JOIN o.tienda t  
WHERE o.revisada = TRUE  
AND o.fecha_publicacion < :fecha  
AND c.slug = :ciudad  
ORDER BY o.fecha_publicacion DESC
```

19.5.3.2 Utiliza la mejor *hidratación*

Después de ejecutar una consulta, Doctrine2 aplica a cada resultado obtenido un proceso conocido como *hidratación*. Este proceso no sólo penaliza seriamente el rendimiento, sino que resulta inne-

cesario 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 Doctrine2 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 OfertaBundle:Oferta o JOIN o.ciudad c
    ORDER BY o.id ASC
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_OBJECT);
```

Resultado:

```
array(
    0 => object(Cupon\OfertaBundle\Entity\Oferta) {
        ["id":protected]=> 1
        ["nombre":protected]=> "Oferta #0-1"
        ["slug":protected]=> "oferta-0-1"
        ["descripcion":protected]=> "Lorem ipsum ..."
        ["condiciones":protected]=> "Lorem ipsum ..."
        ["ruta_foto":protected]=> "foto4.jpg"
        ["precio":protected]=> "28.13"
        ["descuento":protected]=> "9.85"
    }
)
```

```

["fecha_publicacion":protected]=> object(DateTime) { ... }
["fecha_expiracion":protected]=> object(DateTime) { ... }
["compras":protected]=> 0
["umbral":protected]=> 49
["revisada":protected]=> false
["ciudad":protected]=> object(Cupon\CiudadBundle\Entity\Ciudad) {
    ["id":protected]=> 7
    ["nombre":protected]=> "Barcelona"
    ["slug":protected]=> "barcelona"
}
["tienda":protected]=> object(Proxies\CuponTiendaBundleEntityTiendaProxy) {
    ["_entityPersister":...:private]=> object(Doctrine\ORM\Persisters\...)
    ["id":protected]=> 36
    ["nombre":protected]=> "Tienda #36"
    // ... resto de propiedades de la tienda
}

// ... 7.000 líneas más con propiedades de Doctrine2 ...

1 => object(Cupon\OfertaBundle\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, Doctrine2 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 Doctrine2: `\Doctrine\Common\Util\Debug::dump($resultado);`

Hidratación HYDRATE_ARRAY

Consulta:

```
$consulta = $em->createQuery('
    SELECT o, c
```

```

    FROM OfertaBundle: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 ...",
    "ruta_foto" => "foto4.jpg",
    "precio" => "28.13",
    "descuento" => "9.85",
    "fecha_publicacion" => object(DateTime) { ... },
    "fecha_expiracion" => 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 (Doctrine2 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 `fecha_publicacion`) 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 OfertaBundle: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_ruta_foto" => "foto4.jpg",
        "o_precio" => "28.13",
        "o_descuento" => "9.85",
        "o_fecha_publicacion" => object(DateTime) { ... },
        "o_fecha_expiracion" => 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 OfertaBundle:Oferta o JOIN o.ciudad c
        WHERE o.id = 1
');

$resultado = $consulta->getResult(\Doctrine\ORM\Query::HYDRATE_SIMPLEOBJECT);
```

Resultado:

```
array(
    0 => object(Cupon\OfertaBundle\Entity\Oferta) {
        ["id":protected]=> 1
        ["nombre":protected]=> "Oferta #0-1"
        ["slug":protected]=> "oferta-0-1"
        ["descripcion":protected]=> "Lorem ipsum ... "
        ["condiciones":protected]=> "Lorem ipsum ... "
        ["ruta_foto":protected]=> "foto4.jpg"
        ["precio":protected]=> "28.13"
        ["descuento":protected]=> "9.85"
        ["fecha_publicacion":protected]=> object(DateTime) { ... }
        ["fecha_expiracion":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 OfertaBundle: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 quieras 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_OBJECT`, 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*.

Doctrine2 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.

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 Symfony2.

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.

Symfony 1 define su propio mecanismo de caché para abarcar todos estos escenarios, por lo que puedes guardar una página entera o sólo las partes que deseas. Symfony2 va un paso más allá y, en vez de implementar un mecanismo propio de caché, expresa al límite 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 Symfony2 y el navegador del usuario:

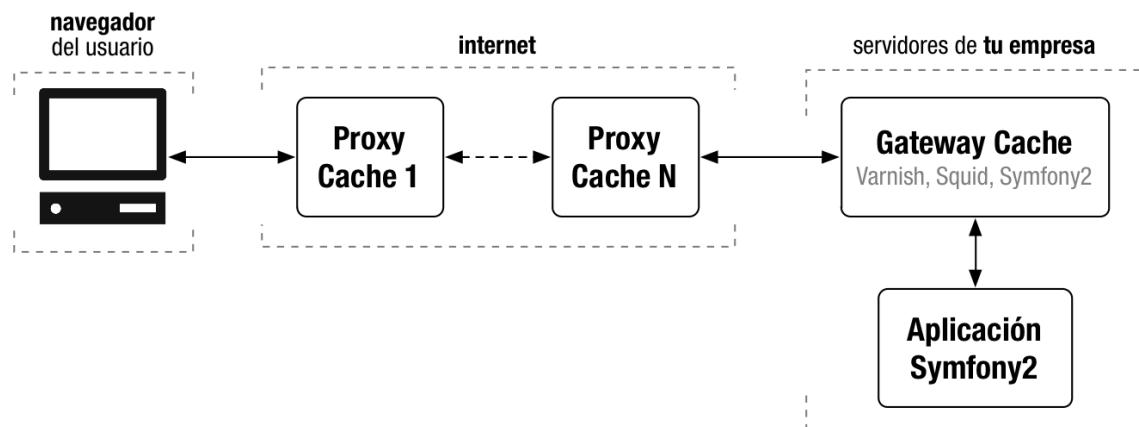


Figura 20.1 Cachés que pueden existir entre tu aplicación Symfony2 y el navegador del usuario

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 Symfony2 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('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora: (la respuesta es pública)
        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',
            array('oferta' => $oferta)
        );
        $respuesta->setPublic();
    }
}
```

```

        return $respuesta;
    }
}

```

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`. Symfony2 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 Symfony2 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:

```

class DefaultController extends Controller
{
    // ...

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',

```

```
        array('oferta' => $oferta)
    );

    $fechaCaducidad = new \DateTime('now + 5 minutes');
    $respuesta->setExpires($fechaCaducidad);

    return $respuesta;
}

}
```

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
```

El método `setExpires()` 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 método `setExpires()` con el valor `null`.

Observa cómo la respuesta también incluye la cabecera `Cache-Control` con el valor `private`. Esta cabecera la añade automáticamente Symfony2 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 ícono 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-revalidate, 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 uso más sencillo de la cabecera [Cache-Control](#) en Symfony2 consiste en utilizar los métodos [setMaxAge\(\)](#) y [setSharedMaxAge\(\)](#) del objeto [Response](#). El siguiente código hace que la caché de la portada sea privada y tenga un tiempo de vida de 5 minutos:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',
            array('oferta' => $oferta)
        );

        $respuesta->setMaxAge(5 * 60);
        return $respuesta;
    }
}
```

Por razones de legibilidad el parámetro del método `setMaxAge()`, que indica el tiempo de vida en segundos, se escribe como `5 * 60`. Si lo prefieres, puedes indicar el valor `300` directamente.

Modificar el objeto de la respuesta con el método `setMaxAge()` 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
```

El método `setMaxAge()` añade la cabecera `Cache-Control` en la respuesta para incluir la opción `max-age` con el número de segundos establecido en el controlador. Además, como no se indica el tipo de caché, Symfony2 añade la opción `private` para hacerla privada.

Después de utilizar el método `setMaxAge()` 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 ícono 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 `max-age` también puedes establecer la opción `s-maxage`. El siguiente código muestra su uso mediante el método `setSharedMaxAge()`:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',
            array('oferta' => $oferta)
        );

        $respuesta->setSharedMaxAge(5 * 60);
        return $respuesta;
    }
}
```

El método `setSharedMaxAge()` 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 `max-age` o `s-maxage` 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 Symfony2 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 Symfony2 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 Symfony2 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:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default: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`:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default: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"

```

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`. Symfony2 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 Symfony2 se establece con el método `setLastModified()` del objeto `Response`:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',
            array('oferta' => $oferta)
        );

        $fecha = $oferta->getUpdatedAt();
        $respuesta->setLastModified($fecha);

        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
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:

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

    public function portadaAction($ciudad)
    {
        // ...

        // Antes:
        // return $this->render('OfertaBundle:Default:portada.html.twig',
        //     array('oferta' => $oferta)
        // );

        // Ahora:
        $respuesta = $this->render('OfertaBundle:Default: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 Symfony2

Symfony2 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, Symfony2 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/) (<https://www.varnish-cache.org/>) y [Squid](http://www.squid-cache.org/) (<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, Symfony2 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 Symfony2

La gran ventaja del *reverse proxy* de Symfony2 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
```

```
// ...  
  
require_once __DIR__.'/../app/AppKernel.php';  
//require_once __DIR__.'/../app/AppCache.php';  
  
$kernel = new AppKernel('prod', false);  
$kernel->loadClassCache();  
//$kernel = new AppCache($kernel);  
$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 Symfony2 y se despacha la petición del usuario.

Para activar el *reverse proxy* de Symfony2, sólo tienes que descomentar las dos líneas que aparecen comentadas en el código anterior:

```
// web/app.php  
  
// ...  
  
require_once __DIR__.'/../app/AppKernel.php';  
require_once __DIR__.'/../app/AppCache.php';  
  
$kernel = new AppKernel('prod', false);  
$kernel->loadClassCache();  
$kernel = new AppCache($kernel);  
$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:

```
// app/AppCache.php  
<?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 Symfony2 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 Symfony2. 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 Symfony2, 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/Cupon/OfertaBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function portadaAction($ciudad)
    {
        // ...

        $respuesta = $this->render('OfertaBundle:Default:portada.html.twig',
            array('oferta' => $oferta)
        );

        $respuesta->setSharedMaxAge(60);

        return $respuesta;
    }
}
```

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 La página de detalle de una oferta

Esta página es muy similar a la portada del sitio, por lo que también se puede guardar en la caché pública. Como en este caso no es tan importante que se actualice frecuentemente, se puede guardar en la caché `10` minutos en vez de `1`:

```
// src/Cupon/OfertaBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function ofertaAction($ciudad, $slug)
    {
        // ...

        $respuesta = $this->render('OfertaBundle:Default:detalle.html.twig',
            array('cercanas' => $cercanas, 'oferta' => $oferta)
        );

        $respuesta->setSharedMaxAge(600);

        return $respuesta;
    }
}
```

En la máquina utilizada para las pruebas de rendimiento, este simple cambio multiplicó por dos el rendimiento de la página de detalle de una oferta.

20.3.2.3 La página de ofertas recientes en una ciudad

Como esta página muestra un simple listado de las ofertas publicadas recientemente en una ciudad, es susceptible de ser cacheada por un largo período de tiempo, como por ejemplo una hora:

```
// src/Cupon/CiudadBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function recientesAction($ciudad)
    {
        // ...

        $respuesta = $this->render(
            'CiudadBundle:Default:recientes.'.$formato.'.twig', array(
                'ciudad' => $ciudad,
                'cercanas' => $cercanas,
                'ofertas' => $ofertas
            ));
        $respuesta->setSharedMaxAge(3600);

        return $respuesta;
    }
}
```

En la máquina utilizada para las pruebas de rendimiento, el rendimiento de esta página se multiplicó por dos después de este cambio.

20.3.2.4 La página de ofertas recientes de una tienda

Esta página es similar a la anterior, ya que incluya una tabla con información sobre las ofertas que ha publicado recientemente una tienda. En este caso también se cachean sus contenidos durante una hora:

```
// src/Cupon/TiendaBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function portadaAction($ciudad, $tienda)
    {
        // ...

        $respuesta = $this->render(
            'TiendaBundle:Default:portada.'.$formato.'.twig', array(
                'tienda' => $tienda,
                'ofertas' => $ofertas,
                'cercanas' => $cercanas
            ));
        $respuesta->setSharedMaxAge(3600);
```

```
    return $respuesta;
}
}
```

En la máquina utilizada para las pruebas de rendimiento, el rendimiento de esta página también se multiplicó por dos después de este cambio.

20.3.2.5 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 Symfony2 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://cupon.local/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/UsuarioBundle/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 Symfony2.

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/UsuarioBundle/Default/cajaLogin.php"
                 alt="/app.php/OfertaBundle/Estaticas/anonimo.html"
                 onerror="continue" />
</section>

<!-- ... -->
```

Symfony2 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, cuenta con la ventaja de que si dejas de utilizar un *reverse proxy*, Symfony2 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 Symfony2. 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 Symfony2 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 Symfony2.

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('UsuarioBundle:Default:cajaLogin', { '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('UsuarioBundle:Default:cajaLogin')) }}
</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 Symfony2 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 Symfony2 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/
        UsuarioBundle%3ADefault%3AcajaLogin/id%3Dportada.html', '', true) ?>
    </section>
    <!-- ... -->
```

Por último, indica en la acción `cajaLogin()` del *bundle* `UsuarioBundle` 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/Cupon/OfertaBundle/Controller/DefaultController.php
public function portadaAction($ciudad)
{
    // ...
```

```

$respuesta = $this->render( ... );
$respuesta->setSharedMaxAge(60);

return $respuesta;
}

// src/Cupon/UsuarioBundle/Controller/DefaultController.php
public function cajaLoginAction($id = '')
{
    // ...

    $respuesta = $this->render( ... );
    $respuesta->setMaxAge(30);
    $respuesta->setPrivate();

    return $respuesta;
}

```

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.

Symfony2 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('UsuarioBundle:Default:estoNoExiste'),
    { 'alt': 'UsuarioBundle:Default: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 por defecto del *bundle* `UsuarioBundle`.

Symfony2 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 Symfony2 ignorará los errores de forma silenciosa, no mostrando nada en el lugar donde se encuentra la llamada a la función `render()`:

```
<section id="login">
{{ render_esi(
    controller('UsuarioBundle:Default:estoNoExiste'),
    {
        'alt': 'UsuarioBundle:Default: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`). Symfony2 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 quieras 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 muy rápido y eficiente. Symfony2 recomienda utilizar Twig para crear todas las plantillas de la aplicación. No obstante, si lo prefieres puedes seguir escribiendo las plantillas con código PHP normal y corriente, como en symfony 1.

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 {{ 'now' | 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,  
'UTF-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 especiales para distinguir el código Twig del resto de contenidos:

- `{{ }}` para mostrar el valor de una variable.
- `{% %}` para añadir lógica en la plantilla.
- `{# #}` para incluir un comentario.

A diferencia de otros motores de plantillas como Smarty y de otros frameworks web como Ruby On Rails, todas las etiquetas de Twig son simétricas, además de ser ligeramente más concisas:

Acción	Twig	Django	Ruby On Rails	Smarty
Incluir un comentario	<code>{# ... #}</code>	<code>{# ... #}</code>	<code><%# ... %></code>	<code>{* ... *}</code>
Mostrar una variable	<code>{{ ... }}</code>	<code>{{ ... }}</code>	<code><%= ... %></code>	<code>{\$... }</code>
Añadir lógica	<code>{% ... %}</code>	<code>{% ... %}</code>	<code><% ... %></code>	<code>{ ... }</code>

A.2 Twig para maquetadores

Twig es tan sencillo que hasta los maquetadores y diseñadores sin formación sobre programación pueden entender el funcionamiento de las plantillas. De hecho, el objetivo último de Twig es conseguir que los maquetadores y diseñadores 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 llenar 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>
```

Si eres un maquetador, lo normal es que preguntes el nombre de las variables a los programadores, que son los que las crean. No obstante, gracias a lenguajes como Twig, está surgiendo el *desarrollo basado en diseño* o DDD ("design-driven development") en el que primero se crean las plantillas y después se programa el resto de la aplicación, utilizando las variables definidas en las plantillas.

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 quieras 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`. No es necesario dejar un espacio en blanco entre la variable, la barra `|` y el filtro, pero si lo haces, la plantilla será más fácil de leer.

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 Symfony2 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(' ') }}
```

También podrías utilizar cualquier otro carácter o texto para unir los elementos:

```
 {{ producto.etiquetas | join(',') }}  
 {{ producto.etiquetas | join('-') }}  
 {{ 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 2013, mostraría '21/7/2013' #}  
 {{ 'today' | date('d/m/Y') }}  
  

{# Si además fuesen las 18:30:22, mostraría '21/7/2013 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, 2013 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 `
`.

```
{# '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 'MENÚ' #
{{ 'Menú' | upper }}

{# 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 'Menú' (no lo modifica) #
{{ 'Menú' | capitalize }}

{# 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 'Información De Contacto' #
{{ 'información de contacto' | title }}

{# 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 quieras 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 }}
```

El filtro `join` aplicado sobre la variable `meses` hace que se muestre como resultado la cadena de texto `EneroFebreroMarzo` todo junto. 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 524) 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:

```
<strong>Lorem ipsum</strong> dolor site <em>amet</em>.
```

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 o incluso negativo, este comportamiento por defecto de Twig es seguramente 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 `Lorem ipsum dolor site amet`. original.

A.2.4 Espacios en blanco

Cuando Twig crea una página a partir de una plantilla, respeta 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. `null`

En primer lugar Twig busca que en la plantilla exista un array llamado `$variable` y que contenga una clave llamada `propiedad`. Si no lo encuentra, trata de buscar un objeto llamado `$variable` que disponga de una propiedad llamada `propiedad`. Si existe el objeto pero no la propiedad, prueba con los *getters* más comunes (`propiedad()`, `getXXX()`, `isXXX()`). Por último, si no encuentra el valor de la propiedad con ninguno de los métodos anteriores, devuelve el valor `null`.

Como en las aplicaciones Symfony2 es habitual trabajar con objetos que representan entidades de Doctrine2, 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, variable3 = valor1, valor2, valor3 %}

{% 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 `-` (guion medio) en el lado por el que quieras eliminar los espacios en blanco:

Operador de Twig	Equivalente PHP	Resultado
<code>{{- variable }}</code>	<code>ltrim(variable)</code>	Elimina los espacios del lado izquierdo
<code>{} variable -}}</code>	<code>rtrim(variable)</code>	Elimina los espacios del lado derecho
<code>{}- variable -}}</code>	<code>trim(variable)</code>	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 } #}
```

A partir de la versión 1.6 de Twig, el filtro `reverse` también funciona sobre las cadenas de texto (lo cual no suele ser muy útil, a menos que seas un aficionado a los palíndromos):

```
{# Muestra: 2ynofmyS #}
{{ 'Symfony2' | reverse }}
```

`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", "email2@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&limite=10' %}
{{ consulta | url_encode }}
[# Muestra: ciudad%3Dparis%26orden%3Dascendente%26limite%3D10 #]

```

`convert_encoding`, transforma una cadena de texto a la codificación indicada. Este filtro está disponible **desde la versión 1.4 de Twig** y 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 Symfony2, 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 escapan. Para no aplicar el mecanismo de escape a ninguna variable, utiliza el valor `false`. Además, existe un tercer valor llamado `js` que aplica un escape más apropiado cuando las plantillas Twig generan JavaScript en vez de HTML.

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 }}
```

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 que contiene 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 Symfony2 y Doctrine2, las colecciones de objetos que devuelven las búsquedas de Doctrine2 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
<code>loop.index</code>	Número de iteración, siendo 1 la primera (1, 2, 3, ... N)

Propiedad	Contenido
<code>loop.index0</code>	Número de iteración, siendo <code>0</code> la primera (<code>0, 1, 2, ... N-1</code>)
<code>loop.revindex</code>	Número de iteraciones que faltan, siendo <code>1</code> la primera (<code>N, N-1, N-2, ... 1</code>)
<code>loop.revindex0</code>	Número de iteraciones que faltan, siendo <code>0</code> la primera (<code>N-1, N-2, N-3, ... 0</code>)
<code>loop.first</code>	<code>true</code> si es la primera iteración, <code>false</code> en cualquier otro caso
<code>loop.last</code>	<code>true</code> si es la última iteración, <code>false</code> en cualquier otro caso
<code>loop.length</code>	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 #}





```

```

{# ... #}
</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 divisibleby(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
<code>constant(valor)</code>	Comprueba si la variable contiene un valor igual a la constante indicada	<code>constant(\$valor) === \$variable</code>
<code>defined</code>	Comprueba que la variable haya sido definida	<code>isset(\$variable)</code>
<code>divisibleby(numero)</code>	Comprueba si la variable es divisible por el valor indicado	<code>0 == \$variable % \$numero</code>
<code>empty</code>	Comprueba si la variable está vacía	<code>false === \$variable (empty(\$variable) && '0' != \$variable)</code>
<code>even</code>	Comprueba si la variable es un número par	<code>0 == \$variable % 2</code>
<code>odd</code>	Comprueba si la variable es un número impar	<code>1 == \$variable % 2</code>
<code>iterable</code>	Comprueba si la variable es una colección de valores sobre la que se puede iterar	<code>\$variable instanceof Traversable is_array(\$variable)</code>
<code>none</code>	Es un alias del test <code>null</code>	
<code>null</code>	Comprueba si la variable es <code>null</code>	<code>null === \$variable</code>
<code>sameas(valor)</code>	Comprueba si una variable es idéntica a otra	<code>\$variable === \$valor</code>

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 <code>true</code> solamente si los dos operandos de la expresión son <code>true</code> Ejemplo: <code>{% if usuario.registrado and usuario.edad > 18 %}</code>
&&	Notación alternativa del operador <code>and</code>
or	Devuelve <code>true</code> si alguno de los dos operandos de la expresión son <code>true</code> Ejemplo: <code>{% if promocionGlobal or producto.enPromocion %}</code>
	Notación alternativa del operador <code>or</code>
not	Devuelve el valor contrario de la expresión evaluada <code>{% if not ultimoElemento %}</code>

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 <code>true</code> si los dos operandos son iguales Ejemplo: <code>{% if pedido.tipo == 'urgente' %}</code>
!=	Devuelve <code>true</code> si los dos operandos son diferentes Ejemplo: <code>{% if pedido.tipo != 'urgente' %}</code>
>	Devuelve <code>true</code> si el primer operando es mayor que el segundo Ejemplo: <code>{% if usuario.edad > 18 %}</code>
<	Devuelve <code>true</code> si el primer operando es menor que el segundo Ejemplo: <code>{% if producto.stock < 10 %}</code>
>=	Devuelve <code>true</code> si el primer operando es mayor o igual que el segundo Ejemplo: <code>{% if credito >= limite %}</code>
<=	Devuelve <code>true</code> si el primer operando es menor o igual que el segundo Ejemplo: <code>{% if producto.stock <= umbral %}</code>

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
<code>+</code>	Suma dos números o el valor de dos variables: <code>{{ 3 + 2 }}</code> = 5 Ejemplo de uso: <code>Total: {{ precio + impuestos }}</code>
<code>-</code>	Resta dos números o el valor de dos variables: <code>{{ 1 - 5 }}</code> = -4 Ejemplo de uso: <code>Total: {{ precio - descuento }}</code>
<code>*</code>	Multiplica dos números o el valor de dos variables: <code>{{ 2 * 7 }}</code> = 14 Ejemplo de uso: <code>Subtotal: {{ precioUnitario * unidades }}</code>
<code>/</code>	Divide dos números o el valor de dos variables: <code>{{ 12 / 5 }}</code> = 2.4 Ejemplo de uso: <code>Precio medio: {{ total / numeroArticulos }}</code>
<code>//</code>	División entera de dos números o del valor de dos variables: <code>{{ 12 // 5 }}</code> = 2 Ejemplo de uso: <code>Jugadas restantes: {{ dinero // precioPorApuesta }}</code>
<code>%</code>	Módulo o resto de la división entera de dos números o del valor de dos variables: <code>{{ 7 % 2 }}</code> = 1
<code>**</code>	Eleva el primer número a la potencia del segundo número: <code>{{ 5 ** 2 }}</code> = 25, <code>{{ 7 ** 3 }}</code> = 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'
}}

```

A.3.7 Otras funciones y etiquetas

Twig tiene una actividad de desarrollo tan frenética que cada pocos meses se lanza una nueva versión. Afortunadamente, todas las versiones son retrocompatibles, por lo que tu aplicación seguirá funcionando bien sin cambios. No obstante, si no repasas de vez en cuando la [documentación oficial de Twig](http://twig.sensiolabs.org/documentation) (<http://twig.sensiolabs.org/documentation>) puedes que te estés perdiendo algunas características interesantes. A continuación se muestran las funciones y etiquetas más recientes añadidas por Twig.

`date()`, además del filtro `date`, Twig incluye a partir de su versión 1.6 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 pasárle 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') }}

```

`do`, etiqueta especial que se emplea para evaluar una expresión sin mostrar nada como resultado:

```

{# en la pantalla se muestra el número '2' #}
{{ 1 + 1 }}

```

```
#{# Twig hace la suma, pero no muestra nada por pantalla #}
{% do 1 + 1 %}
```

El uso de la etiqueta `do` puede parecer absurdo, pero en ocasiones puede ser interesante para manipular información sin mostrar nada por pantalla:

```
{% do menuNavegacion.append({ 'titulo': ' url' }) %}
```

`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 dia</h1>
    {# ... #}
{% endblock %}
```

Cuando una plantilla hereda de otra, su primera etiqueta debe ser `{% extends %}`. De esta forma se indica la ruta de la plantilla de la que hereda. Si utilizas Twig en Symfony2, indica esta ruta en *notación bundle* (`bundle:carpeta:plantilla`). Si utilizas Twig en un proyecto PHP independiente, indica la ruta de forma relativa respecto al directorio de plantillas.

Una vez añadida la etiqueta `{% extends %}`, esta plantilla ya sólo puede llenar 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 llenar 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 quieras 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 úsalo 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' %}
```

A partir de la versión 1.2 de Twig la etiqueta `{% extends %}` dispone de un funcionamiento todavía más avanzado. Además de indicar el nombre de una plantilla, ahora también puedes indicar una colección de plantillas. Twig utiliza la primera plantilla que exista:

```
{% 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.categoría ~ '.html.twig',
    'sección_' ~ noticia.sección ~ '.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 `sección_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.título }}</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.título }}</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 siguiente etiqueta `{% 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 etiqueta `{% 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 palabras claves `with` y `only` a qué variables pueden acceder las plantillas incluidas. Si no quieres que accedan a ninguna variable, indícalo mediante `only`:

```

{# ... #-}
<h1>Ofertas recientes</h1>
{% include 'listado.html.twig' only %}

```

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.

Impedir el acceso a todas las variables no es algo común, pero si lo es restringir el número de variables a las que puede acceder la plantilla incluida. Para ello, además de `only`, se utiliza la palabra reservada `with` pasándole un array asociativo con las variables que se pasan a la plantilla incluida:

```
{# ... #}
<h1>Ofertas recientes</h1>
{% include 'listado.html.twig' with { 'ofertas': ofertas } only %}
```

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' with datos only %}
```

El uso de `with` no sólo es útil para restringir el acceso a las variables, sino que es imprescindible para que la etiqueta `{% include %}` pueda funcionar en las aplicaciones web reales. 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` espera que la colección se llame `ofertas`, no va a funcionar bien dentro de ninguna plantilla.

La solución consiste en utilizar la palabra reservada `with` para cambiar el nombre de la variable y ajustarlo al que utiliza la plantilla incluida:

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

{% block contenido %}
    <h1>Ofertas destacadas</h1>
    {% include 'listado.html.twig' with { 'ofertas': destacadas } %}
{% endblock %}

{# recientes.html.twig #}
{% extends 'base.html.twig' %}

{% block contenido %}
    <h1>Ofertas recientes</h1>
    {% include 'listado.html.twig' with { 'ofertas': recientes } %}
{% endblock %}

{# ciudad.html.twig #}
<!DOCTYPE html>
<html>
    {# ... #}

    <aside>
        <h3>Ofertas cercanas</h3>
        {% include 'listado.html.twig' with { '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`.

A partir de la versión 1.2 de Twig la etiqueta `{% include %}` también permite controlar su comportamiento cuando la plantilla incluida no existe. En primer lugar, las palabras reservadas `ignore missing` indican que si la plantilla incluida no existe, se ignore completamente la etiqueta `{% include %}`:

```
{% set seccion = ... %}
{% include 'lateral_' ~ seccion ~ '.html.twig' ignore missing %}
```

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 etiqueta `{% include %}` y no se produce ningún error.

Las palabras `ignore missing` deben incluirse justo después del nombre de la plantilla. Esto es muy importante cuando además de incluir la plantilla quieras controlar las variables que se le pasan:

```
{% include 'lateral_' ~ seccion ~ '.html.twig' ignore missing %}
{% include 'lateral_' ~ seccion ~ '.html.twig' ignore missing only %}
{% include 'lateral_' ~ seccion ~ '.html.twig' ignore missing with {
    'cotizaciones': true, 'titulares': false } %}
{% include 'lateral_' ~ seccion ~ '.html.twig' ignore missing with {
    'cotizaciones': true, 'titulares': false } only %}
```

Igualmente, también **a partir de la versión 1.2 de Twig**, 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 las palabras clave `ignore missing`:

```
{% set seccion = ... %}
{% set categoria = ... %}

{% include [
    'lateral_' ~ categoria ~ '.html.twig',
    'lateral_' ~ seccion ~ '.html.twig',
    'lateral.html.twig'
] ignore missing %}
```

A.3.9.2 Herencia adaptable

A partir de su versión 1.8, 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 etiqueta `{% include %}` no se puede utilizar en este caso, ya que sólo incluye los contenidos que le indicas y no puedes modificarlos (no podrías llenar 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 etiqueta `{% 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 etiqueta `{% include %}`, por lo que puedes pasarle variables (`with`), limitar el acceso a las variables de la plantilla principal (`only`) e incluso no mostrar ningún error cuando no exista la plantilla que quieras embeber (`ignore missing`).

A.3.10 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 versiones más recientes de Twig definen siete tipos de extensiones:

- `global`, permiten definir variables globales que están disponibles en todas las plantillas de la aplicación.

- **macro**, se emplean normalmente para generar parte del contenido HTML de la propia plantilla. Conceptualmente son equivalentes a las funciones de los lenguajes de programación.
- **function**, también se emplean para generar contenidos dentro de la plantilla. Twig incluye las siguientes funciones: `attribute`, `block`, `constant`, `cycle`, `date`(también es un filtro), `dump`, `parent`, `random`, `range`.
- **filter**, transforman los contenidos antes de mostrarlos por pantalla. Twig incluye los siguientes filtros: `abs`, `capitalize`, `convert_encoding`, `date`, `date_modify`, `default`, `escape`, `format`, `join`, `json_encode`, `keys`, `length`, `lower`, `merge`, `n12br`, `number_format`, `raw`, `replace`, `reverse`, `slice`, `sort`, `striptags`, `title`, `trim`, `upper`, `url_encode`.
- **tag**, son las etiquetas con las que se construye la lógica de las plantillas. Twig incluye las siguientes etiquetas: `autoescape`, `block`, `do`, `embed`, `extends`, `filter`, `flush`, `for`, `from`, `import`, `include`, `macro`, `raw`, `sandbox`, `set`, `spaceless`, `use` y las correspondientes etiquetas de cierre (`endautoescape`, `endblock`, etc.)
- **test**, algunos parecen funciones y otros parecen etiquetas. Se emplean para evaluar expresiones o el contenido de una variable. Los tests que incluye Twig son: `constant`, `defined`, `divisibleby`, `empty`, `even`, `iterable`, `null`, `odd`, `sameas`.
- **operator**: son los operadores que combinan variables o expresiones para obtener otro valor. Twig incluye los siguientes operadores: `in`, `is`, operadores lógicos (`and`, `&&`, `or`, `|| not`, `(`, `)`), operadores de comparación (`==`, `==>`, `!=`, `<`, `<=`, `>`, `>=`), operadores matemáticos (`+`, `-`, `*`, `/`, `%`, `**`) y otros operadores (`..`, `|`, `~`, `...`, `[`, `]`, `?:`).

Las extensiones más comunes en las aplicaciones web son los macros, las variables globales, las funciones y los filtros. Crear una etiqueta es realmente costoso y difícil, pero prácticamente nunca es necesario hacerlo. Los tests y operadores incluidos en Twig también son más que suficientes para cualquier aplicación web, por lo que tampoco es habitual crear nuevos.

A.3.11 Creando extensiones propias en Symfony2

Antes de explicar detalladamente cada una de las extensiones anteriores, resulta esencial conocer cómo se crean las extensiones propias en Symfony2. Salvo las variables globales y las macros, el resto de extensiones se definen de una manera especial dentro de Symfony2.

Independientemente del tipo o cantidad de extensiones que definas, todas ellas se incluyen en clases PHP denominadas *extensiones propias de Twig*. Estas clases heredan de `Twig_Extension`, se crean en el directorio `Twig/Extension/` del *bundle* (que hay que crear manualmente) y su nombre termina en `Extension`.

Así que si quieras definir por ejemplo una extensión propia llamada `Utilidades` en el *bundle* `OfertaBundle` de tu proyecto de Symfony2, debes crear la siguiente clase:

```
// src/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\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.

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 599) del apéndice B:

```
# app/config/config.yml
services:
    twig.extension.utilidades:
        class: Cupon\OfertaBundle\Twig\Extension\UtilidadesExtension
        tags:
            - { name: twig.extension }
```

A.3.12 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.

Si utilizas Twig en Symfony2, las variables globales se definen en el archivo de configuración global de la aplicación:

```
# app/config/config.yml
twig:
    globals:
        impuestos: 18
        categoria_por_defecto: 'novedades'
```

Si utilizas Twig en un proyecto PHP independiente, debes añadirlas mediante el método `addGlobal()` del objeto que crea el entorno de ejecución de Twig:

```
// ...
$loader = new Twig_Loader_Filesystem(__DIR__.'/plantillas');
$twig = new Twig_Environment($loader);

$twig->addGlobal('impuestos', 18);
$twig->addGlobal('categoria_por_defecto', 'novedades');
$twig->addGlobal('utilidades', new Util());
```

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.3.13 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, id) %}
    <input type="{{ tipo | default('text') }}" name="{{ nombre }}" id="{{ id | default(nombre) }}"
        value="{{ valor }}" {{ requerido ? 'required="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. Para establecer el valor inicial de las variables, se utiliza el filtro `default()`. 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 prefijando su nombre con `_self`. Así que la macro de este ejemplo se puede ejecutar con la instrucción `{{ _self.campo(...) }}`, tal y como muestra el siguiente ejemplo:

```
{% macro campo(nombre, requerido, valor, tipo, id) %}
    <input type="{{ tipo|default('text') }}" name="{{ nombre }}" id="{{ id | default(nombre) }}"
        value="{{ valor }}" {{ requerido ? 'required="required"' : '' }} />
    {% endmacro %}

Nombre: {{ _self.campo('nombre', true, 'José') }}
Apellidos: {{ _self.campo('apellidos', true, 'García Pérez') }}
Teléfono: {{ _self.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="required" />
Apellidos: <input type="text" name="apellidos" id="apellidos" value="García
    Pérez" required="required" />
Teléfono: <input type="text" name="telefono" id="telefono" value="" />
```

Si quieras 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, id) %}
    <input type="{{ tipo|default('text') }}" name="{{ nombre }}" id="{{ id |
```

```
default(nombre)}" value="{{ valor }}" {{ requerido ? 'required="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, id) %}
    <input type="{{ tipo|default('text') }}" name="{{ nombre }}" id="{{ id | default(nombre) }}" value="{{ valor }}" {{ requerido ? 'required="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 %}







{# ... #-}

{% from 'utilidades.html.twig' import item as campo %}



{{ formulario.campo('nombre', true, 'José') }}
{{ formulario.campo('apellidos', true, 'García Pérez') }}
{{ formulario.campo('telefono') }}

```

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 etiqueta `{% include %}`.

A.3.14 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);
```

A.3.14.1 Creando filtros en Symfony2

Los filtros en Symfony2 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/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\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/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            'longitud' => new \Twig_Filter_Method($this, 'longitud'),
        );
    }

    public function getName()
    {
        return 'utilidades';
    }
}
```

El método `getFilters()` devuelve un array asociativo con todos los filtros definidos por la extensión. La clave de cada elemento del array es el nombre del filtro. Este nombre es el que tendrás que escribir en las plantillas para utilizar el filtro y debe ser único en la aplicación.

Cada filtro se declara con la clase `Twig_Filter_Method`. Su primer argumento es el nombre de la clase donde se encuentra el filtro (normalmente `$this`) y el segundo argumento es el nombre del método que implementa el filtro. Para finalizar la creación del filtro, añade un método `longitud()` en la clase de la extensión e incluye en su interior toda la lógica del filtro:

```
// src/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;
```

```

class UtilidadesExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            'longitud' => new \Twig_Filter_Method($this, 'longitud'),
        );
    }

    public function longitud($valor)
    {
        return strlen($valor);
    }

    public function getName()
    {
        return 'utilidades';
    }
}

```

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/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\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.3.14.2 Creando filtros en PHP

Si utilizas Twig en un proyecto PHP independiente, puedes definir la función PHP del filtro dentro del mismo *script* que renderiza las plantillas. Después, añade el nuevo filtro con el método `addFilter()`:

```
// ...
$loader = new Twig_Loader_Filesystem(__DIR__.'/plantillas');
$twig = new Twig_Environment($loader);

function longitud($valor) {
    return strlen($valor);
}

$twig->addFilter('longitud', new Twig_Filter_Function('longitud'));
```

Ahora ya puedes utilizar el nuevo filtro en cualquier plantilla:

```
 {{ variable | longitud }}
```

Para no *ensuciar* el *script* que renderiza plantillas, es mejor definir todos los filtros propios como métodos estáticos en una clase auxiliar. Si por ejemplo esa clase se llama `Utilidades`, el nuevo filtro se definiría así:

```
// clase Utilidades.php
class Utilidades {
    public static function longitud($valor) {
        return strlen($valor);
    }
}

// script que renderiza plantillas
$loader = new Twig_Loader_Filesystem(__DIR__.'/plantillas');
$twig = new Twig_Environment($loader);

$twig->addFilter('longitud',new Twig_Filter_Function('Utilidades::longitud'));
```

A.3.14.3 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 Symfony2
public function getFilters()
{
    return array(
        'mi_filtro' => new \Twig_Filter_Method($this, 'miFiltro', array(
            'is_safe' => array('html')
        )),
    );
}
```

```

    );
}

// En proyectos PHP independientes
$twig->addFilter(
    'mi_filtro',
    new Twig_Filter_Function('MisExtensiones::mi_filtro', array(
        'is_safe' => array('html')
    )
);

```

Por otra parte, si quieras que Twig aplique el mecanismo de escape al valor que pasa al filtro, añade la opción `pre_escape`:

```

// En Symfony2
public function getFilters()
{
    return array(
        'mi_filtro' => new \Twig_Filter_Method($this, 'miFiltro', array(
            'pre_escape' => array('html')
        )),
    );
}

// En proyectos PHP independientes
$twig->addFilter(
    'mi_filtro',
    new Twig_Filter_Function('MisExtensiones::mi_filtro', array(
        'pre_escape' => array('html')
    )
);

```

A.3.14.4 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 Symfony2
public function getFilters()
{
    return array(
        'longitud' => new \Twig_Filter_Method($this, 'longitud', array(
            'needs_environment' => true
        )),
    );
}

// En proyectos PHP independientes
$twig->addFilter(
    'longitud',
    new Twig_Filter_Function('Utilidades::longitud', array(
        'needs_environment' => true
    )
);

```

Después, modifica el código del filtro, ya que ahora Symfony2 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.3.14.5 Filtros dinámicos

A partir de su versión 1.5, Twig también permite definir filtros dinámicos. Observa el siguiente ejemplo en el que una plantilla utiliza tres filtros diferentes para indicar cómo se muestra un determinado contenido:

```
{{ contenido | mostrar_como_lista }}
{{ contenido | mostrar_como_lista('ol') }}
{{ contenido | mostrar_como_tabla }}
{{ contenido | mostrar_como_titular }}
{{ contenido | mostrar_como_titular('h2') }}
```

Utilizando los filtros normales de Twig, deberías definir tres filtros diferentes. Haciendo uso de los filtros dinámicos, puedes definir un único filtro llamado `mostrar_como_*` (con el asterisco al final). Cuando el nombre de un filtro contiene un asterisco, Twig entiende que esa parte es variable y puede contener cualquier cadena de texto.

Para crear un filtro dinámico, primero añade un `*` en su nombre al registrar el nuevo filtro. Después, en la función que procesa el filtro ten en cuenta que el primer argumento que le pasa Twig es

precisamente el valor que tiene la parte variable del nombre del filtro. El siguiente ejemplo muestra el código necesario para procesar el filtro dinámico del ejemplo anterior:

```
$twig->addFilter('mostrar_como_*', new Twig_Filter_Function('mostrar'));

function mostrar($tipo, $opciones)
{
    switch ($tipo) {
        case 'lista':
            // ...
            break;

        case 'tabla':
            // ...
            break;

        case 'titular':
            // ...
            break;
    }
}
```

Los filtros dinámicos pueden tener más de una parte variable, por lo que el ejemplo anterior se podría haber resuelto también de la siguiente manera:

```
{{ contenido | mostrar_como_lista_ul }}
{{ contenido | mostrar_como_lista_ol }}
{{ contenido | mostrar_como_tabla_normal }}
{{ contenido | mostrar_como_titular_h1 }}
{{ contenido | mostrar_como_titular_h2 }}
```

El nombre del filtro en este caso sería `mostrar_como_*_*`.

A.3.15 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) }}
```

A.3.15.1 Creando funciones en Symfony2

Definir una función de Twig en Symfony2 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_Function_Method` en vez de `Twig_Filter_Method`. El siguiente código muestra cómo definir una función llamada `mi_funcion()`:

```
// src/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\Twig\Extension;

class UtilidadesExtension extends \Twig_Extension
{
    public function getFunctions()
    {
        return array(
            'mi_funcion' => new \Twig_Function_Method($this, 'miFuncion'),
        );
    }

    public function miFuncion()
    {
        // ...

        return $respuesta;
    }

    public function getName()
    {
        return 'utilidades';
    }
}
```

Si la función admite parámetros, Symfony2 los pasa automáticamente al método de la función en el mismo orden en el que se escriben en la plantilla:

```
// src/Cupon/OfertaBundle/Twig/Extension/UtilidadesExtension.php
namespace Cupon\OfertaBundle\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.3.15.2 Creando funciones en PHP

De manera similar a los filtros, si utilizas Twig en un proyecto PHP independiente, debes declarar las funciones con el método `addFunction()`:

```
// ...
$loader = new Twig_Loader_Filesystem(__DIR__ . '/plantillas');
$twig = new Twig_Environment($loader);

function mi_funcion() {
    // ...

    return $respuesta;
}

$twig->addFunction('mi_funcion', new Twig_Function_Function('mi_funcion'));
```

A.3.15.3 Funciones dinámicas

A partir de su versión 1.5, Twig también permite definir funciones dinámicas, un concepto idéntico al de los filtros dinámicos explicados anteriormente. Estas funciones son muy útiles para casos como el siguiente:

```
 {{ the_id() }}
{{ the_title() }}
{{ the_time() }}
{{ the_content() }}
{{ the_category() }}
{{ the_shortlink() }}
```

El código anterior muestra algunas de las funciones que utiliza el sistema de blogs WordPress en sus plantillas para mostrar las propiedades de una página o un artículo. Gracias a las funciones dinámicas de Twig puedes definir una única función llamada `the_*`() que se encargue de procesar todas ellas.

Para crear una función dinámica, primero añade un `*` en su nombre al registrar la nueva función. Después, en la función que procesa la función dinámica ten en cuenta que el primer argumento que le pasa Twig es precisamente el valor que tiene la parte variable del nombre de la función. El siguiente ejemplo muestra el código necesario para procesar el filtro dinámico del ejemplo anterior:

```
$twig->addFunction('the_*', new Twig_Function_Function('wordpress'));

function wordpress($propiedad, $opciones)
{
    switch ($propiedad) {
        // ...
```

```

    }
}

```

Si la plantilla pasa argumentos a las funciones dinámicas, Twig los incluye después del primer argumento:

```

{# se ejecuta: wordpress('id') #}
{{ the_id() }}

{# se ejecuta: wordpress('title', '<h3>', '</h3>') #}
{{ the_title('<h3>', '</h3>') }}

```

A.4 Usando Twig en proyectos PHP propios

Twig es un proyecto mantenido por los mismos creadores de Symfony2. Por eso los dos se integran perfectamente y puedes utilizar Twig en tus aplicaciones Symfony2 sin esfuerzo.

No obstante, también resulta muy sencillo utilizar Twig en tus propios proyectos PHP. En los siguientes ejemplos se supone que la estructura de directorios de la aplicación es la siguiente:

```

proyecto/
├── pagina.php
├── cache/
├── plantillas/
│   └── plantilla.twig
└── vendor/
    └── twig/

```

A.4.1 Instalación

Para instalar la librería de Twig, descarga o clona con Git su código desde la dirección <https://github.com/fabpot/Twig> y guárdalo en el directorio [vendor/twig/](#) del proyecto.

También puedes instalar Twig mediante PEAR ejecutando los siguientes comandos:

```
$ pear channel-discover pear.twig-project.org
$ pear install twig/Twig
```

Por último, Twig también se puede instalar mediante Composer. De hecho, esta es la forma recomendada de hacerlo, ya que Composer gestiona las dependencias de los proyectos mucho mejor que PEAR o que las instalaciones realizadas a mano. Añade en primer lugar la siguiente dependencia en el archivo [composer.json](#) de tu proyecto:

```

{
    "require": {
        "twig/twig": "1.*"
    }
}

```

Después, ejecuta el comando habitual para actualizar las dependencias de los proyectos:

```
$ composer update
```

NOTA Para que te funcione este comando, debes instalar Composer globalmente, tal y como se explica en la siguiente guía: <http://symfony.es/documentacion/guia-de-instalacion-de-composer/>

A.4.2 Configuración

Antes de poder utilizar Twig, debes registrar su *autoloader* o cargador de clases en todos los *scripts* PHP en los que vayas a utilizar plantillas Twig.

Si has instalado Twig a mano, clonando el repositorio Git o mediante PEAR, añade estas dos líneas al principio del *script* `página.php`:

```
<?php  
require_once __DIR___. '/vendor/twig/lib/Twig/Autoloader.php';  
Twig_Autoloader::register();  
  
// ...
```

Asegúrate de indicar la ruta correcta hasta el archivo `Autoloader.php` según la estructura de directorios de tu proyecto. Si la ruta no es correcta, no se cargará ninguna clase de Twig y por tanto, no funcionará ninguna plantilla.

Si has instalado Twig mediante Composer, reemplaza el código anterior por lo siguiente:

```
<?php  
require_once __DIR___. '/vendor/autoload.php';  
  
// ...
```

Al margen del *autoloader*, los dos elementos más importantes para el correcto funcionamiento de Twig son el cargador de plantillas y el entorno de ejecución.

Las plantillas de Twig normalmente se crean en archivos, por lo que se utiliza el cargador llamado `Twig_Loader_Filesystem`. Si creas las plantillas dinámicamente, debes utilizar el cargador `Twig_Loader_String` para pasar directamente el contenido de la plantilla Twig en una cadena de texto.

El cargador de plantillas `Twig_Loader_Filesystem` espera como argumento la ruta hasta el directorio donde se encuentran las plantillas. Con la estructura de directorios mostrada anteriormente, esta ruta es `__DIR__. '/plantillas'`:

```
<?php  
require_once __DIR___. '/vendor/twig/lib/Twig/Autoloader.php';  
Twig_Autoloader::register();  
  
$loader = new Twig_Loader_Filesystem(__DIR__. '/plantillas');
```

Después, configura el entorno de ejecución de Twig con la clase `Twig_Environment`, que espera como primer argumento un cargador de plantillas:

```
<?php
require_once __DIR___. '/vendor/twig/lib/Twig/Autoloader.php';
Twig_Autoloader::register();

$loader = new Twig_Loader_Filesystem(__DIR___. '/plantillas');
$twig = new Twig_Environment($loader);
```

El constructor de la clase `Twig_Environment` también permite indicar como segundo parámetro un array con las siguientes opciones de configuración:

- `autoescape`: indica si se aplica automáticamente el mecanismo de escape en todas las plantillas. Su valor por defecto es `true`.
- `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`.
- `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. Su valor por defecto es `false`, lo que significa que Twig no utiliza ninguna caché.
- `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. Su valor por defecto es `false`.
- `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`.

Ejemplo de configuración del entorno de ejecución de Twig:

```
$twig = new Twig_Environment($loader, array(
    'debug'    => true,
    'strict_variables' => true,
    'charset'  => 'UTF-8',
```

```
'cache' => __DIR__ . '/cache'
));
```

A.4.3 Renderizando plantillas

Twig ya está correctamente cargado y configurado. Ahora, para renderizar una plantilla, cárgala con el método `loadTemplate()` y renderízala con el método `render()`:

```
<?php
require_once __DIR__ . '/vendor/twig/lib/Twig/Autoloader.php';
Twig_Autoloader::register();

$loader = new Twig_Loader_Filesystem(__DIR__ . '/plantillas');
$twig = new Twig_Environment($loader);

$plantilla = $twig->loadTemplate('plantilla.twig');
$codigoHtml = $plantilla->render(array(
    'variable1' => 'valor1',
    'variable2' => 'valor2'
));
echo $codigoHtml;
```

El método `loadTemplate()` espera como argumento la ruta de la plantilla con respecto al directorio de plantillas configurado previamente. Como en este caso la plantilla se encuentra en ese mismo directorio, solamente hay que indicar el nombre de la plantilla.

Una vez cargada la plantilla, genera las páginas con el método `render()`. A este método se le pasa como argumento un array con las variables que estarán disponibles en la plantilla. El método `render()` devuelve directamente la plantilla renderizada (normalmente código HTML). Para mostrarlo por pantalla, guárdalo en una variable o utiliza una instrucción `echo` de PHP.

Las plantillas de Twig también disponen de un método llamado `display()` que internamente funciona como `render()` pero no devuelve el contenido de la página sino que lo muestra directamente, por lo que no hace falta añadir ningún `echo`:

```
$plantilla = $twig->loadTemplate('plantilla.twig');
$plantilla->display(array(
    'variable1' => 'valor1',
    'variable2' => 'valor2'
));
```

Por último, Twig también permite cargar y renderizar la plantilla con una única instrucción. Para ello, utiliza el método `render()` de Twig, no de la plantilla. Indica como primer argumento la ruta de la plantilla y como segundo argumento el array con las variables de la plantilla:

```
$codigoHtml = $twig->render('plantilla.twig', array(
    'variable1' => 'valor1',
    'variable2' => 'valor2'
));
```

A.5 Usando Twig en Symfony2

Utilizar Twig dentro de una aplicación Symfony2 es todavía mejor y más fácil que usarlo de forma independiente: las plantillas se cargan automáticamente con la *notación bundle*, 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`. Symfony2 define las opciones comunes en el archivo `app/config/config.yml` y después añade alguna opción particular para los archivos `app/config/config_dev.yml` y `app/config/config_prod.yml`.

A continuación se muestran las opciones disponibles y sus valores por defecto:

```
twig:
    autoescape: ~
    base_template_class: ~
    cache: %kernel.cache_dir%/twig
    charset: %kernel.charset%
    debug: %kernel.debug%
    strict_variables: ~
    auto_reload: ~
```

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) %}
```

Sin embargo, el código anterior no funciona por defecto en Symfony2, ya que la función `dump()` no está disponible. Esta función pertenece a la extensión `debug` que Symfony2 incluye pero no activa por defecto. Así que añade la configuración correspondiente para activar esta extensión, tal como se explica en la sección *Definiendo servicios especiales* (página 599) del apéndice B:

```
# app/config/config.yml
services:
    twig.extension.debug:
        class: Twig_Extensions_Extension_Debug
        tags:
            - { name: twig.extension }
```

A.5.3 Variables globales

Symfony2 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.security`, devuelve un objeto con toda la información del contexto de seguridad. Se obtiene lo mismo que ejecutando `$this->get('security.context')` en un controlador. En la práctica suele utilizarse más la siguiente opción `app.user`.
- `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 Doctrine2, 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.

Todas estas variables globales se definen en la clase `Symfony\Bundle\FrameworkBundle\Templating\GlobalVariables.php`

A.5.4 Servicios como variables globales

Además de valores simples, las variables globales de Twig también pueden guardar servicios de Symfony2. Imagina que dispones de una clase llamada `Util` para agrupar los métodos y funciones útiles que se emplean en toda la aplicación. Si la clase se encuentra en un *bundle* llamado `OfertaBundle` e incluye un método `getSlug()`, su código será similar al siguiente:

```
namespace Cupon\OfertaBundle\Util;

class Util
{
    static public function getSlug($cadena)
    {
        // ...

        return $slug;
    }
}
```

La clave para disponer de la clase `Util` y sus métodos en cualquier plantilla de Twig es convertir la clase en un servicio. Para ello, abre el archivo de configuración `app/config/config.yml` y define un nuevo servicio bajo la clave `services`:

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

```
# ...
cupon.utilidades:
    class: Cupon\OfertaBundle\Util\Util
```

A continuación, define una variable global de Twig utilizando este nuevo servicio:

```
# app/config/config.yml
twig:
    globals:
        utilidades: @cupon.utilidades
```

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 `utilidades` para acceder a los métodos de la clase `Util`:

```
{{ utilidades.getSlug('Lorem ipsum dolor sit amet') }}
#{ el resultado es: lorem-ipsum-dolor-sit-amet #}
```

A.5.5 Internacionalización

Cuando se utiliza dentro de Symfony2, 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
eiusmod tempor incididunt ut [...] laboris nisi ut aliquip ex ea commodo
consequat.{% 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 283) dedicado a internacionalizar el sitio web explica en detalle todas las opciones del componente de traducción de Symfony2 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 enruteamiento depende de la aplicación en la que se utilice Twig. Symfony2 sí 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 Symfony2 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 Symfony2.

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="form_url" name="form[url]" required="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 Symfony2.

De esta forma, el ejemplo anterior hace que todos los campos de tipo URL muestren el texto `http://` 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:

```
{# src/Cupon/OfertaBundle/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 'OfertaBundle:Form:form.html.twig' %}

{# ... #}

{{ form_row(url) }}
```

Por último, si quieras 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.

Symfony2 define una opción de Twig llamada `form` 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:
    resources:
      - 'form_div_layout.html.twig'
```

Además de `form_div_layout.html.twig` Symfony2 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:
    resources:
      - '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 %}
```

```
{% 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 `resources`:

```
# app/config/config.yml
twig:
    # ...
    form:
        resources:
            - 'MiBundle: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 Symfony2. Junto al *contenedor de servicios* (del inglés, *service container*) es lo que hace que Symfony2 sea tan rápido y 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 Symfony2, 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 Symfony2 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 Symfony2 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 Symfony2 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. Symfony2 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 Symfony2

Hacer uso de la inyección de dependencias en las aplicaciones Symfony2 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 Symfony2 se encarga de procesar esa configuración y prepara todas las dependencias entre clases.

Si quieras 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 Symfony2. También puedes instalar el componente mediante PEAR (más información: <http://pear.symfony.com/>). 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 Symfony2, 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 Symfony2, lo habitual no es definir servicios en el contenedor de inyección de dependencias, sino utilizar los servicios que incluye Symfony2 y los que definen los *bundles* desarrollados por terceros. De hecho, a menos que la aplicación sea muy compleja, es posible desarrollar una aplicación Symfony2 completa sin crear ningún servicio propio.

Observa el siguiente código de un controlador imaginario de Symfony2:

```
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 Symfony2 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 Symfony2 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 Symfony2 busca entre todos los servicios definidos aquel que se llame `request` y realiza todas las tareas necesarias para instanciar el objeto correspondiente y entregarlo.

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 quieras 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 Symfony2) lo primero que debes hacer es configurarlos adecuadamente.

Las opciones de configuración disponibles y sus posibles valores se indican en la documentación de Symfony2 o del *bundle* que estés utilizando. La mayoría de opciones de Symfony2 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 Symfony2:

```
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 quieras 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 quieras 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 Symfony2. Las consultas a la base de datos se realizan mediante el *entity manager* de Doctrine2:

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

El método `getDoctrine()` es un atajo disponible en todos los controladores que extienden de la clase `Controller` de Symfony2. 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 Symfony2:

```
$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 Symfony2 son controladores, entidades de Doctrine2, 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 `Util` para definir los métodos útiles que se usan en toda la aplicación. Esta clase se encuentra en el archivo `src/Cupon/OfertaBundle/Util/Util.php` y su código simplificado es el que se muestra a continuación:

```
// src/Cupon/OfertaBundle/Util/Util.php
namespace Cupon\OfertaBundle\Util;

class Util
{
    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:
    cupon.utilidades:
        class: Cupon\OfertaBundle\Util\Util
```

Ya está! Ya has creado un servicio listo para el contenedor de inyección de dependencias de Symfony2. Para ello sólo hay que asignar al servicio un nombre único (en este caso, `cupon.utilidades`) 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 Symfony2:

```
class DefaultController extends Controller
{
    public function portadaAction()
    {
        // Acceder al servicio
        $utilidades = $this->get('cupon.utilidades');

        // Utilizar directamente un método del servicio
        $slug = $this->get('cupon.utilidades')->getSlug($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:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        arguments: ['utf-8']
```

Cuando hagas uso del servicio `cupon.utilidades`, 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/Cupon/OfertaBundle/Util/Util.php
namespace Cupon\OfertaBundle\Util;

class Util
{
```

```

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:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        arguments: ['utf-8', true, 10, ..., null]

```

Cuando quieras 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 Symfony2, encierra su nombre entre %:

```

# app/config/config.yml
parameters:
    codificacion: 'utf-8'

services:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        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 `cupon.utilidades.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 Symfony2 define varios parámetros que también puedes utilizar en tu código encerrando su nombre entre %:

Parámetro	Valor
<code>kernel.environment</code>	Nombre del entorno en el que se está ejecutando la aplicación (<code>dev</code> , <code>prod</code> , etc.)
<code>kernel.cache_dir</code>	Ruta del directorio de la caché de la aplicación (normalmente <code>app/cache/%kernel.environment%</code>)
<code>kernel.logs_dir</code>	Ruta del directorio de logs de la aplicación (normalmente <code>app/logs</code>)
<code>kernel.debug</code>	Indica si la aplicación se está ejecutando con la depuración activada (por defecto vale <code>false</code> en el entorno <code>prod</code> y <code>true</code> en el entorno <code>dev</code>)
<code>kernel.name</code>	Nombre del kernel de la aplicación (normalmente <code>app</code>)
<code>kernel.bundle</code>	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 Symfony2 también soporta la inyección de parámetros mediante métodos *setter*:

```
# app/config/config.yml
parameters:
    cupon.codificacion: 'utf-8'

services:
    cupon.utilidades:
        class: Cupon\OfertaBundle\Util\Util
        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 Symfony2 que al crear el servicio `cupon.utilidades` 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 `Util` para añadir el *setter* que va a invocar el contenedor:

```
// src/Cupon/OfertaBundle/Util/Util.php
namespace Cupon\OfertaBundle\Util;

class Util
{
    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 `cupon.utilidades` 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 `cupon.utilidades` también añade información de depuración en los archivos de log, debería configurar otra dependencia con el servicio `logger` de Symfony2.

Para definir las dependencias con otros servicios, sólo tienes que *inyectarlos* mediante la opción `arguments` del servicio:

```
# app/config/config.yml  
services:  
    cupon.utilidades:  
        class:      Cupon\OfertaBundle\Util\Util  
        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 `cupon.utilidades`, el contenedor creará primero los servicios `logger` y `router` para pasárselos al constructor 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/Cupon/OfertaBundle/Util/Util.php  
class Util  
{  
    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 Symfony2 más utilizados:

- `doctrine`, servicio definido por Doctrine2, 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.
- `form.factory`, servicio del componente de formularios, a través del cual se crean todos los formularios de Symfony2.
- `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 Symfony2 y que permite acceder a toda la información sobre la ejecución y rendimiento de cada petición.
- `request`, servicio mediante el que se obtiene el objeto de tipo `Request` que contiene toda la información sobre la petición del usuario.
- `response`, servicio mediante el que se obtiene el objeto de tipo `Response` que representa a la respuesta que se envía al usuario.
- `security.context`, servicio definido por el componente de seguridad. Normalmente se utiliza para obtener el objeto que representa al usuario actualmente logueado.
- `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 Symfony2 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 Symfony2, 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:
    cupon.utilidades:
        # ...

```

Otra solución recomendada es incluir la configuración de los servicios de cada *bundle* en su propio *bundle*. Así por ejemplo, el servicio `cupon.utilidades` se definiría en el *bundle* `OfertaBundle`. Si te fijas en el contenido de cualquier *bundle*, verás que Symfony2 crea un archivo llamado `services.yml` dentro del directorio `Resources/config/`. Por defecto su contenido es el siguiente:

```

# src/Cupon/OfertaBundle/Resources/config/services.yml

parameters:
#     oferta.example.class: Cupon\OfertaBundle\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 Symfony2 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 Symfony2.

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 Symfony2.

Imagina que has creado una extensión llamada `CuponExtension` dentro del *bundle OfertaBundle*:

```
// src/Cupon/OfertaBundle/Twig/Extension/CuponExtension.php
namespace Cupon\OfertaBundle\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: Cupon\OfertaBundle\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: Cupon\OfertaBundle\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

Symfony2 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/Cupon/OfertaBundle/Listener/RequestListener.php
namespace Cupon\OfertaBundle\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: Cupon\OfertaBundle\Listener\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request }
```

Si al procesar la petición del usuario se notifica el evento `kernel.request`, Symfony2 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: Cupon\OfertaBundle\Listener\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request,
                method: setNuevoFormatoPeticion }
```

Otros de los eventos más populares en Symfony2 son los eventos definidos por Doctrine2, 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: Cupon\OfertaBundle\Listener\Normalizar  
        tags:  
            - { name: doctrine.event_listener, event: prePersist }
```

Doctrine2 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 Doctrine2. 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 Symfony2 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

Symfony2 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 Symfony2 añade también varios atajos para obtener los servicios más utilizados en los controladores:

- Petición:
 - Atajo: `$this->getRequest()`
 - Código equivalente: `$this->container->get('request')`
- Doctrine2:
 - Atajo: `$this->getDoctrine()`
 - Código equivalente: `$this->container->get('doctrine')`
- Formularios:
 - Atajo: `$this->createForm(...)`
 - Código equivalente: `$this->container->get('form.factory')->createForm(...)`
- 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 Symfony2:

```
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 `Util` para definir los métodos útiles que se usan en toda la aplicación. Esta clase se encuentra en el archivo `src/Cupon/OfertaBundle/Util/Util.php` y su código simplificado es el que se muestra a continuación:

```
// src/Cupon/OfertaBundle/Util/Util.php
namespace Cupon\OfertaBundle\Util;

class Util
{
    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:
    cupon.utilidades:
        class:     Cupon\OfertaBundle\Util\Util
```

A continuación, inyéctale como argumento el propio contenedor de servicios, disponible a través de un servicio llamado `service_container`:

```
# app/config/config.yml
services:
    cupon.utilidades:
```

```
class:      Cupon\OfertaBundle\Util\Util
arguments:  [@service_container]
```

Por último, añade un constructor en la clase `Util` para recoger el objeto del contenedor que le pasa Symfony2:

```
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 Symfony2 dispone de muchas otras opciones y características avanzadas, como las que se enumeran a continuación.

B.2.5.1 Depurando el contenedor

Symfony2 incluye un comando llamado `container:debug` 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 container:debug

[container] Public services
Name          Scope     Class Name
-----
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 información que proporciona el comando `container:debug` 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 quieras 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 Symfony2 que incluye una representación gráfica de las relaciones entre los diferentes servicios de la aplicación.

B.2.5.2 Servicios privados

El comando anterior solamente muestra los servicios públicos definidos en el contenedor. Por defecto todos los servicios de Symfony2 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:

```
# app/config/config.yml
services:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        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('cupon.utilidades')` dejen de funcionar y muestren un error de tipo *"You have requested a non-existent service cupon.utilidades"*.

El comando `container:debug` también permite ver los servicios privados de la aplicación. Para ello, añade la opción `--show-private`:

```
$ php app/console container:debug --show-private
[container] Public and private services
Name          Scope  Class Name
-----
353a5...f9e7ee1f4_1  container  [...]\Controller\ControllerResolver
353a5...f9e7ec01_10  container  [...]\DependencyInjection\Extension
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 Symfony2.

B.2.5.3 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 Symfony2:

```
# app/config/config.yml
services:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        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:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        arguments:  [@?logger]
```

Si el servicio `logger` existe en la aplicación, el contenedor lo inyectará normalmente y tu servicio podrá hacer uso de él. 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 Util
{
    private $logger;

    public function __construct($logger)
    {
        $this->logger = $logger;
    }

    // ...
}

// Ahora
```

```

class Util
{
    private $logger;

    public function __construct($logger = null)
    {
        $this->logger = $logger;
    }

    // ...
}

```

B.2.5.4 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`, `cupon.ofertabundle.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:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        arguments:  [@service_container]

    util:
        alias: cupon.utilidades

```

Con la configuración anterior, ya puedes acceder a la clase de utilidades mediante el código `$this->get('util')`.

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.2.5.5 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 Symfony2 es la definición de una extensión propia de Twig:

```
# app/config/config.yml
services:
    twig.extension.cupon:
        class: Cupon\OfertaBundle\Twig\Extension\CuponExtension
        arguments: [@translator]
        tags:
            - { name: twig.extension }
```

La etiqueta `twig.extension` indica a Symfony2 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 Symfony2 o por algún *bundle* desarrollado por terceros.

Las etiquetas disponibles en Symfony2 son las siguientes:

- `assetic.filter`
- `assetic.templates.php`
- `data_collector`
- `form.field_factory.guesser`
- `kernel.cache_warmer`
- `kernel.event_listener`
- `monolog.logger`
- `routing.loader`
- `security.listener.factory`
- `security.voter`
- `templating.helper`
- `twig.extension`
- `translation.loader`
- `validator.constraint_validator`

B.2.5.6 Scopes

El comando `container:debug` no sólo permite obtener el listado completo de servicios de la aplicación, sino que también muestra todos los detalles sobre cada servicio. Para ello simplemente debes indicar el nombre del servicio sobre el que quieras conocer sus detalles:

```
$ php app/console container:debug twig.extension.cupon

[container] Information for service twig.extension.cupon

Service Id    twig.extension.cupon
Class         Cupon\OfertaBundle\Twig\Extension\CuponExtension
Tags          twig.extension
Scope         container
Public        yes
```

Observa cómo en la información sobre el servicio se muestra el valor de una opción llamada `scope`. El `scope` o ámbito de un servicio indica durante cuánto tiempo debe el contenedor mantener la misma instancia del objeto. Esta opción admite uno de los tres siguientes valores:

- `container`, indica que el contenedor crea una sola instancia del objeto. Esta única instancia es la que se devuelve siempre que se solicita el servicio. Este es el valor que aplica por defecto Symfony2 a todos los servicios.
- `prototype`, obliga al contenedor a crear una nueva instancia del objeto cada vez que se solicite el servicio.
- `request`, se crea una instancia para cada petición y para cada sub-petición interna creada por Symfony2. Estos servicios no se pueden utilizar fuera de las peticiones HTTP, por lo que no está disponible por ejemplo en la consola de comandos.

Para modificar el `scope` de un servicio, añade la opción `scope` en su definición:

```
# app/config/config.yml
services:
    cupon.utilidades:
        class:      Cupon\OfertaBundle\Util\Util
        arguments:  [@service_container]
        scope:      prototype
```

A menos que desarrolles una aplicación muy compleja, es improbable que tengas que utilizar la opción `scope`. No obstante, al utilizar servicios desarrollados por terceros es posible que Symfony2 muestre un mensaje de error haciendo referencia a los `scopes` de los servicios.

El motivo es que un servicio no puede depender de servicios que tengan un `scope` más limitado (`request` y `prototype` son más limitados que `container`). La solución es tan sencilla como modificar la opción `scope` de tu servicio para igualar su nivel al de los otros servicios. Normalmente basta con establecer el `scope` al nivel `request`.

B.2.5.7 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 Symfony2 como WordPress, Drupal o Magento.

Para incluir un archivo PHP, indica su ruta en la opción `file` del servicio. Symfony2 hará un `require_once()` del archivo antes de crear el servicio:

```
# app/config/config.yml
services:
    cupon.utilidades:
        class: Cupon\OfertaBundle\Util\Util
        file: %kernel.root_dir%../../src/Cupon/OfertaBundle/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 Symfony2 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 Symfony2.

Entre todas esas personas anónimas, merecen ser destacadas las siguientes por su gran contribución a la documentación oficial de Symfony2:

- Fabien Potencier (<http://github.com/fabpot>)
- Ryan Weaver (<http://github.com/weaverryan>)
- 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 Baixaulli (<http://github.com/jbaixaulli>)

- 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 Martín (<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/raulgmm>)
- 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 Fuentes (<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/ggamm>)
- Jose Carlos Liebana (<http://github.com/Greibit>)

Envía tus comentarios, sugerencias y correcciones a **javier.eguiluz@gmail.com** para aparecer en la lista de la próxima edición.