Manual de Symfony2

Release 2.0.1

Traducido por Nacho Pacheco

August 29, 2011

Índice general

Parte I

Guía de inicio rápido

Empieza a trabajar rápidamente con la *Guía de inicio rápido* (Página 5) de *Symfony*:

CAPÍTULO 1

Guía de inicio rápido

1.1 Un primer vistazo

¡Empieza a usar *Symfony2* en 10 minutos! Este capítulo te guiará a través de algunos de los conceptos más importantes detrás de *Symfony2* y explica cómo puedes empezar a trabajar rápidamente, mostrándote un simple proyecto en acción.

Si ya has usado una plataforma para desarrollo web, seguramente te sentirás a gusto con *Symfony2*. Si no es tu caso, ¡bienvenido a una nueva forma de desarrollar aplicaciones web!

Truco: ¿Quieres saber por qué y cuándo es necesario utilizar una plataforma? Lee el documento "Symfony en 5 minutos".

1.1.1 Descargando *Symfony2*

En primer lugar, comprueba que tienes instalado y configurado un servidor web (como Apache) con *PHP 5.3.2* o superior.

¿Listo? Empecemos descargando la "Edición estándar de Symfony2", una *distribución* de *Symfony* preconfigurada para la mayoría de los casos y que también contiene algún código de ejemplo que demuestra cómo utilizar *Symfony2* (consigue el paquete que incluye *proveedores* para empezar aún más rápido).

Después de extraer el paquete bajo el directorio raíz del servidor web, deberías tener un directorio `Symfony/` con una estructura como esta:

```
www/ <- el directorio raíz de tu servidor web
    Symfony/ <- el archivo desempacado
        app/
            cache/
            config/
            logs/
            Resources/
        bin/
        src/
            Acme/
                DemoBundle/
                    Controller/
                    Resources/
```

```
            ...
vendor/
    symfony/
    doctrine/
    ...
web/
    app.php
    ...
```

Nota: Si descargaste la *Edición estándar* `sin vendors`, basta con ejecutar la siguiente orden para descargar todas las bibliotecas de proveedores:

```
php bin/vendors install
```

1.1.2 Verificando tu configuración

Symfony2 integra una interfaz visual para probar la configuración del servidor, muy útil para solucionar problemas relacionados con el servidor Web o una incorrecta configuración de *PHP*. Usa la siguiente url para examinar el diagnóstico:

```
http://localhost/Symfony/web/config.php
```

Si se listan errores o aspectos de configuración pendientes, corrígelos; Puedes realizar los ajustes siguiendo las recomendaciones. Cuando todo esté bien, haz clic en "*Pospón la configuración y llévame a la página de bienvenida*" para solicitar tu primera página web "real" en *Symfony2*:

```
http://localhost/Symfony/web/app_dev.php/
```

¡*Symfony2* debería felicitarte por tu arduo trabajo hasta el momento!

1.1.3 Comprendiendo los fundamentos

Uno de los principales objetivos de una plataforma es garantizar la separación de responsabilidades. Esto mantiene tu código organizado y permite a tu aplicación evolucionar fácilmente en el tiempo, evitando mezclar llamadas a la base de datos, etiquetas *HTML* y el código de lógica del negocio en un mismo archivo. Para alcanzar este objetivo, debes aprender algunos conceptos y términos fundamentales.

Truco: ¿Quieres más pruebas de que usar una plataforma es mucho mejor que mezclar todo en un mismo archivo? Lee el capítulo del libro "*Symfony2 frente a PHP simple* (Página 41)".

La distribución viene con algún código de ejemplo que puedes utilizar para aprender más sobre los principales conceptos de *Symfony2*. Ingresa a la siguiente *URL* para recibir un saludo de *Symfony2* (reemplaza *Nacho* con tu nombre):

```
http://localhost/Symfony/web/app_dev.php/demo/hola/Nacho
```

¿Qué sucedió? Bien, diseccionemos la url:

- `app_dev.php`: Es un *controlador frontal*. Es el único punto de entrada de la aplicación, mismo que responde a todas las peticiones del usuario;
- `/demo/hola/Nacho`: Esta es la *ruta virtual* a los recursos que el usuario quiere acceder.

Tu responsabilidad como desarrollador es escribir el código que asigna la *petición* del usuario (`/demo/hola/Nacho`) al *recurso* asociado con ella (la página *HTML* `¡Hola Nacho!`).

Enrutando

Symfony2 encamina la petición al código que la maneja tratando de hacer coincidir la *URL* solicitada contra algunos patrones configurados. De forma predeterminada, estos patrones (llamados rutas) se definen en el archivo de configuración `app/config/routing.yml`: Cuando estás en el *entorno* (Página 11) `dev` - indicado por el controlador frontal **app_dev.php** - el archivo de configuración `app/config/routing_dev.yml` también es cargado. En la edición estándar, las rutas a estas páginas de "demostración" se encuentran en ese archivo:

```
# app/config/routing_dev.yml
_bienvenida:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Bienvenida:index }

_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo

# ...
```

Las primeras tres líneas (después del comentario) definen el código que se ejecuta cuando el usuario solicita el recurso "/" (es decir, la página de bienvenida que viste anteriormente). Cuando así lo solicite, el controlador `AcmeDemoBundle:Bienvenida:index` será ejecutado. En la siguiente sección, aprenderás exactamente lo que eso significa.

Truco: La *Edición estándar de Symfony2* utiliza YAML para sus archivos de configuración, pero *Symfony2* también es compatible con *XML*, *PHP* y anotaciones nativas. Los diferentes formatos son compatibles y se pueden utilizar indistintamente en una aplicación. Además, el rendimiento de tu aplicación no depende del formato de configuración que elijas, ya que todo se memoriza en caché en la primer petición.

Controladores

Un controlador es un nombre elegante para una función o método *PHP* que se encarga de las *peticiones* entrantes y devuelve las *respuestas* (a menudo código *HTML*). En lugar de utilizar variables globales y funciones *PHP* (como `$_GET` o `header()`) para manejar estos mensajes *HTTP*, *Symfony* utiliza objetos: `Symfony\Component\HttpFoundation\Request` y `Symfony\Component\HttpFoundation\Response`. El controlador más simple posible es crear la respuesta a mano, basándote en la petición:

```
use Symfony\Component\HttpFoundation\Response;

$nombre = $peticion->query->get('nombre');

return new Response('Hola '.$nombre, 200, array('Content-Type' => 'text/plain'));
```

Nota: *Symfony2* abarca la especificación *HTTP*, esta contiene las reglas que gobiernan todas las comunicaciones en la web. Lee el capítulo "*Symfony2 y fundamentos HTTP* (Página 33)" del libro para aprender más acerca de esto y la potencia que ello conlleva.

Symfony2 elige el controlador basándose en el valor de `_controller` de la configuración de enrutado: `AcmeDemoBundle:Bienvenida:index`. Esta cadena es el nombre lógico del *controlador*, y hace referencia al método `indexAction` de la clase `Acme\DemoBundle\Controller\BienvenidaController`:

```
// src/Acme/DemoBundle/Controller/BienvenidaController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BienvenidaController extends Controller
{
    public function indexAction()
    {
        return $this->render('AcmeDemoBundle:Bienvenida:index.html.twig');
    }
}
```

Truco: Podrías haber usado el nombre completo de la clase y método - `Acme\DemoBundle\Controller\BienvenidaController::indexAction` - para el valor del `_controller`. Pero si sigues algunas simples convenciones, el nombre lógico es más conciso y te permite mayor flexibilidad.

La clase `BienvenidaController` extiende la clase integrada `Controller`, la cual proporciona útiles atajos a métodos, como el **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::render'** que carga y reproduce una plantilla (`AcmeDemoBundle:Bienvenida:index.html.twing`). El valor devuelto es un objeto `Respuesta` poblado con el contenido reproducido. Por lo tanto, si surge la necesidad, la `Respuesta` se puede ajustar antes de enviarla al navegador:

```
public function indexAction()
{
    $respuesta = $this->render('AcmeDemoBundle:Bienvenida:index.txt.twig');
    $respuesta->headers->set('Content-Type', 'text/plain');

    return $respuesta;
}
```

Pero en todos los casos, el trabajo final del controlador siempre es devolver el objeto `Respuesta` que será entregado al usuario. Este objeto `Respuesta` se puede poblar con código *HTML*, representar una redirección al cliente, e incluso devolver el contenido de una imagen JPG con una cabecera `Content-Type` de `image/jpg`.

Truco: Derivar de la clase base `Controlller` es opcional. De hecho, un controlador puede ser una simple función *PHP* e incluso un cierre *PHP*. El capítulo "*Controlador* (Página 71)" del libro abarca todo sobre los controladores de *Symfony2*.

El nombre de la plantilla, `AcmeDemoBundle:Bienvenida:index.html.twig`, es el *nombre lógico* de la plantilla y hace referencia al archivo `Resources/views/Bienvenida/index.html.twig` dentro del `AcmeDemoBundle` (ubicado en `src/Acme/DemoBundle`). En la sección paquetes, a continuación, explicaré por qué esto es útil.

Ahora, de nuevo echa un vistazo a la configuración de enrutado y encuentra la clave `_demo`:

```
# app/config/routing_dev.yml
_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo
```

Symfony2 puede leer/importar la información de enrutado desde diferentes archivos escritos en *YAML*, *XML*, **PHP* o, incluso, incorporada en anotaciones *PHP*. En este caso, el *nombre lógico* del recurso es `@AcmeDemoBundle/Controller/DemoController.php` y se refiere al archivo `src/Acme/DemoBundle/Controller/DemoController.php`. En este archivo, las rutas se definen como anotaciones sobre los métodos de acción:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

class DemoController extends Controller
{
    /**
     * @Route("/hola/{nombre}", name="_demo_hola")
     * @Template()
     */
    public function holaAction($nombre)
    {
        return array('nombre' => $nombre);
    }

    // ...
}
```

La anotación `@Route()` define una nueva ruta con un patrón de `/hola/{nombre}` que ejecuta el método `holaAction` cuando concuerda. Una cadena encerrada entre llaves como `{nombre}` se conoce como marcador de posición. Como puedes ver, su valor se puede recuperar a través del argumento `$nombre` del método.

Nota: Incluso si las anotaciones no son compatibles nativamente en *PHP*, las utilizamos ampliamente en *Symfony2* como una conveniente manera de configurar el comportamiento de la plataforma y mantener la configuración del lado del código.

Si echas un vistazo más de cerca al código de la acción del controlador, puedes ver que en lugar de reproducir una plantilla y devolver un objeto `Respuesta` como antes, sólo devuelve una matriz de parámetros. La anotación `@Template()` le dice a *Symfony* que reproduzca la plantilla por ti, pasando cada variable del arreglo a la plantilla. El nombre de la plantilla reproducida sigue al nombre del controlador. Por lo tanto, en este ejemplo, se reproduce la plantilla `AcmeDemoBundle:Demo:hola.html.twig` (ubicada en `src/Acme/DemoBundle/Resources/views/Demo/hola.html.twig`).

Truco: Las anotaciones `@Route()` y `@Template()` son más poderosas que lo mostrado en el ejemplo simple de esta guía. Aprende más sobre las "anotaciones en controladores" en la documentación oficial.

Plantillas

El controlador procesa la plantilla `src/Acme/DemoBundle/Resources/views/Demo/hola.html.twig` (o `AcmeDemoBundle:Demo:hola.html.twig` si utilizas el nombre lógico):

```
{# src/Acme/DemoBundle/Resources/views/Demo/hola.html.twig #}
{% extends "AcmeDemoBundle::base.html.twig" %}

{% block titulo "Hola " ~ nombre %}

{% block contenido %}
    <h1>Hola {{ nombre }}!</h1>
{% endblock %}
```

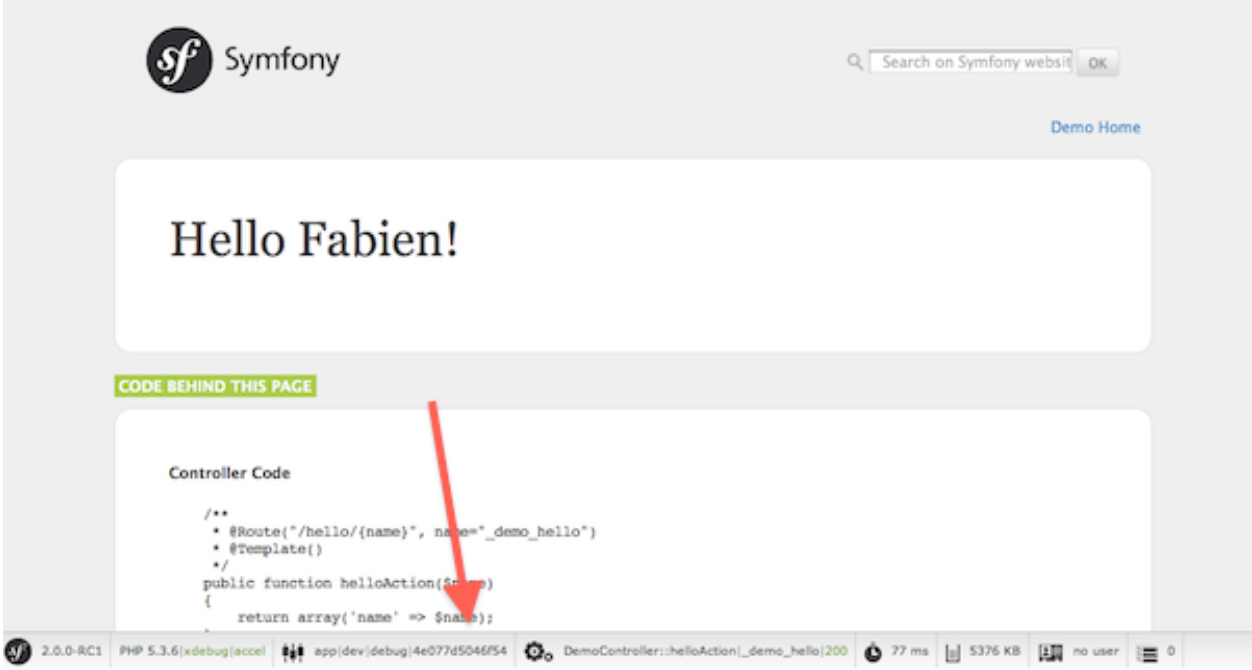
Por omisión, *Symfony2* utiliza Twig como motor de plantillas, pero también puede utilizar plantillas *PHP* tradicionales si lo deseas. El siguiente capítulo es una introducción a cómo trabajan las plantillas en *Symfony2*.

Paquetes

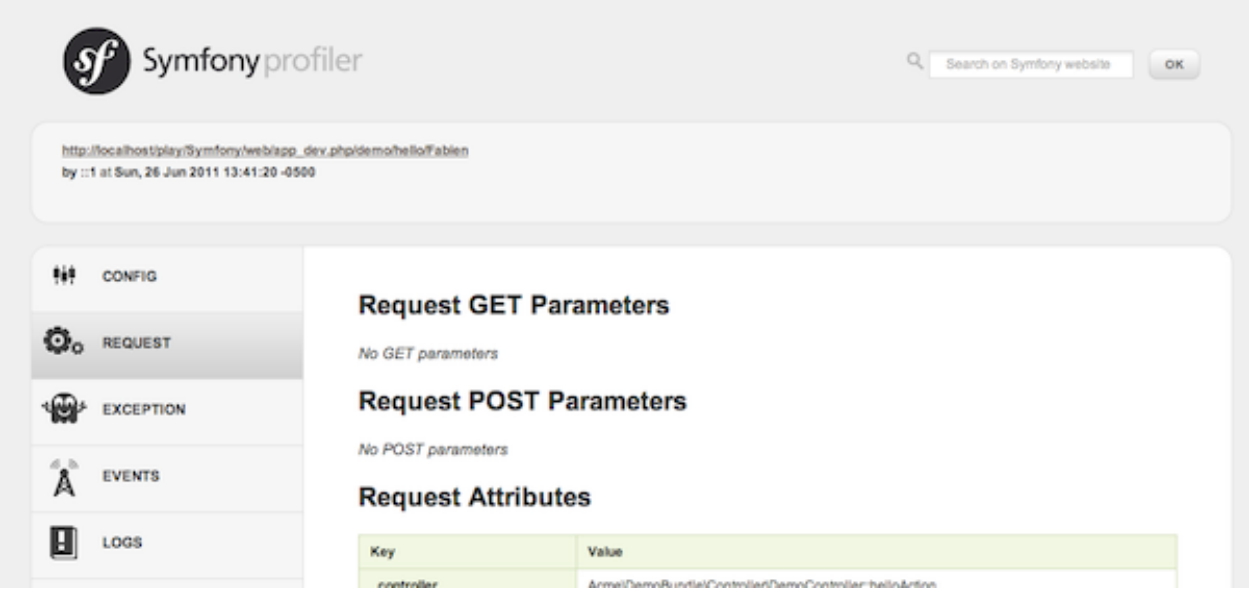
Posiblemente te hayas preguntado por qué la palabra bundle (*paquete* en adelante), se utiliza en muchos de los nombres que hemos visto hasta ahora. Todo el código que escribas para tu aplicación está organizado en paquetes. Hablando en *Symfony2*, un paquete es un conjunto estructurado de archivos (archivos *PHP*, hojas de estilo, JavaScript, imágenes, ...) que implementa una sola característica (un blog, un foro, ...) y que fácilmente se puede compartir con otros desarrolladores. Hasta ahora, hemos manipulado un paquete, `AcmeDemoBundle`. Aprenderás más acerca de los paquetes en el último capítulo de esta guía.

1.1.4 Trabajando con entornos

Ahora que tienes una mejor comprensión de cómo funciona *Symfony2*, dale una mirada más atenta a la parte inferior de cualquier página reproducida por *Symfony2*. Deberás notar una pequeña barra con el logotipo de *Symfony2*. Esta se conoce como la "barra de depuración web" y es la mejor amiga del desarrollador.



Pero lo que ves al principio es sólo la punta del iceberg; haz clic en el extraño número hexadecimal para revelar otra muy útil herramienta de depuración de *Symfony2*: el generador de perfiles.



Por supuesto, no querrás mostrar estas herramientas al desplegar tu aplicación en producción. Es por eso que encontrarás otro controlador frontal en el directorio `web/` (`app.php`), el cual está optimizado para el entorno de producción:

```
http://localhost/Symfony/web/app.php/demo/hola/Nacho
```

Y si utilizas Apache con `mod_rewrite` habilitado, incluso puedes omitir la parte `app.php` de la *URL*:

```
http://localhost/Symfony/web/demo/hola/Nacho
```

Por último pero no menos importante, en los servidores en producción, debes apuntar tu directorio web raíz al directorio `web/` para proteger tu instalación e incluso, para que tus direcciones *URL* tengan un mejor aspecto:

```
http://localhost/demo/hola/Nacho
```

Para hacer que la aplicación responda más rápido, *Symfony2* mantiene una caché en el directorio `app/cache/`. En el entorno de desarrollo (`app_dev.php`), esta caché se vacía automáticamente cada vez que realizas cambios en cualquier código o configuración. Pero ese no es el caso en el entorno de producción (`app.php`) donde el rendimiento es clave. Es por eso que siempre debes utilizar el entorno de desarrollo al estar desarrollando tu aplicación.

Diferentes *entornos* de una determinada aplicación sólo se diferencian en su configuración. De hecho, una configuración puede heredar de otra:

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

web_profiler:
    toolbar: true
    intercept_redirects: false
```

El entorno `dev` (el cual carga el archivo de configuración `config_dev.yml`) importa el archivo global `config.yml` y luego lo modifica, en este ejemplo, activando la barra de herramientas para depuración web.

1.1.5 Consideraciones finales

¡Enhorabuena! Has tenido tu primera experiencia codificando en *Symfony2*. No fue tan difícil, ¿cierto? Hay mucho más por explorar, pero ya debes tener una idea de cómo *Symfony2* facilita la implementación de mejores y más rápidos sitios web. Si estás interesado en aprender más acerca de *Symfony2*, sumérgete en la siguiente sección: "*La vista* (Página 13)".

1.2 La vista

Después de leer la primera parte de esta guía, has decidido que bien valen la pena otros 10 minutos en *Symfony2*. ¡Buena elección! En esta segunda parte, aprenderás más sobre el motor de plantillas de *Symfony2*, Twig. *Twig* es un motor de plantillas flexible, rápido y seguro para *PHP*. Este hace tus plantillas más legibles y concisas, además de hacerlas más amigables para los diseñadores web.

Nota: En lugar de *Twig*, también puedes utilizar *PHP* (Página 403) para tus plantillas. Ambos motores de plantillas son compatibles con *Symfony2*.

1.2.1 Familiarizándote con *Twig*

Truco: Si quieres aprender *Twig*, te recomendamos que leas la documentación oficial. Esta sección es sólo una descripción rápida de los conceptos principales.

Una plantilla *Twig* es un archivo de texto que puede generar cualquier tipo de contenido (*HTML*, *XML*, *CSV*, *LaTeX*, ...). *Twig* define dos tipos de delimitadores:

- `{{ ... }}`: Imprime una variable o el resultado de una expresión;
- `{% ... %}`: Controla la lógica de la plantilla; se utiliza para ejecutar `bucles for` y declaraciones `if`, por ejemplo.

A continuación mostramos una plantilla mínima que ilustra algunos conceptos básicos, usando dos variables `titulo_pag` y `navegacion`, mismas que se pasaron a la plantilla:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Mi página Web</title>
    </head>
    <body>
        <h1>{{ titulo_pag }}</h1>

        <ul id="navegacion">
            {% for elemento in navegacion %}
                <li><a href="{{ elemento.href }}">{{ elemento.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Truco: Puedes incluir comentarios dentro de las plantillas con el delimitador `{# ... #}`.

Para reproducir una plantilla en *Symfony*, utiliza el método `render` dentro de un controlador, suministrando cualquier variable necesaria en la plantilla:

```
$this->render('AcmeDemoBundle:Demo:hola.html.twig', array(
    'nombre' => $nombre,
));
```

Las variables pasadas a una plantilla pueden ser cadenas, matrices e incluso objetos. *Twig* abstrae la diferencia entre ellas y te permite acceder a los "atributos" de una variable con la notación de punto (`.`):

```
{# array('nombre' => 'Fabien') #}
{{ nombre }}

{# array('usuario' => array('nombre' => 'Fabien')) #}
{{ usuario.nombre }}

{# obliga a verlo como arreglo #}
{{ usuario['nombre'] }}

{# array('usuario' => new Usuario('Fabien')) #}
{{ usuario.nombre }}
{{ usuario.getNombre }}

{# obliga a ver el nombre como método #}
{{ usuario.nombre() }}
{{ usuario.getNombre() }}

{# pasa argumentos al método #}
{{ usuario.fecha('Y-m-d') }}
```

Nota: Es importante saber que las llaves no son parte de la variable, sino de la declaración de impresión. Si accedes a variables dentro de las etiquetas no las envuelvas con llaves.

1.2.2 Decorando plantillas

Muy a menudo, las plantillas en un proyecto comparten elementos comunes, como los bien conocidos encabezados y pies de página. En *Symfony2*, nos gusta pensar en este problema de forma diferente: una plantilla se puede decorar con otra. Esto funciona exactamente igual que las clases *PHP*: la herencia de plantillas permite crear un "diseño" de plantilla básico que contiene todos los elementos comunes de tu sitio y define "bloques" que las plantillas descendientes pueden reemplazar.

La plantilla `hola.html.twig` hereda de `base.html.twig`, gracias a la etiqueta `extends`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hola.html.twig #}
{% extends "AcmeDemoBundle::base.html.twig" %}

{% block titulo "Hola " ~ nombre %}

{% block contenido %}
    <h1>Hola {{ nombre }}!</h1>
{% endblock %}
```

La notación `AcmeDemoBundle::base.html.twig` suena familiar, ¿no? Es la misma notación utilizada para hacer referencia a una plantilla regular. La parte `::` simplemente significa que el elemento controlador está vacío, por lo tanto el archivo correspondiente se almacena directamente bajo el directorio `Resources/views/`.

Ahora, echemos un vistazo a un `base.html.twig` simplificado:

```
{# src/Acme/DemoBundle/Resources/views/base.html.twig #}
<div class="contenido-symfony">
    {% block contenido %}
    {% endblock %}
</div>
```

La etiqueta `{% block %}` define bloques que las plantillas derivadas pueden llenar. Todas las etiquetas de bloque le dicen al motor de plantillas que una plantilla derivada puede reemplazar esas porciones de la plantilla.

En este ejemplo, la plantilla `hola.html.twig` sustituye el bloque `contenido`, lo cual significa que el texto "Hola Fabien" se reproduce dentro del elemento `div.contenido-symfony`.

1.2.3 Usando etiquetas, filtros y funciones

Una de las mejores características de *Twig* es su extensibilidad a través de etiquetas, filtros y funciones. *Symfony2* viene empacado con muchas de estas integradas para facilitar el trabajo del diseñador de la plantilla.

Incluyendo otras plantillas

La mejor manera de compartir un fragmento de código entre varias plantillas diferentes es crear una nueva plantilla, que luego puedas incluir en otras plantillas.

Crea una plantilla `integrada.html.twig`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/integrada.html.twig #}
Hola {{ nombre }}
```

Y cambia la plantilla `index.html.twig` para incluirla:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hola.html.twig #}
{% extends "AcmeDemoBundle::base.html.twig" %}

{# sustituye el bloque 'contenido' con el de integrada.html.twig #}
```

```
{% block contenido %}
    {% include "AcmeDemoBundle:Demo:integrada.html.twig" %}
{% endblock %}
```

Integrando otros controladores

¿Y si deseas incrustar el resultado de otro controlador en una plantilla? Eso es muy útil cuando se trabaja con Ajax, o cuando la plantilla incrustada necesita alguna variable que no está disponible en la plantilla principal.

Supongamos que has creado una acción `maravillosa`, y deseas incluirla dentro de la plantilla principal `index`. Para ello, utiliza la etiqueta `render`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/index.html.twig #}
{% render "AcmeDemoBundle:Demo:maravillosa" with { 'nombre': nombre, 'color': 'verde' } %}
```

Aquí, la cadena `AcmeDemoBundle:Demo:maravillosa` se refiere a la acción `maravillosa` del controlador `Demo`. Los argumentos (`nombre` y `color`) actúan como variables de la petición simulada (como si `maravillosaAction` estuviera manejando una petición completamente nueva) y se pone a disposición del controlador:

```
// src/Acme/DemoBundle/Controller/DemoController.php

class DemoController extends Controller
{
    public function maravillosaAction($nombre, $color)
    {
        // crea algún objeto, basándose en la variable $color
        $objeto = ...;

        return $this->render('AcmeDemoBundle:Demo:maravillosa.html.twig', array('nombre' => $nombre,
    }

    // ...
}
```

Creando enlaces entre páginas

Hablando de aplicaciones web, crear enlaces entre páginas es una necesidad. En lugar de codificar las direcciones *URL* en las plantillas, la función `path` sabe cómo generar direcciones *URL* basándose en la configuración de enrutado. De esta manera, todas tus direcciones *URL* se pueden actualizar fácilmente con sólo cambiar la configuración:

```
<a href="{{ path('_demo_hola', { 'nombre': 'Tomás' }) }}">¡Hola Tomás!</a>
```

La función `path` toma el nombre de la ruta y una matriz de parámetros como argumentos. El nombre de la ruta es la clave principal en la cual se hace referencia a las rutas y los parámetros son los valores de los marcadores de posición definidos en el patrón de rutas:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hola/{nombre}", name="_demo_hola")
 * @Template()
 */
public function holaAction($nombre)
```

```
{
    return array('nombre' => $nombre);
}
```

Truco: La función `url` genera direcciones *URL absolutas*: `{{ url('_demo_hola', { 'nombre': 'Tomás' }) }}`.

Incluyendo Activos: imágenes, JavaScript y hojas de estilo

¿Qué sería de Internet sin imágenes, JavaScript y hojas de estilo? *Symfony2* proporciona la función `asset` para hacerles frente fácilmente:

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

<img src="{{ asset('images/logo.png') }}" />
```

El propósito principal de la función `asset` es hacer más portátil tu aplicación. Gracias a esta función, puedes mover el directorio raíz de la aplicación a cualquier lugar bajo tu directorio web raíz sin cambiar nada en el código de tus plantillas.

1.2.4 Escapando variables

Twig se configura de forma automática escapando toda salida de forma predeterminada. Lee la documentación de *Twig* para obtener más información sobre el mecanismo de escape y la extensión 'Escaper'.

1.2.5 Consideraciones finales

Twig es simple pero potente. Gracias a los diseños, bloques, plantillas e inclusión de acciones, es muy fácil organizar tus plantillas de manera lógica y extensible. Sin embargo, si no te sientes cómodo con *Twig*, siempre puedes utilizar las plantillas de *PHP* dentro de *Symfony* sin ningún problema.

Sólo has estado trabajando con *Symfony2* durante unos 20 minutos, pero ya puedes hacer cosas muy sorprendentes con él. Ese es el poder de *Symfony2*. Aprender los conceptos básicos es fácil, y pronto aprenderás que esta simplicidad está escondida bajo una arquitectura muy flexible.

Pero me estoy adelantando demasiado. En primer lugar, necesitas aprender más sobre el controlador y ese es exactamente el tema de la *siguiente parte de esta guía* (Página 17). ¿Listo para otros 10 minutos con *Symfony2*?

1.3 El controlador

¿Todavía con nosotros después de las dos primeras partes? ¡Ya te estás volviendo adicto a *Symfony2*! Sin más preámbulos, vamos a descubrir lo que los controladores pueden hacer por ti.

1.3.1 Usando Formatos

Hoy día, una aplicación web debe ser capaz de ofrecer algo más que solo páginas *HTML*. Desde *XML* para alimentadores *RSS* o Servicios *Web*, hasta *JSON* para peticiones *Ajax*, hay un montón de formatos diferentes a elegir. Apoyar estos formatos en *Symfony2* es sencillo. Modifica la ruta añadiendo un valor predeterminado de `xml` a la variable `_format`:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hola/{nombre}", defaults={"_format"="xml"}, name="_demo_hola")
 * @Template()
 */
public function holaAction($nombre)
{
    return array('nombre' => $nombre);
}
```

Al utilizar el formato de la petición (como lo define el valor `_format`), *Symfony2* automáticamente selecciona la plantilla adecuada, aquí `hola.xml.twig`:

```
<!-- src/Acme/DemoBundle/Resources/views/Demo/hola.xml.twig -->
<hola>
    <nombre>{{ nombre }}</nombre>
</hola>
```

Eso es todo lo que hay que hacer. Para los formatos estándar, *Symfony2* también elije automáticamente la mejor cabecera `Content-Type` para la respuesta. Si quieres apoyar diferentes formatos para una sola acción, en su lugar, usa el marcador de posición `{_format}` en el patrón de la ruta:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hola/{nombre}.{_format}", defaults={"_format"="html"}, requirements={"_format"="html|xml
 * @Template()
 */
public function holaAction($nombre)
{
    return array('nombre' => $nombre);
}
```

El controlador ahora será llamado por la URL como `/demo/hola/Fabien.xml` o `/demo/hola/Fabien.json`.

La entrada `requirements` define las expresiones regulares con las cuales los marcadores de posición deben coincidir. En este ejemplo, si tratas de solicitar el recurso `/demo/hola/Fabien.js`, obtendrás un error HTTP 404, ya que no coincide con el requisito de `_format`.

1.3.2 Redirigiendo y reenviando

Si deseas redirigir al usuario a otra página, utiliza el método `redirect()`:

```
return $this->redirect($this->generateUrl('_demo_hola', array('nombre' => 'Lucas')));
```

El método `generateUrl()` es el mismo que la función `path()` que utilizamos en las plantillas. Este toma el nombre de la ruta y una serie de parámetros como argumentos y devuelve la *URL* amigable asociada.

Además, fácilmente puedes reenviar a otra acción con el método `forward()`. Internamente, *Symfony* hace una "subpetición", y devuelve el objeto `Respuesta` desde la subpetición:

```
$respuesta = $this->forward('AcmeDemoBundle:Hola:maravillosa', array('nombre' => $nombre, 'color' =>

// hace algo con la respuesta o la devuelve directamente
```

1.3.3 Obteniendo información de la petición

Además del valor de los marcadores de posición de enrutado, el controlador también tiene acceso al objeto `Petición`:

```
$peticion = $this->getRequest();

$peticion->isXmlHttpRequest(); // ¿es una petición Ajax?

$peticion->getPreferredLanguage(array('en', 'es'));

$peticion->query->get('pag'); // consigue un parámetro $_GET

$peticion->request->get('pag'); // consigue un parámetro $_POST
```

En una plantilla, también puedes acceder al objeto `Petición` por medio de la variable `app.request`:

```
{{ app.request.query.get('pag') }}

{{ app.request.parameter('pag') }}
```

1.3.4 Persistiendo datos en la sesión

Aunque el protocolo *HTTP* es sin estado, *Symfony2* proporciona un agradable objeto sesión que representa al cliente (sea una persona real usando un navegador, un robot o un servicio web). Entre dos peticiones, *Symfony2* almacena los atributos en una `cookie` usando las sesiones nativas de *PHP*.

Almacenar y recuperar información de la sesión se puede conseguir fácilmente desde cualquier controlador:

```
$sesion = $this->getRequest()->getSession();

// guarda un atributo para reutilizarlo durante una posterior petición del usuario
$sesion->set('foo', 'bar');

// en otro controlador por otra petición
$foo = $sesion->get('foo');

// fija la configuración regional del usuario
$sesion->setLocale('es');
```

También puedes almacenar pequeños mensajes que sólo estarán disponibles para la siguiente petición:

```
// guarda un mensaje para la siguiente petición (en un controlador)
$sesion->setFlash('aviso', '¡Felicidades, tu acción fue exitosa!');

// muestra el mensaje de nuevo en la siguiente petición (en una plantilla)
{{ app.session.flash('aviso') }}
```

Esto es útil cuando es necesario configurar un mensaje de éxito antes de redirigir al usuario a otra página (la cual entonces mostrará el mensaje).

1.3.5 Protegiendo recursos

La edición estándar de *Symfony* viene con una configuración de seguridad sencilla adaptada a las necesidades más comunes:

```
# 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' ] }

    firewalls:
        dev:
            pattern:  ^/(_(profiler|wdt)|css|images|js)/
            security: false

        login:
            pattern:  ^/demo/secured/login$
            security: false

        secured_area:
            pattern:    ^/demo/secured/
            form_login:
                check_path: /demo/secured/login_check
                login_path: /demo/secured/login
            logout:
                path:   /demo/secured/logout
                target: /demo/
```

Esta configuración requiere que los usuarios inicien sesión para cualquier *URL* que comienza con `/demo/secured/` y define dos usuarios válidos: `user` y `admin`. Por otra parte, el usuario `admin` tiene un rol `ROLE_ADMIN`, el cual incluye el rol `ROLE_USER` también (consulta el ajuste `role_hierarchy`).

Truco: Para facilitar la lectura, las contraseñas se almacenan en texto plano en esta configuración simple, pero puedes usar cualquier algoritmo de codificación ajustando la sección `encoders`.

Al ir a la dirección `http://localhost/Symfony/web/app_dev.php/demo/secured/hola` automáticamente redirigirá al formulario de acceso, porque el recurso está protegido por un `cortafuegos`.

También puedes forzar la acción para exigir un determinado rol usando la anotación `@Secure` en el controlador:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Route("/hola/admin/{nombre}", name="_demo_secured_hola_admin")
 * @Secure(roles="ROLE_ADMIN")
 * @Template()
```

```
 */
public function holaAdminAction($nombre)
{
    return array('nombre' => $nombre);
}
```

Ahora, inicia sesión como `user` (el cual no *tiene* el rol `ROLE_ADMIN`) y desde la página protegida `hola`, haz clic en el enlace "Hola recurso protegido". *Symfony2* debe devolver un código de estado *HTTP 403*, el cual indica que el usuario tiene "prohibido" el acceso a ese recurso.

Nota: La capa de seguridad de *Symfony2* es muy flexible y viene con muchos proveedores de usuario diferentes (por ejemplo, uno para el *ORM* de *Doctrine*) y proveedores de autenticación (como *HTTP* básica, *HTTP digest* o certificados X509). Lee el capítulo "*Seguridad* (Página 181)" del libro para más información en cómo se usa y configura.

1.3.6 Memorizando recursos en caché

Tan pronto como tu sitio web comience a generar más tráfico, tendrás que evitar se genere el mismo recurso una y otra vez. *Symfony2* utiliza cabeceras de caché *HTTP* para administrar los recursos en caché. Para estrategias de memorización en caché simples, utiliza la conveniente anotación `@Cache()`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;

/**
 * @Route("/hola/{nombre}", name="_demo_hola")
 * @Template()
 * @Cache(maxage="86400")
 */
public function holaAction($nombre)
{
    return array('nombre' => $nombre);
}
```

En este ejemplo, el recurso se mantiene en caché por un día. Pero también puedes utilizar validación en lugar de caducidad o una combinación de ambas, si se ajusta mejor a tus necesidades.

El recurso memorizado en caché es gestionado por el delegado inverso integrado en *Symfony2*. Pero debido a que la memorización en caché se gestiona usando cabeceras de caché *HTTP*, puedes reemplazar el delegado inverso integrado, con *Varnish* o *Squid* y escalar tu aplicación fácilmente.

Nota: Pero ¿qué pasa si no puedes guardar en caché todas las páginas? *Symfony2* todavía tiene la solución vía ESI (Edge Side Includes o Inclusión de borde lateral), con la cual es compatible nativamente. Consigue más información leyendo el capítulo "*Caché HTTP* (Página 210)" del libro.

1.3.7 Consideraciones finales

Eso es todo lo que hay que hacer, y ni siquiera estoy seguro de que hayan pasado los 10 minutos completos. Presentamos brevemente los paquetes en la primera parte, y todas las características que hemos explorado hasta ahora son parte del paquete básico de la plataforma. Pero gracias a los paquetes, todo en *Symfony2* se puede ampliar o sustituir. Ese, es el tema de la *siguiente parte de esta guía* (Página 22).

1.4 La arquitectura

¡Eres mi héroe! ¿Quién habría pensado que todavía estarías aquí después de las tres primeras partes? Tu esfuerzo pronto será bien recompensado. En las tres primeras partes no vimos en demasiada profundidad la arquitectura de la plataforma. Porque esta hace que *Symfony2* esté al margen de la multitud de plataformas, ahora vamos a profundizar en la arquitectura.

1.4.1 Comprendiendo la estructura de directorios

La estructura de directorios de una *aplicación Symfony2* es bastante flexible, pero la estructura de directorios de la distribución de la *edición estándar* refleja la estructura típica y recomendada de una aplicación *Symfony2*:

- `app/`: La configuración de la aplicación;
- `src/`: El código *PHP* del proyecto;
- `vendor/`: Las dependencias de terceros;
- `web/`: La raíz del directorio web.

El Directorio `web/`

El directorio web raíz, es el hogar de todos los archivos públicos y estáticos tales como imágenes, hojas de estilo y archivos *JavaScript*. También es el lugar donde vive cada *controlador frontal*:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$nucleo = new AppKernel('prod', false);
$nucleo->loadClassCache();
$nucleo->handle(Request::createFromGlobals())->send();
```

El núcleo requiere en primer lugar el archivo `bootstrap.php.cache`, el cual arranca la plataforma y registra el cargador automático (ve más abajo).

Al igual que cualquier controlador frontal, `app.php` utiliza una clase del núcleo, `AppKernel`, para arrancar la aplicación.

El directorio `app/`

La clase `AppKernel` es el punto de entrada principal para la configuración de la aplicación y, como tal, se almacena en el directorio `app/`.

Esta clase debe implementar dos métodos:

- `registerBundles()` debe devolver una matriz de todos los paquetes necesarios para ejecutar la aplicación;
- `registerContainerConfiguration()` carga la configuración de la aplicación (más sobre esto más adelante).

La carga automática de clases *PHP* se puede configurar a través de `app/autoload.php`:

```
// app/autoload.php
use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony'          => array(__DIR__.'/../vendor/symfony/src', __DIR__.'/../vendor/bundles'),
    'Sensio'           => __DIR__.'/../vendor/bundles',
    'JMS'              => __DIR__.'/../vendor/bundles',
    'Doctrine\\Common' => __DIR__.'/../vendor/doctrine-common/lib',
    'Doctrine\\DBAL'   => __DIR__.'/../vendor/doctrine-dbal/lib',
    'Doctrine'         => __DIR__.'/../vendor/doctrine/lib',
    'Monolog'          => __DIR__.'/../vendor/monolog/src',
    'Assetic'          => __DIR__.'/../vendor/assetic/src',
    'Metadata'         => __DIR__.'/../vendor/metadata/src',
));
$loader->registerPrefixes(array(
    'Twig_Extensions_' => __DIR__.'/../vendor/twig-extensions/lib',
    'Twig_'            => __DIR__.'/../vendor/twig/lib',
));

// ...

$cargador->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
$cargador->register();
```

La `Symfony\Component\ClassLoader\UniversalClassLoader` se usa para cargar automáticamente archivos que respetan tanto los estándares de interoperabilidad técnica de los espacios de nombres de *PHP 5.3* como la convención de nomenclatura de las clases *PEAR*. Como puedes ver aquí, todas las dependencias se guardan bajo el directorio `vendor/`, pero esto es sólo una convención. Puedes guardarlas donde quieras, a nivel global en el servidor o localmente en tus proyectos.

Nota: Si deseas obtener más información sobre la flexibilidad del autocargador de *Symfony2*, lee la fórmula "*Cómo cargar clases automáticamente* (Página 407)" en el recetario.

1.4.2 Comprendiendo el sistema de paquetes

Esta sección introduce una de las más importantes y poderosas características de *Symfony2*, el sistema de *paquetes*.

Un paquete es un poco como un complemento en otros programas. Así que ¿por qué se llama *paquete* y no *complemento*? Esto se debe a que en *Symfony2* **todo** es un paquete, desde las características del núcleo de la plataforma hasta el código que escribes para tu aplicación. Los paquetes son ciudadanos de primera clase en *Symfony2*. Esto te proporciona la flexibilidad para utilizar las características preconstruidas envasadas en paquetes de terceros o para distribuir tus propios paquetes. Además, facilita la selección y elección de las características por habilitar en tu aplicación y optimizarlas en la forma que desees. Y al final del día, el código de tu aplicación es tan *importante* como el mismo núcleo de la plataforma.

Registrando un paquete

Una aplicación se compone de paquetes tal como está definido en el método `registerBundles()` de la clase `AppKernel`. Cada paquete vive en un directorio que contiene una única clase `Paquete` que lo describe:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

Además de `AcmeDemoBundle` del cual ya hemos hablado, observa que el núcleo también habilita otros paquetes como `FrameworkBundle`, `DoctrineBundle`, `SwiftmailerBundle` y `AsseticBundle`. Todos ellos son parte del núcleo de la plataforma.

Configurando un paquete

Cada paquete se puede personalizar a través de los archivos de configuración escritos en *YAML*, *XML* o *PHP*. Echa un vistazo a la configuración predeterminada:

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

framework:
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    session:
        default_locale: %locale%
        auto_start:     true

# Configuración Twig
twig:
    debug:            %kernel.debug%
    strict_variables: %kernel.debug%

# Configuración de Assetic
```

```
assetic:
    debug:          %kernel.debug%
    use_controller: false
    filters:
        cssrewrite: ~
        # closure:
        #     jar: %kernel.root_dir%/java/compiler.jar
        # yui_css:
        #     jar: %kernel.root_dir%/java/yuicompressor-2.4.2.jar

# Configuración de Doctrine
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
        charset:  UTF8

    orm:
        auto_generate_proxy_classes: %kernel.debug%
        auto_mapping: true

# Configuración de Swiftmailer
swiftmailer:
    transport: %mailer_transport%
    host:      %mailer_host%
    username:  %mailer_user%
    password:  %mailer_password%

jms_security_extra:
    secure_controllers:  true
    secure_all_services: false
```

Cada entrada como `framework` define la configuración de un paquete específico. Por ejemplo, `framework` configura el `FrameworkBundle` mientras que `swiftmailer` configura el `SwiftmailerBundle`.

Cada *entorno* puede reemplazar la configuración predeterminada proporcionando un archivo de configuración específico. Por ejemplo, el entorno `dev` carga el archivo `config_dev.yml`, el cual carga la configuración principal (es decir, `config.yml`) y luego la modifica agregando algunas herramientas de depuración:

```
# 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

monolog:
    handlers:
        main:
            type:  stream
            path:  %kernel.logs_dir%/%kernel.environment%.log
```

```
            level: debug
        firephp:
            type:  firephp
            level: info

assetic:
    use_controller: true
```

Extendiendo un paquete

Además de ser una buena manera de organizar y configurar tu código, un paquete puede extender otro paquete. La herencia de paquetes te permite sustituir cualquier paquete existente con el fin de personalizar sus controladores, plantillas, o cualquiera de sus archivos. Aquí es donde son útiles los nombres lógicos (por ejemplo, `@AcmeDemoBundle/Controller/SecuredController.php`): estos abstraen en dónde se almacena el recurso.

Nombres lógicos de archivo

Cuando quieras hacer referencia a un archivo de un paquete, utiliza esta notación: `@NOMBRE_PAQUETE/ruta/al/archivo`; *Symfony2* resolverá `@NOMBRE_PAQUETE` a la ruta real del paquete. Por ejemplo, la ruta lógica `@AcmeDemoBundle/Controller/DemoController.php` se convierte en `src/Acme/DemoBundle/Controller/DemoController.php`, ya que *Symfony* conoce la ubicación del `AcmeDemoBundle`.

Nombres lógicos de Controlador

Para los controladores, necesitas hacer referencia a los nombres de método usando el formato `NOMBRE_PAQUETE:NOMBRE_CONTROLADOR:NOMBRE_ACCIÓN`. Por ejemplo, `AcmeDemoBundle:Bienvenida:index` representa al método `indexAction` de la clase `Acme\DemoBundle\Controller\BienvenidaController`.

Nombres lógicos de plantilla

Para las plantillas, el nombre lógico `AcmeDemoBundle:Bienvenida:index.html.twig` se convierte en la ruta del archivo `src/Acme/DemoBundle/Resources/views/Bienvenida/index.html.twig`. Incluso las plantillas son más interesantes cuando te das cuenta que no es necesario almacenarlas en el sistema de archivos. Puedes guardarlas fácilmente en una tabla de la base de datos, por ejemplo.

Extendiendo paquetes

Si sigues estas convenciones, entonces puedes utilizar *herencia de paquetes* (Página 350) para "redefinir" archivos, controladores o plantillas. Por ejemplo, si un nuevo paquete llamado `AcmeNuevoBundle` extiende el `AcmeDemoBundle`, entonces *Symfony* primero intenta cargar el controlador `AcmeDemoBundle:Bienvenida:index` de `AcmeNuevoBundle`, y luego ve dentro de `AcmeDemoBundle`.

¿Entiendes ahora por qué *Symfony2* es tan flexible? Comparte tus paquetes entre aplicaciones, guárdalas local o globalmente, tú eliges.

1.4.3 Usando vendors

Lo más probable es que tu aplicación dependerá de bibliotecas de terceros. Estas se deberían guardar en el directorio `vendor/`. Este directorio ya contiene las bibliotecas *Symfony2*, la biblioteca *SwiftMailer*, el *ORM* de *Doctrine*, el sistema de plantillas *Twig* y algunas otras bibliotecas y paquetes de terceros.

1.4.4 Comprendiendo la caché y los registros

Symfony2 probablemente es una de las plataformas más rápidas hoy día. Pero ¿cómo puede ser tan rápida si analiza e interpreta decenas de archivos *YAML* y *XML* por cada petición? La velocidad, en parte, se debe a su sistema de caché. La configuración de la aplicación sólo se analiza en la primer petición y luego se compila hasta código *PHP* simple y se guarda en el directorio `app/cache/`. En el entorno de desarrollo, *Symfony2* es lo suficientemente inteligente como para vaciar la caché cuando cambias un archivo. Pero en el entorno de producción, es tu responsabilidad borrar la caché cuando actualizas o cambias tu código o configuración.

Al desarrollar una aplicación web, las cosas pueden salir mal de muchas formas. Los archivos de registro en el directorio `app/logs/` dicen todo acerca de las peticiones y ayudan a solucionar rápidamente el problema.

1.4.5 Usando la interfaz de línea de ordenes

Cada aplicación incluye una herramienta de interfaz de línea de ordenes (`app/console`) que te ayuda a mantener la aplicación. Esta proporciona ordenes que aumentan tu productividad automatizando tediosas y repetitivas tareas.

Ejecútalo sin argumentos para obtener más información sobre sus posibilidades:

```
php app/console
```

La opción `--help` te ayuda a descubrir el uso de una orden:

```
php app/console router:debug --help
```

1.4.6 Consideraciones finales

Llámame loco, pero después de leer esta parte, debes sentirte cómodo moviendo cosas y haciendo que *Symfony2* trabaje por ti. Todo en *Symfony2* está diseñado para allanar tu camino. Por lo tanto, no dudes en renombrar y mover directorios como mejor te parezca.

Y eso es todo para el inicio rápido. Desde probar hasta enviar mensajes de correo electrónico, todavía tienes que aprender mucho para convertirte en gurú de *Symfony2*. ¿Listo para zambullirte en estos temas ahora? No busques más - revisa el *Libro* (Página 33) oficial y elije cualquier tema que desees.

- *Un primer vistazo* (Página 5) >
- *La vista* (Página 13) >
- *El controlador* (Página 17) >
- *La arquitectura* (Página 22)

Parte II

Libro

Sumérgete en *Symfony2* con las guías temáticas:

Libro

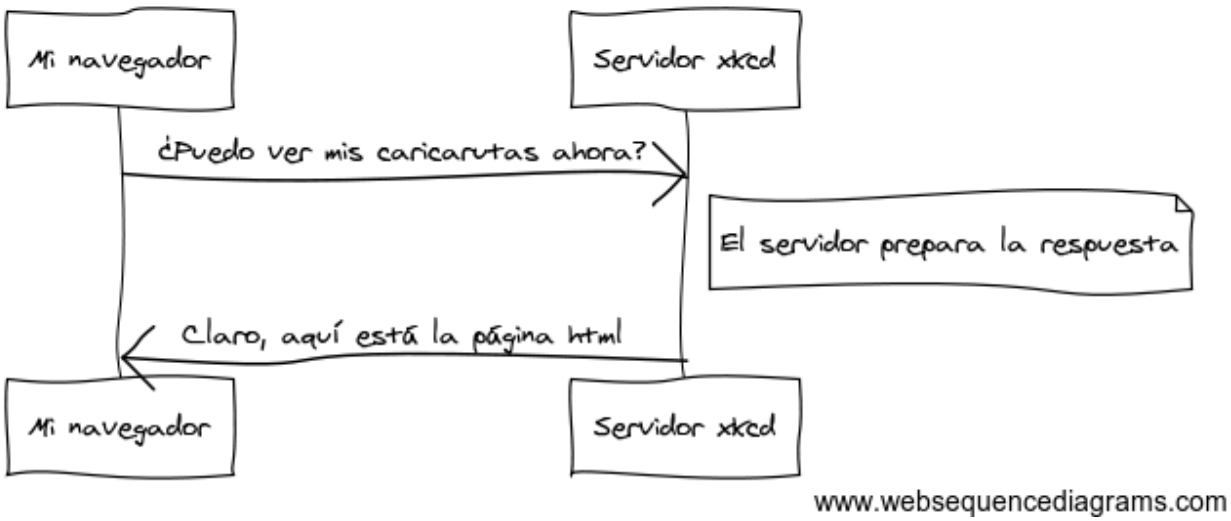
2.1 *Symfony2* y fundamentos *HTTP*

¡Enhorabuena! Al aprender acerca de *Symfony2*, vas bien en tu camino para llegar a ser un más *productivo*, bien *enfocado* y *popular* desarrollador web (en realidad, en la última parte, estás por tu cuenta). *Symfony2* está diseñado para volver a lo básico: las herramientas de desarrollo que te permiten desarrollar más rápido y construir aplicaciones más robustas, mientras que permanece fuera de tu camino. *Symfony* está basado en las mejores ideas de muchas tecnologías: las herramientas y conceptos que estás a punto de aprender representan el esfuerzo de miles de personas, durante muchos años. En otras palabras, no estás aprendiendo *"Symfony"*, estás aprendiendo los fundamentos de la *web*, buenas prácticas de desarrollo, y cómo utilizar muchas nuevas y asombrosas bibliotecas *PHP*, dentro o independientemente de *Symfony2*. Por lo tanto, ¡prepárate!

Fiel a la filosofía *Symfony2*, este capítulo comienza explicando el concepto fundamental común para el desarrollo *web*: *HTTP*. Independientemente de tus antecedentes o lenguaje de programación preferido, este capítulo es una **lectura obligada** para todo mundo.

2.1.1 *HTTP* es Simple

HTTP ("HyperText Transfer Protocol" para los apasionados y, en Español *Protocolo de transferencia hipertexto*) es un lenguaje de texto que permite a dos máquinas comunicarse entre sí. ¡Eso es todo! Por ejemplo, al comprobar las últimas noticias acerca de cómica xkcd, la siguiente conversación (aproximadamente) se lleva a cabo:



Y aunque el lenguaje real utilizado es un poco más formal, sigue siendo bastante simple. *HTTP* es el término utilizado para describir este lenguaje simple basado en texto. Y no importa cómo desarrolles en la web, el objetivo de tu servidor *siempre* es entender las peticiones de texto simple, y devolver respuestas en texto simple.

Symfony2 está construido basado en torno a esa realidad. Ya sea que te des cuenta o no, *HTTP* es algo que usas todos los días. Con *Symfony2*, aprenderás a dominarlo.

Paso 1: El cliente envía una petición

Todas las conversaciones en la web comienzan con una *petición*. La petición es un mensaje de texto creado por un cliente (por ejemplo un navegador, una aplicación para el *iPhone*, etc.) en un formato especial conocido como *HTTP*. El cliente envía la petición a un servidor, y luego espera la respuesta.

Echa un vistazo a la primera parte de la interacción (la petición) entre un navegador y el servidor web *xkcd*:

Hablando en *HTTP*, esta petición *HTTP* en realidad se vería algo parecida a esto:

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

Este sencillo mensaje comunica *todo* lo necesario sobre qué recursos exactamente solicita el cliente. La primera línea de una petición *HTTP* es la más importante y contiene dos cosas: la *URI* y el método *HTTP*.

La *URI* (por ejemplo, `/`, `/contacto`, etc.) es la dirección o ubicación que identifica unívocamente al recurso que el cliente quiere. El método *HTTP* (por ejemplo, `GET`) define lo que quieres *hacer* con el recurso. Los métodos *HTTP* son los *verbos* de la petición y definen las pocas formas más comunes en que puedes actuar sobre el recurso:

GET	Recupera el recurso desde el servidor
POST	Crea un recurso en el servidor
PUT	Actualiza el recurso en el servidor
DELETE	Elimina el recurso del servidor

Con esto en mente, te puedes imaginar que una petición *HTTP* podría ser similar a eliminar una entrada de blog específica, por ejemplo:

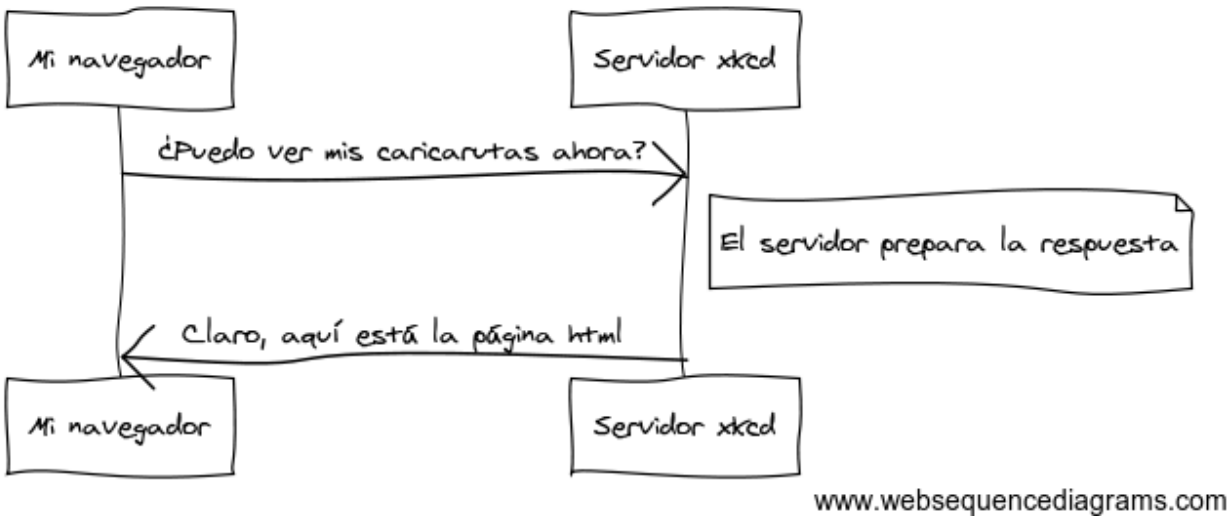
```
DELETE /blog/15 HTTP/1.1
```

Nota: En realidad, hay nueve métodos *HTTP* definidos por la especificación *HTTP*, pero muchos de ellos no se utilizan o apoyan ampliamente. En realidad, muchos navegadores modernos no apoyan los métodos `PUT` y `DELETE`.

Además de la primera línea, una petición *HTTP* invariablemente contiene otras líneas de información conocidas como cabeceras de petición. Las cabeceras pueden suministrar una amplia gama de información como el `Host` solicitado, los formatos de respuesta que acepta el cliente (`Accept`) y la aplicación que utiliza el cliente para realizar la petición (`User-Agent`). Existen muchas otras cabeceras y se pueden encontrar en el artículo Lista de campos de las cabeceras HTTP en la Wikipedia.

Paso 2: El servidor devuelve una respuesta

Una vez que un servidor ha recibido la petición, sabe exactamente qué recursos necesita el cliente (a través de la *URI*) y lo que el cliente quiere hacer con ese recurso (a través del método). Por ejemplo, en el caso de una petición *GET*, el servidor prepara el recurso y lo devuelve en una respuesta *HTTP*. Considera la respuesta del servidor web, xkcd:



Traducida a *HTTP*, la respuesta enviada de vuelta al navegador se verá algo similar a esto:

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html
```

```
<html>
  <!-- HTML para la caricatura xkcd -->
</html>
```

La respuesta *HTTP* contiene el recurso solicitado (contenido *HTML* en este caso), así como otra información acerca de la respuesta. La primera línea es especialmente importante y contiene el código de estado *HTTP* (200 en este caso) de la respuesta. El código de estado comunica el resultado global de la petición devuelto al cliente. ¿Tuvo éxito la petición? ¿Hubo algún error? Existen diferentes códigos de estado que indican éxito, un error o qué más se necesita hacer con el cliente (por ejemplo, redirigirlo a otra página). La lista completa se puede encontrar en el artículo Lista de códigos de estado HTTP en la Wikipedia.

Al igual que la petición, una respuesta *HTTP* contiene datos adicionales conocidos como cabeceras *HTTP*. Por ejemplo, una importante cabecera de la respuesta *HTTP* es `Content-Type`. El cuerpo del mismo recurso se puede devolver en múltiples formatos, incluyendo *HTML*, *XML* o *JSON* por nombrar unos cuantos. La cabecera `Content-Type` indica al cliente en qué formato se está devolviendo.

Existen muchas otras cabeceras, algunas de las cuales son muy poderosas. Por ejemplo, ciertas cabeceras se pueden usar para crear un poderoso sistema de memoria caché.

Peticiones, respuestas y desarrollo Web

Esta conversación petición-respuesta es el proceso fundamental que impulsa toda la comunicación en la web. Y tan importante y poderoso como es este proceso, inevitablemente es simple.

El hecho más importante es el siguiente: independientemente del lenguaje que utilices, el tipo de aplicación que construyas (web, móvil, *API JSON*), o la filosofía de desarrollo que sigas, el objetivo final de una aplicación siempre es **entender** cada petición y crear y devolver la respuesta adecuada.

Symfony está diseñado para adaptarse a esta realidad.

Truco: Para más información acerca de la especificación *HTTP*, lee la referencia original HTTP 1.1 RFC o HTTP Bis, el cual es un esfuerzo activo para aclarar la especificación original. Una gran herramienta para comprobar tanto la petición como las cabeceras de la respuesta mientras navegas es la extensión Cabeceras HTTP en vivo (*Live HTTP Headers*) para Firefox.

2.1.2 Peticiones y respuestas en *PHP*

Entonces ¿cómo interactúas con la "petición" y creas una "respuesta" utilizando *PHP*? En realidad, *PHP* te abstrae un poco de todo el proceso:

```
<?php
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

header('Content-type: text/html');
echo 'La URI solicitada es: '.$uri;
echo 'El valor del parámetro "foo" es: '.$foo;
```

Por extraño que parezca, esta pequeña aplicación, de hecho, está tomando información de la petición *HTTP* y la utiliza para crear una respuesta *HTTP*. En lugar de analizar el mensaje *HTTP* de la petición, *PHP* prepara variables superglobales tales como `$_SERVER` y `$_GET` que contienen toda la información de la petición. Del mismo modo, en lugar de devolver la respuesta *HTTP* con formato de texto, puedes usar la función `header()` para crear las cabeceras de la respuesta y simplemente imprimir el contenido real que será la porción que contiene el mensaje de la respuesta. *PHP* creará una verdadera respuesta *HTTP* y la devolverá al cliente:

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

La URI solicitada es: /probando?foo=symfony
El valor del parámetro "foo" es: symfony
```

2.1.3 Peticiones y respuestas en *Symfony*

Symfony ofrece una alternativa al enfoque de *PHP* a través de dos clases que te permiten interactuar con la petición *HTTP* y la respuesta de una manera más fácil. La clase `Symfony\Component\HttpFoundation\Request` es una sencilla representación orientada a objeto del mensaje de la petición *HTTP*. Con ella, tienes toda la información a tu alcance:

```
use Symfony\Component\HttpFoundation\Request;

$peticion = Request::createFromGlobals();

// la URI solicitada (por ejemplo, /sobre) menos los parámetros de la consulta
$peticion->getPathInfo();

// recupera las variables GET y POST respectivamente
$peticion->query->get('foo');
$peticion->request->get('bar');

// recupera una instancia del archivo subido identificado por foo
$peticion->files->get('foo');

$peticion->getMethod();          // GET, POST, PUT, DELETE, HEAD
$peticion->getLanguages();       // un arreglo de idiomas aceptados por el cliente
```

Como bono adicional, en el fondo la clase `Petición` hace un montón de trabajo del cual nunca tendrás que preocuparte. Por ejemplo, el método `isSecure()` comprueba *tres* diferentes valores en *PHP* que pueden indicar si el usuario está conectado a través de una conexión segura (es decir, `https`).

Symfony también proporciona una clase `Respuesta`: una simple representación *PHP* de un mensaje de respuesta *HTTP*. Esto permite que tu aplicación utilice una interfaz orientada a objetos para construir la respuesta que será devuelta al cliente:

```
use Symfony\Component\HttpFoundation\Response;
$respuesta = new Response();

$respuesta->setContent('<html><body><h1>¡Hola mundo!</h1></body></html>');
$respuesta->setStatusCode(200);
$respuesta->headers->set('Content-Type', 'text/html');

// imprime las cabeceras HTTP seguidas por el contenido
$respuesta->send();
```

Si *Symfony* no ofreciera nada más, ya tendrías un conjunto de herramientas para acceder fácilmente a la información de la petición y una interfaz orientada a objetos para crear la respuesta. Incluso, a medida que aprendas muchas de las poderosas características de *Symfony*, nunca olvides que el objetivo de tu aplicación es *interpretar una petición y crear la respuesta apropiada basada en la lógica de tu aplicación*.

Truco: Las clases `Respuesta` y `Petición` forman parte de un componente independiente incluido en *Symfony*

llamado `HttpFoundation`. Este componente se puede utilizar completamente independiente de *Symfony* y también proporciona clases para manejar sesiones y subir archivos.

2.1.4 El viaje desde la petición hasta la respuesta

Al igual que el mismo *HTTP*, los objetos `Petición` y `Respuesta` son bastante simples. La parte difícil de la construcción de una aplicación es escribir lo que viene en el medio. En otras palabras, el verdadero trabajo viene al escribir el código que interpreta la información de la petición y crea la respuesta.

Tu aplicación probablemente hace muchas cosas, como enviar correo electrónico, manejar los formularios presentados, guardar cosas en una base de datos, reproducir las páginas *HTML* y proteger el contenido con seguridad. ¿Cómo puedes manejar todo esto y todavía mantener tu código organizado y fácil de mantener?

Symfony fue creado para resolver estos problemas para que no tengas que hacerlo personalmente.

El controlador frontal

Tradicionalmente, las aplicaciones eran construidas de modo que cada "página" de un sitio tenía su propio archivo físico:

```
index.php
contacto.php
blog.php
```

Hay varios problemas con este enfoque, incluyendo la falta de flexibilidad de las *URL* (¿qué pasa si quieres cambiar `blog.php` a `noticias.php` sin romper todos los vínculos?) y el hecho de que cada archivo *debe* incluir manualmente un conjunto de archivos básicos para la seguridad, conexiones a base de datos y que el "aspecto" del sitio pueda permanecer constante.

Una solución mucho mejor es usar un *controlador frontal*: un solo archivo *PHP* que se encargue de todas las peticiones que llegan a tu aplicación. Por ejemplo:

`/index.php`	ejecuta `index.php`
`/index.php/contacto`	ejecuta `index.php`
`/index.php/blog`	ejecuta `index.php`

Truco: Usando `mod_rewrite` de Apache (o equivalente con otros servidores web), las direcciones *URL* se pueden limpiar fácilmente hasta ser sólo `/`, `/contacto` y `/blog`.

Ahora, cada petición se maneja exactamente igual. En lugar de direcciones *URL* individuales ejecutando diferentes archivos *PHP*, el controlador frontal *siempre* se ejecuta, y el enrutado de diferentes direcciones *URL* a diferentes partes de tu aplicación se realiza internamente. Esto resuelve los problemas del enfoque original. Casi todas las aplicaciones web modernas lo hacen - incluyendo aplicaciones como WordPress.

Mantente organizado

Pero dentro de tu controlador frontal, ¿cómo sabes qué página debe reproducir y cómo puedes reproducir cada una en forma sana? De una forma u otra, tendrás que comprobar la *URI* entrante y ejecutar diferentes partes de tu código en función de ese valor. Esto se puede poner feo rápidamente:

```
// index.php

$peticion = Request::createFromGlobals();
$ruta = $peticion->getPathInfo(); // la URL solicitada
```

```
if (in_array($ruta, array('', '/')) {
    $respuesta = new Response('Bienvenido a la página inicial.');
} elseif ($ruta == '/contacto') {
    $respuesta = new Response('Contáctanos');
} else {
    $respuesta = new Response('Página no encontrada.', 404);
}
$respuesta->send();
```

La solución a este problema puede ser difícil. Afortunadamente esto es *exactamente* para lo que *Symfony* está diseñado.

El flujo de las aplicaciones *Symfony*

Cuando dejas que *Symfony* controle cada petición, la vida es mucho más fácil. *Symfony* sigue el mismo patrón simple en cada petición:

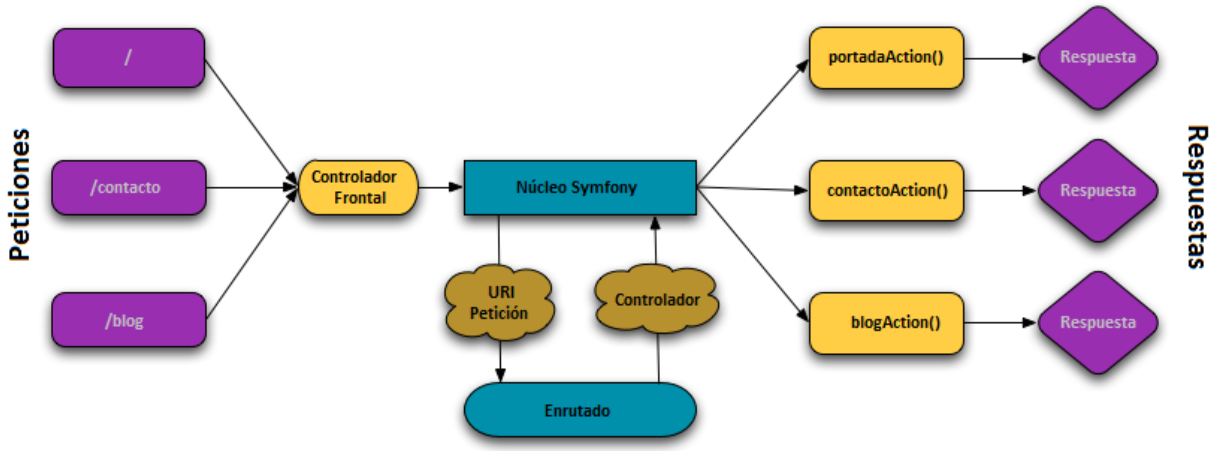


Figura 2.1: Las peticiones entrantes son interpretadas por el enrutador y pasadas a las funciones controladoras que regresan objetos `Respuesta`.

Cada "página" de tu sitio está definida en un archivo de configuración de enrutado que asigna las diferentes direcciones *URL* a diferentes funciones *PHP*. El trabajo de cada función *PHP* conocida como *controlador*, es utilizar la información de la petición - junto con muchas otras herramientas que *Symfony* pone a tu disposición - para crear y devolver un objeto `Respuesta`. En otras palabras, el controlador es donde *está tu* código: ahí es dónde se interpreta la petición y crea una respuesta.

¡Así de fácil! Repasemos:

- Cada petición ejecuta un archivo controlador frontal;
- El sistema de enrutado determina cual función *PHP* se debe ejecutar en base a la información de la petición y la configuración de enrutado que hemos creado;
- La función *PHP* correcta se ejecuta, donde tu código crea y devuelve el objeto `Respuesta` adecuado.

Una petición *Symfony* en acción

Sin bucear demasiado en los detalles, veamos este proceso en acción. Supongamos que deseas agregar una página `/contacto` a tu aplicación *Symfony*. En primer lugar, empezamos agregando una entrada `/contacto` a tu archivo

de configuración de enrutado:

```
contacto:
    pattern:  /contacto
    defaults: { _controller: AcmeDemoBundle:Principal:contacto }
```

Nota: En este ejemplo utilizamos *YAML* (Página 565) para definir la configuración de enrutado. La configuración de enrutado también se puede escribir en otros formatos como *XML* o *PHP*.

Cuando alguien visita la página `/contacto`, esta ruta coincide, y se ejecuta el controlador especificado. Como veremos en el capítulo *Enrutando* (Página 81), La cadena `AcmeDemoBundle:Principal:contacto` es una sintaxis corta que apunta hacia el método *PHP* `contactoAction` dentro de una clase llamada `PrincipalController`:

```
class PrincipalController
{
    public function contactoAction()
    {
        return new Response('<h1>¡Contáctanos!</h1>');
    }
}
```

En este ejemplo muy simple, el controlador simplemente crea un objeto `Respuesta` con el código *HTML* "<h1>¡Contáctanos!</h1>". En el capítulo *Controlador* (Página 71), aprenderás cómo un controlador puede reproducir plantillas, permitiendo que tu código de "presentación" (es decir, algo que en realidad escribe *HTML*) viva en un archivo de plantilla separado. Esto libera al controlador de preocuparse sólo de las cosas difíciles: la interacción con la base de datos, la manipulación de los datos presentados o el envío de mensajes de correo electrónico.

2.1.5 *Symfony2*: Construye tu aplicación, no tus herramientas.

Ahora sabemos que el objetivo de cualquier aplicación es interpretar cada petición entrante y crear una respuesta adecuada. Cuando una aplicación crece, es más difícil mantener organizado tu código y que a la vez sea fácil darle mantenimiento. Invariablemente, las mismas tareas complejas siguen viniendo una y otra vez: la persistencia de cosas en la base de datos, procesamiento y reutilización de plantillas, manejo de formularios presentados, envío de mensajes de correo electrónico, validación de entradas del usuario y administración de la seguridad.

La buena nueva es que ninguno de estos problemas es único. *Symfony* proporciona una plataforma completa, con herramientas que te permiten construir tu aplicación, no tus herramientas. Con *Symfony2*, nada se te impone: eres libre de usar la plataforma *Symfony* completa, o simplemente una pieza de *Symfony* por sí misma.

Herramientas independientes: *Componentes* de *Symfony2*

Entonces, ¿qué *es Symfony2*? En primer lugar, *Symfony2* es una colección de más de veinte bibliotecas independientes que se pueden utilizar dentro de *cualquier* proyecto *PHP*. Estas bibliotecas, llamadas *componentes de Symfony2*, contienen algo útil para casi cualquier situación, independientemente de cómo desarrolles tu proyecto. Para nombrar algunos:

- HttpFoundation - Contiene las clases `Petición` y `Respuesta`, así como otras clases para manejar sesiones y carga de archivos;
- Routing - Potente y rápido sistema de enrutado que te permite asignar una *URI* específica (por ejemplo `/contacto`) a cierta información acerca de cómo dicha petición se debe manejar (por ejemplo, ejecutar el método `contactoAction()`);
- Form - Una completa y flexible plataforma para crear formularios y manipular la presentación de los mismos;

- Validator Un sistema para crear reglas sobre datos y entonces, cuando el usuario presenta los datos comprobar si son válidos o no siguiendo esas reglas;
- ClassLoader Una biblioteca para carga automática que permite utilizar clases *PHP* sin necesidad de `requerir` manualmente los archivos que contienen esas clases;
- Templating Un juego de herramientas para reproducir plantillas, la cual gestiona la herencia de plantillas (es decir, una plantilla está decorada con un diseño) y realiza otras tareas de plantilla comunes;
- Security - Una poderosa biblioteca para manejar todo tipo de seguridad dentro de una aplicación;
- Translation Una plataforma para traducir cadenas en tu aplicación.

Todos y cada uno de estos componentes se desacoplan y se pueden utilizar en *cualquier* proyecto *PHP*, independientemente de si utilizas la plataforma *Symfony2*. Cada parte está hecha para utilizarla si es conveniente y sustituirse cuando sea necesario.

La solución completa: La *plataforma Symfony2*

Entonces, ¿qué *es* la *plataforma Symfony2*? La *plataforma Symfony2* es una biblioteca *PHP* que realiza dos distintas tareas:

1. Proporciona una selección de componentes (es decir, los componentes *Symfony2*) y bibliotecas de terceros (por ejemplo, `SwiftMailer` para enviar mensajes de correo electrónico);
2. Proporciona configuración sensible y un "pegamento" que une la biblioteca con todas estas piezas.

El objetivo de la plataforma es integrar muchas herramientas independientes con el fin de proporcionar una experiencia coherente al desarrollador. Incluso la propia plataforma es un paquete *Symfony2* (es decir, un complemento) que se puede configurar o sustituir completamente.

Symfony2 proporciona un potente conjunto de herramientas para desarrollar aplicaciones web rápidamente sin imponerse en tu aplicación. Los usuarios normales rápidamente pueden comenzar el desarrollo usando una distribución *Symfony2*, que proporciona un esqueleto del proyecto con parámetros predeterminados. Para los usuarios más avanzados, el cielo es el límite.

2.2 *Symfony2* frente a *PHP* simple

¿Por qué Symfony2 es mejor que sólo abrir un archivo y escribir PHP simple?

Si nunca has usado una plataforma *PHP*, no estás familiarizado con la filosofía MVC, o simplemente te preguntas qué es todo ese *alboroto* en torno a *Symfony2*, este capítulo es para ti. En vez de *decirte* que *Symfony2* te permite desarrollar software más rápido y mejor que con *PHP* simple, debes verlo tú mismo.

En este capítulo, vamos a escribir una aplicación sencilla en *PHP* simple, y luego la reconstruiremos para que esté mejor organizada. Podrás viajar a través del tiempo, viendo las decisiones de por qué el desarrollo web ha evolucionado en los últimos años hasta donde está ahora.

Al final, verás cómo *Symfony2* te puede rescatar de las tareas cotidianas y te permite recuperar el control de tu código.

2.2.1 Un sencillo blog en *PHP* simple

En este capítulo, crearemos una simbólica aplicación de blog utilizando sólo *PHP* simple. Para empezar, crea una página que muestra las entradas del blog que se han persistido en la base de datos. Escribirla en *PHP* simple es rápido y sucio:

```
<?php
// index.php

$enlace = mysql_connect('localhost', 'miusuario', 'mipase');
mysql_select_db('bd_blog', $enlace);

$resultado = mysql_query('SELECT id, titulo FROM comunicado', $enlace);
?>

<html>
    <head>
        <title>Lista de comunicados</title>
    </head>
    <body>
        <h1>Lista de comunicados</h1>
        <ul>
            <?php while ($fila = mysql_fetch_assoc($resultado)): ?>
            <li>
                <a href="/show.php?id=<?php echo $fila['id'] ?>">
                    <?php echo $fila['titulo'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>

<?php
mysql_close($enlace);
```

Eso es fácil de escribir, se ejecuta rápido, y, cuando tu aplicación crece, imposible de mantener. Hay varios problemas que es necesario abordar:

- **No hay comprobación de errores**: ¿Qué sucede si falla la conexión a la base de datos?
- **Deficiente organización**: Si la aplicación crece, este único archivo cada vez será más difícil de mantener, hasta que finalmente sea imposible. ¿Dónde se debe colocar el código para manejar un formulario enviado? ¿Cómo se pueden validar los datos? ¿Dónde debe ir el código para enviar mensajes de correo electrónico?
- **Es difícil reutilizar el código**: Ya que todo está en un archivo, no hay manera de volver a utilizar alguna parte de la aplicación en otras "páginas" del blog.

Nota: Otro problema no mencionado aquí es el hecho de que la base de datos está vinculada a *MySQL*. Aunque no se ha tratado aquí, *Symfony2* integra Doctrine plenamente, una biblioteca dedicada a la abstracción y asignación de bases de datos.

Vamos a trabajar en la solución de estos y muchos problemas más.

Aislando la presentación

El código inmediatamente se puede beneficiar de la separación entre la "lógica" de la aplicación y el código que prepara la "presentación" *HTML*:

```
<?php
// index.php

$enlace = mysql_connect('localhost', 'miusuario', 'mipase');
```

```
mysql_select_db('bd_blog', $enlace);

$resultado = mysql_query('SELECT id, titulo FROM comunicado', $enlace);

$comunicados = array();
while ($fila = mysql_fetch_assoc($resultado)) {
    $comunicados[] = $fila;
}

mysql_close($enlace);

// incluye el código HTML de la presentación
require 'plantillas/lista.php';
```

Ahora el código *HTML* está guardado en un archivo separado (`plantillas/lista.php`), el cual principalmente es un archivo *HTML* que utiliza una sintaxis de plantilla tipo *PHP*:

```
<html>
    <head>
        <title>Lista de comunicados</title>
    </head>
    <body>
        <h1>Lista de comunicados</h1>
        <ul>
            <?php foreach ($comunicados as $comunicado): ?>
            <li>
                <a href="/leer?id=<?php echo $comunicado['id'] ?>">
                    <?php echo $comunicado['titulo'] ?>
                </a>
            </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

Por convención, el archivo que contiene toda la lógica de la aplicación - `index.php` - se conoce como *"controlador"*. El término *controlador* es una palabra que se escucha mucho, independientemente del lenguaje o plataforma que utilices. Simplemente se refiere a la zona de *tu código* que procesa la entrada del usuario y prepara la respuesta.

En este caso, nuestro controlador prepara los datos de la base de datos y, luego los incluye en una plantilla para presentarlos. Con el controlador aislado, fácilmente podríamos cambiar *sólo* el archivo de plantilla si es necesario procesar las entradas del blog en algún otro formato (por ejemplo, `lista.json.php` para el formato *JSON*).

Aislando la lógica de la aplicación (el dominio)

Hasta ahora, la aplicación sólo contiene una página. Pero ¿qué pasa si una segunda página necesita utilizar la misma conexión a la base de datos, e incluso la misma matriz de entradas del blog? Reconstruye el código para que el comportamiento de las funciones básicas de acceso a datos de la aplicación esté aislado en un nuevo archivo llamado `modelo.php`:

```
<?php
// modelo.php

function abre_conexion_bd()
{
    $enlace = mysql_connect('localhost', 'miusuario', 'mipase');
    mysql_select_db('bd_blog', $enlace);
```

```
    return $enlace;
}

function cierra_conexion_bd($enlace)
{
    mysql_close($enlace);
}

function obt_comunicados()
{
    $enlace = abre_conexion_bd();

    $resultado = mysql_query('SELECT id, titulo FROM comunicado', $enlace);
    $comunicados = array();
    while ($fila = mysql_fetch_assoc($resultado)) {
        $comunicados[] = $fila;
    }
    cierra_conexion_bd($enlace);

    return $comunicados;
}
```

Truco: Utilizamos el nombre de archivo `modelo.php` debido a que el acceso a la lógica y los datos de una aplicación, tradicionalmente, se conoce como la capa del "modelo". En una aplicación bien organizada, la mayoría del código que representa tu "lógica de negocio" debe vivir en el modelo (en lugar de vivir en un controlador). Y, a diferencia de este ejemplo, sólo una parte (o ninguna) del modelo realmente está interesada en acceder a la base de datos.

El controlador (`index.php`) ahora es muy sencillo:

```
<?php
require_once 'modelo.php';

$comunicados = obt_comunicados();

require 'plantillas/lista.php';
```

Ahora, la única tarea del controlador es conseguir los datos de la capa del modelo de la aplicación (el modelo) e invocar a una plantilla que reproduce los datos. Este es un ejemplo muy simple del patrón modelo-vista-controlador.

Aislando el diseño

En este punto, hemos reconstruido la aplicación en tres piezas distintas, mismas que nos ofrecen varias ventajas y la oportunidad de volver a utilizar casi todo en diferentes páginas.

La única parte del código que *no* se puede reutilizar es el diseño de la página. Corregiremos esto creando un nuevo archivo `boceto.php`:

```
<!-- plantillas/boceto.php -->
<html>
    <head>
        <title><?php echo $titulo ?></title>
    </head>
    <body>
        <?php echo $contenido ?>
```

```
    </body>
</html>
```

La plantilla (`plantillas/lista.php`) ahora se puede simplificar para "extender" el diseño:

```
<?php $titulo = 'Lista de comunicados' ?>

<?php ob_start() ?>
    <h1>Lista de comunicados</h1>
    <ul>
        <?php foreach ($comunicados as $comunicado): ?>
        <li>
            <a href="/leer?id=<?php echo $comunicado['id'] ?>">
                <?php echo $comunicado['titulo'] ?>
            </a>
        </li>
        <?php endforeach; ?>
    </ul>
<?php $contenido = ob_get_clean() ?>

<?php include 'boceto.php' ?>
```

Ahora hemos introducido una metodología que nos permite reutilizar el diseño. Desafortunadamente, para lograrlo, estamos obligados a utilizar algunas desagradables funciones de *PHP* (`ob_start()`, `ob_get_clean()`) en la plantilla. *Symfony2* utiliza un componente `Templating` que nos permite realizar esto limpia y fácilmente. En breve lo verás en acción.

2.2.2 Agregando una página "show" blog

La página "lista" del blog se ha rediseñado para que el código esté mejor organizado y sea reutilizable. Para probarlo, añade una página "show" al blog, que muestre una entrada individual del blog identificada por un parámetro de consulta `id`.

Para empezar, crea una nueva función en el archivo `modelo.php` que recupere un resultado individual del blog basándose en un identificador dado:

```
// modelo.php
function obt_comunicado_por_id($id)
{
    $enlace = abre_conexion_bd();

    $id = mysql_real_escape_string($id);
    $consulta = 'SELECT fecha, titulo, mensaje FROM comunicado WHERE id = '.$id;
    $resultado = mysql_query($consulta);
    $fila = mysql_fetch_assoc($resultado);

    cierra_conexion_bd($enlace);

    return $fila;
}
```

A continuación, crea un nuevo archivo llamado `show.php` - el controlador para esta nueva página:

```
<?php
require_once 'modelo.php';

$comunicado = obt_comunicado_por_id($_GET['id']);
```

```
require 'plantillas/show.php';
```

Por último, crea el nuevo archivo de plantilla - `plantillas/show.php` - para reproducir una entrada individual del blog:

```
<?php $titulo = $comunicado['titulo'] ?>

<?php ob_start() ?>
    <h1><?php echo $comunicado['titulo'] ?></h1>

    <div class="date"><?php echo $comunicado['date'] ?></div>
    <div class="body">
        <?php echo $comunicado['body'] ?>
    </div>
<?php $contenido = ob_get_clean() ?>

<?php include 'boceto.php' ?>
```

Ahora, es muy fácil crear la segunda página y sin duplicar código. Sin embargo, esta página introduce problemas aún más perniciosos que una plataforma puede resolver por ti. Por ejemplo, un parámetro `id` ilegal u omitido en la consulta hará que la página se bloquee. Sería mejor si esto reprodujera una página 404, pero sin embargo, en realidad esto no se puede hacer fácilmente. Peor aún, si olvidaras desinfectar el parámetro `id` por medio de la función `mysql_real_escape_string()`, tu base de datos estaría en riesgo de un ataque de inyección `SQL`.

Otro importante problema es que cada archivo de controlador individual debe incluir al archivo `modelo.php`. ¿Qué pasaría si cada archivo de controlador de repente tuviera que incluir un archivo adicional o realizar alguna tarea global (por ejemplo, reforzar la seguridad)? Tal como está ahora, el código tendría que incluir todos los archivos de los controladores. Si olvidas incluir algo en un solo archivo, esperemos que no sea alguno relacionado con la seguridad...

2.2.3 El "controlador frontal" al rescate

La solución es utilizar un *controlador frontal*: un único archivo *PHP* a través del cual se procesan *todas* las peticiones. Con un controlador frontal, la *URI* de la aplicación cambia un poco, pero se vuelve más flexible:

```
Sin controlador frontal
/index.php          => (ejecuta index.php) la página lista de mensajes.
/show.php           => (ejecuta show.php)  la página muestra un mensaje particular.

Con index.php como controlador frontal
/index.php          => (ejecuta index.php) la página lista de mensajes.
/index.php/show     => (ejecuta index.php) la página muestra un mensaje particular.
```

Truco: Puedes quitar la porción `index.php` de la *URI* si utilizas las reglas de reescritura de Apache (o equivalentes). En ese caso, la *URI* resultante de la página show del blog simplemente sería `/show`.

Cuando se usa un controlador frontal, un solo archivo *PHP* (`index.php` en este caso) procesa todas las peticiones. Para la página 'show' del blog, `/index.php/show` realmente ejecuta el archivo `index.php`, que ahora es el responsable de dirigir internamente las peticiones basándose en la *URI* completa. Como puedes ver, un controlador frontal es una herramienta muy poderosa.

Creando el controlador frontal

Estás a punto de dar un **gran** paso en la aplicación. Con un archivo manejando todas las peticiones, puedes centralizar cosas tales como el manejo de la seguridad, la carga de configuración y enrutado. En esta aplicación, `index.php`

ahora debe ser lo suficientemente inteligente como para reproducir la lista de entradas del blog *o* mostrar la página de una entrada particular basándose en la *URI* solicitada:

```
<?php
// index.php

// carga e inicia algunas bibliotecas globales
require_once 'modelo.php';
require_once 'controladores.php';

// encamina la petición internamente
$uri = $_SERVER['REQUEST_URI'];
if ($uri == '/index.php') {
    lista_action();
} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Página no encontrada</h1></body></html>';
}
```

Por organización, ambos controladores (antes `index.php` y `show.php`) son funciones *PHP* y cada una se ha movido a un archivo separado, `controladores.php`:

```
function lista_action()
{
    $comunicados = obt_comunicados();
    require 'plantillas/lista.php';
}

function show_action($id)
{
    $comunicado = obt_comunicado_por_id($id);
    require 'plantillas/show.php';
}
```

Como controlador frontal, `index.php` ha asumido un papel completamente nuevo, el cual incluye la carga de las bibliotecas del núcleo y enrutar la aplicación para invocar a uno de los dos controladores (las funciones `lista_action()` y `show_action()`). En realidad, el controlador frontal está empezando a verse y actuar como el mecanismo *Symfony2* para la manipulación y enrutado de peticiones.

Truco: Otra ventaja del controlador frontal es la flexibilidad de las direcciones *URL*. Ten en cuenta que la *URL* a la página 'show' del blog se puede cambiar de `/show` a `/leer` cambiando el código solamente en una única ubicación. Antes, era necesario cambiar todo un archivo para cambiar el nombre. En *Symfony2*, incluso las direcciones *URL* son más flexibles.

Por ahora, la aplicación ha evolucionado de un único archivo *PHP*, a una estructura organizada y permite la reutilización de código. Debes estar feliz, pero aún lejos de estar satisfecho. Por ejemplo, el sistema de "enrutado" es voluble, y no reconoce que la página 'lista' (`/index.php`) también debe ser accesible a través de `/` (si se han agregado las reglas de reescritura de Apache). Además, en lugar de desarrollar el blog, una gran cantidad del tiempo se ha gastado trabajando en la "arquitectura" del código (por ejemplo, el enrutado, invocando controladores, plantillas, etc.) Se tendrá que gastar más tiempo para manejar la presentación de formularios, validación de entradas, registro de sucesos y seguridad. ¿Por qué tienes que reinventar soluciones a todos estos problemas rutinarios?

Añadiendo un toque *Symfony2*

Symfony2 al rescate. Antes de utilizar *Symfony2* realmente, debes asegurarte de que *PHP* sabe cómo encontrar las clases *Symfony2*. Esto se logra a través de un cargador automático que proporciona *Symfony*. Un cargador automático es una herramienta que permite empezar a utilizar clases *PHP* sin incluir explícitamente el archivo que contiene la clase.

Primero, descarga symfony y colócalo en el directorio `vendor/symfony/`. A continuación, crea un archivo `app/bootstrap.php`. Se usa para `requerir` los dos archivos en la aplicación y para configurar el cargador automático:

```
<?php
// bootstrap.php
require_once 'modelo.php';
require_once 'controladores.php';
require_once 'vendor/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

$loader = new Symfony\Component\ClassLoader\UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/vendor/symfony/src',
));

$loader->register();
```

Esto le dice al cargador automático donde están las clases de *Symfony*. Con esto, puedes comenzar a utilizar las clases de *Symfony* sin necesidad de utilizar la declaración `requiere` en los archivos que las utilizan.

La esencia de la filosofía *Symfony* es la idea de que el trabajo principal de una aplicación es interpretar cada petición y devolver una respuesta. Con este fin, *Symfony2* proporciona ambas clases `Symfony\Component\HttpFoundation\Request` y `Symfony\Component\HttpFoundation\Response`. Estas clases son representaciones orientadas a objetos de la petición *HTTP* que se está procesando y la respuesta *HTTP* que devolverá. Úsalas para mejorar el blog:

```
<?php
// index.php
require_once 'app/bootstrap.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$peticion = Request::createFromGlobals();

$uri = $peticion->getPathInfo();
if ($uri == '/') {
    $respuesta = lista_action();
} elseif ($uri == '/show' && $peticion->query->has('id')) {
    $respuesta = show_action($peticion->query->get('id'));
} else {
    $html = '<html><body><h1>Página no encontrada</h1></body></html>';
    $respuesta = new Response($html, 404);
}

// difunde las cabeceras y envía la respuesta
$respuesta->send();
```

Los controladores son responsables de devolver un objeto `Respuesta`. Para facilitarnos esto, puedes agregar una nueva función `reproduce_plantilla()`, la cual, por cierto, actúa un poco como el motor de plantillas de *Symfony2*:

```
// controladores.php
use Symfony\Component\HttpFoundation\Response;

function lista_action()
{
    $comunicados = obt_comunicados();
    $html = reproduce_plantilla('plantillas/lista.php', array('comunicados' => $comunicados));

    return new Response($html);
}

function show_action($id)
{
    $comunicado = obt_comunicado_por_id($id);
    $html = reproduce_plantilla('plantillas/show.php', array('post' => $comunicado));

    return new Response($html);
}

// función ayudante para reproducir plantillas
function reproduce_plantilla($ruta, array $args)
{
    extract($args);
    ob_start();
    require $ruta;
    $html = ob_get_clean();

    return $html;
}
```

Al reunir una pequeña parte de *Symfony2*, la aplicación es más flexible y fiable. La `Petición` proporciona una manera confiable para acceder a información de la petición *HTTP*. Especialmente, el método `getPathInfo()` devuelve una *URI* limpia (siempre devolviendo `/show` y nunca `/index.php/show`). Por lo tanto, incluso si el usuario va a `/index.php/show`, la aplicación es lo suficientemente inteligente para encaminar la petición hacia `show_action()`.

El objeto `Respuesta` proporciona flexibilidad al construir la respuesta *HTTP*, permitiendo que las cabeceras *HTTP* y el contenido se agreguen a través de una interfaz orientada a objetos. Y aunque las respuestas en esta aplicación son simples, esta flexibilidad pagará dividendos en cuanto tu aplicación crezca.

Aplicación de ejemplo en *Symfony2*

El blog ha *avanzado*, pero todavía contiene una gran cantidad de código para una aplicación tan simple. De paso, también inventamos un sencillo sistema de enrutado y un método que utiliza `ob_start()` y `ob_get_clean()` para procesar plantillas. Si, por alguna razón, necesitas continuar la construcción de esta "plataforma" desde cero, por lo menos puedes usar los componentes independientes de *Symfony* Routing y Templating, que resuelven estos problemas.

En lugar de resolver problemas comunes de nuevo, puedes dejar que *Symfony2* se preocupe de ellos por ti. Aquí está la misma aplicación de ejemplo, ahora construida en *Symfony2*:

```
<?php
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
class BlogController extends Controller
{
    public function listAction()
    {
        $comunicados = $this->get('doctrine')->getEntityManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Comunicado p')
            ->execute();

        return $this->render('AcmeBlogBundle:Comunicado:lista.html.php', array('comunicados' => $comu
    }

    public function showAction($id)
    {
        $comunicado = $this->get('doctrine')
            ->getEntityManager()
            ->getRepository('AcmeBlogBundle:Comunicado')
            ->find($id);

        if (!$comunicado) {
            // provoca que se muestre el error 404 Página no encontrada
            throw $this->createNotFoundException();
        }

        return $this->render('AcmeBlogBundle:Comunicado:show.html.php', array('comunicado' => $comuni
    }
}
```

Los dos controladores siguen siendo ligeros. Cada uno utiliza la biblioteca *ORM* de *Doctrine* para recuperar objetos de la base de datos y el componente `Templating` para reproducir una plantilla y devolver un objeto `Respuesta`. La plantilla lista ahora es un poco más simple:

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/lista.html.php -->
<?php $view->extend('::base.html.php') ?>

<?php $view['slots']->set('titulo', 'Lista de comunicados') ?>

<h1>Lista de comunicados</h1>
<ul>
    <?php foreach ($comunicados as $comunicado): ?>
    <li>
        <a href="<?php echo $view['router']->generate('blog_show', array('id' => $comunicado->getId()
            <?php echo $comunicado->getTitulo() ?>
        </a>
    </li>
    <?php endforeach; ?>
</ul>
```

El diseño es casi idéntico:

```
<!-- app/Resources/views/base.html.php -->
<html>
    <head>
        <title><?php echo $view['slots']->salida('titulo', 'Título predeterminado') ?></title>
    </head>
    <body>
        <?php echo $view['slots']->salida('_content') ?>
    </body>
</html>
```

Nota: Te vamos a dejar como ejercicio la plantilla 'show', porque debería ser trivial crearla basándote en la plantilla lista.

Cuando arranca el motor *Symfony2* (llamado `kernel`), necesita un mapa para saber qué controladores ejecutar basándose en la información solicitada. Un mapa de configuración de enrutado proporciona esta información en formato legible:

```
# app/config/routing.yml
blog_list:
    pattern:  /blog
    defaults: { _controller: AcmeBlogBundle:Blog:lista }

blog_show:
    pattern:  /blog/show/{id}
    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

Ahora que *Symfony2* se encarga de todas las tareas rutinarias, el controlador frontal es muy simple. Y ya que hace tan poco, nunca tienes que volver a tocarlo una vez creado (y si utilizas una distribución *Symfony2*, ¡ni siquiera tendrás que crearlo!):

```
<?php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$nucleo = new AppKernel('prod', false);
$nucleo->handle(Request::createFromGlobals())->send();
```

El único trabajo del controlador frontal es iniciar el motor de *Symfony2* (el `núcleo`) y pasarle un objeto `Petición` para que lo manipule. El núcleo de *Symfony2* entonces utiliza el mapa de enrutado para determinar qué controlador invocar. Al igual que antes, el método controlador es el responsable de devolver el objeto `Respuesta` final. Realmente no hay mucho más sobre él.

Para conseguir una representación visual de cómo maneja *Symfony2* cada petición, consulta el *diagrama de flujo de la petición* (Página 39).

Qué más ofrece *Symfony2*

En los siguientes capítulos, aprenderás más acerca de cómo funciona cada pieza de *Symfony* y la organización recomendada de un proyecto. Por ahora, vamos a ver cómo, migrar el blog de *PHP* simple a *Symfony2* nos ha mejorado la vida:

- Tu aplicación cuenta con **código claro y organizado consistentemente** (aunque *Symfony* no te obliga a ello). Promueve la **reutilización** y permite a los nuevos desarrolladores ser productivos en el proyecto con mayor rapidez.
- 100 % del código que escribes es para *tu* aplicación. **No necesitas desarrollar o mantener servicios públicos de bajo nivel** tales como la *carga automática* (Página 64) de clases, el *enrutado* (Página 81) o la reproducción de *controladores* (Página 71).
- *Symfony2* te proporciona **acceso a herramientas de código abierto** tales como *Doctrine*, plantillas, seguridad, formularios, validación y traducción (por nombrar algunas).
- La aplicación ahora disfruta de **direcciones URL totalmente flexibles** gracias al componente `Routing`.

- La arquitectura centrada en *HTTP* de *Symfony2* te da acceso a poderosas herramientas, tal como la **memoria caché HTTP** impulsadas por la **caché HTTP interna de Symfony2** o herramientas más poderosas, tales como Varnish. Esto se trata posteriormente en el capítulo todo sobre *caché* (Página 210).

Y lo mejor de todo, utilizando *Symfony2*, ¡ahora tienes acceso a un conjunto de herramientas de **alta calidad de código abierto desarrolladas por la comunidad Symfony2**! Para más información, visita Symfony2Bundles.org

2.2.4 Mejores plantillas

Si decides utilizarlo, *Symfony2* de serie viene con un motor de plantillas llamado Twig el cual hace que las plantillas se escriban más rápido y sean más fáciles de leer. Esto significa que, incluso, ¡la aplicación de ejemplo podría contener mucho menos código! Tomemos, por ejemplo, la plantilla lista escrita en *Twig*:

```
{# src/Acme/BlogBundle/Resources/views/Blog/lista.html.twig #}

{% extends "::base.html.twig" %}
{% block titulo %}Lista de comunicados{% endblock %}

{% block cuerpo %}
    <h1>Lista de comunicados</h1>
    <ul>
        {% for comunicado in comunicados %}
        <li>
            <a href="{{ path('blog_show', { 'id': comunicado.id }) }}">
                {{ comunicado.titulo }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

También es fácil escribir la plantilla `base.html.twig` correspondiente:

```
{# app/Resources/views/base.html.twig #}

<html>
    <head>
        <title>{% block titulo %}Título predeterminado{% endblock %}</title>
    </head>
    <body>
        {% block cuerpo %}{% endblock %}
    </body>
</html>
```

Twig es compatible con *Symfony2*. Y si bien las plantillas *PHP* siempre contarán con el apoyo de *Symfony2*, vamos a seguir explicando muchas de las ventajas de *Twig*. Para más información, consulta el capítulo *Plantillas* (Página 98).

2.2.5 Aprende más en el recetario

- *Cómo usar plantillas PHP en lugar de Twig* (Página 403)
- *Cómo definir controladores como servicios* (Página 278)

2.3 Instalando y configurando *Symfony*

El objetivo de este capítulo es el de empezar a trabajar con una aplicación funcionando incorporada en lo alto de *Symfony*. Afortunadamente, *Symfony* dispone de "distribuciones", que son proyectos *Symfony* funcionales desde el "arranque", los cuales puedes descargar y comenzar a desarrollar inmediatamente.

Truco: Si estás buscando instrucciones sobre la mejor manera de crear un nuevo proyecto y guardarlo vía el control de código fuente, consulta Usando control de código fuente (Página 56).

2.3.1 Descargando una distribución de *Symfony2*

Truco: En primer lugar, comprueba que tienes instalado y configurado un servidor web (como Apache) con *PHP 5.3.2* o superior. Para más información sobre los requisitos de *Symfony2*, consulta los *requisitos en la referencia* (Página 570).

Los paquetes de las "distribuciones" de *Symfony2*, son aplicaciones totalmente funcionales que incluyen las bibliotecas del núcleo de *Symfony2*, una selección de útiles paquetes, una sensible estructura de directorios y alguna configuración predeterminada. Al descargar una distribución *Symfony2*, estás descargando el esqueleto de una aplicación funcional que puedes utilizar inmediatamente para comenzar a desarrollar tu aplicación.

Empieza por visitar la página de descarga de *Symfony2* en http://symfony.com/download. En esta página, puedes encontrar la *Edición estándar de Symfony*, que es la distribución principal de *Symfony2*. En este caso, necesitas hacer dos elecciones:

- Descargar o bien un archivo `.tgz` o `.zip` - ambos son equivalentes, descarga aquel con el que te sientas más cómodo;
- Descarga la distribución con o sin `vendors`. Si tienes instalado Git en tu ordenador, debes descargar *Symfony2* "sin `vendors`", debido a que esto añade un poco más de flexibilidad cuando incluyas bibliotecas de terceros.

Descarga uno de los archivos en algún lugar bajo el directorio raíz de tu servidor web local y descomprímelo. Desde una línea de ordenes de UNIX, esto se puede hacer con una de las siguientes ordenes (sustituye `###` con el nombre de archivo real):

```
# para un archivo .tgz
tar zxvf Symfony_Standard_Vendors_2.0.###.tgz

# para un archivo .zip
unzip Symfony_Standard_Vendors_2.0.###.zip
```

Cuando hayas terminado, debes tener un directorio `Symfony/` que se ve algo como esto:

```
www/ <- tu directorio raíz del servidor web
    Symfony/ <- el archivo extraído
        app/
            cache/
            config/
            logs/
        src/
            ...
        vendor/
            ...
        web/
```

```
app.php
...
```

Actualizando `vendors`

Por último, si descargaste el archivo "sin `vendors`", instala tus proveedores ejecutando el siguiente método desde la línea de ordenes:

```
php bin/vendors install
```

Esta orden descarga todas las bibliotecas de terceros necesarias - incluyendo al mismo *Symfony* - en el directorio `vendor/`. Para más información acerca de cómo se manejan las bibliotecas de terceros dentro de *Symfony2*, consulta "*Gestionando bibliotecas de proveedores con bin/vendors y deps* (Página 276)".

Instalando y configurando

En este punto, todas las bibliotecas de terceros necesarias ahora viven en el directorio `vendor/`. También tienes una instalación predeterminada de la aplicación en `app/` y algunos ejemplos de código dentro de `src/`.

Symfony2 viene con una interfaz visual para probar la configuración del servidor, muy útil para ayudarte a solucionar problemas relacionados con la configuración de tu servidor web y *PHP* para utilizar *Symfony*. Usa la siguiente *URL* para examinar tu configuración:

```
http://localhost/Symfony/web/config.php
```

Si hay algún problema, corrígelo antes de continuar.

Configurando permisos

Un problema común es que ambos directorios `app/cache` y `app/logs` deben tener permiso de escritura, tanto para el servidor web cómo para la línea de ordenes del usuario. En un sistema UNIX, si el usuario del servidor web es diferente de tu usuario de línea de ordenes, puedes ejecutar las siguientes ordenes una sola vez en el proyecto para garantizar que los permisos se configuran correctamente. Cambia `www-data` al usuario del servidor web y `tuNombre` a tu usuario de la línea de ordenes:

1. Usando ACL en un sistema que admite chmod +a

Muchos sistemas te permiten utilizar la orden chmod +a. Intenta esto primero, y si se produce un error - prueba el método siguiente:

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

sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
sudo chmod +a "tuNombre allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
```

2. Usando ACL en un sistema que no es compatible con un chmod +a

Algunos sistemas, no son compatibles con `chmod +a`, pero son compatibles con otra utilidad llamada `setfacl`. Posiblemente tengas que habilitar la compatibilidad con ACL en tu partición e instalas setfacl antes de usarlo (como es el caso de Ubuntu), así:

```
sudo setfacl -R -m u:www-data:rwx -m u:tuNombre:rwx app/cache app/logs
sudo setfacl -dR -m u:www-data:rwx -m u:tuNombre:rwx app/cache app/logs
```

3. Sin usar ACL

Si no tienes acceso para modificar los directorios *ACL*, tendrás que cambiar la `umask` para que los directorios `cache/` y `logs/` se puedan escribir por el grupo o por cualquiera (dependiendo de si el usuario del servidor web y el usuario de la línea de ordenes están en el mismo grupo o no). Para ello, pon la siguiente línea al comienzo de los archivos `app/console`, `web/app.php` y `web/app_dev.php`:

```
umask(0002); // Esto permitirá que los permisos sean 0775

// o

umask(0000); // Esto permitirá que los permisos sean 0777
```

Ten en cuenta que el uso de *ACL* se recomienda cuando tienes acceso a ellos en el servidor porque cambiar la `umask` no es seguro en subprocesos.

Cuando todo esté listo, haz clic en el enlace "Visita la página de Bienvenida" para ver tu primer aplicación "real" en *Symfony2*:

```
http://localhost/Symfony/web/app_dev.php/
```

¡*Symfony2* debería felicitarte por tu arduo trabajo hasta el momento!

2.3.2 Empezando a desarrollar

Ahora que tienes una aplicación *Symfony2* completamente funcional, ¡puedes comenzar el desarrollo! Tu distribución puede contener algún código de ejemplo - comprueba el archivo `README.rst` incluido con la distribución (ábrelo como un archivo de texto) para saber qué código de ejemplo incluye tu distribución y cómo lo puedes eliminar más tarde.

Si eres nuevo en *Symfony*, alcánzanos en "*Creando páginas en Symfony2* (Página 57)", donde aprenderás a crear páginas, cambiar la configuración, y todo lo demás que necesitas en tu nueva aplicación.

2.3.3 Usando control de código fuente

Si estás utilizando un sistema de control de versiones como `Git` o `Subversion`, puedes configurar tu sistema de control de versiones y empezar a consignar el proyecto de manera normal. La *Edición estándar de Symfony* **es** el punto de partida para tu nuevo proyecto.

Para instrucciones específicas sobre la mejor manera de configurar el proyecto para almacenarlo en *git*, consulta *Cómo crear y guardar un proyecto Symfony2 en git* (Página 275).

Ignorando el directorio `vendor/`

Si has descargado el archivo *sin proveedores*, puedes omitir todo el directorio `vendor/` y no consignarlo al control de código fuente. Con `Git`, esto se logra creando un archivo `.gitignore` y añadiendo lo siguiente:

```
vendor/
```

Ahora, el directorio de proveedores no será consignado al control de código fuente. Esto está muy bien (en realidad, ¡es genial!) porque cuando alguien más clone o coteje el proyecto, él/ella simplemente puede ejecutar el archivo `php bin/vendors.php` para descargar todas las bibliotecas de proveedores necesarias.

2.4 Creando páginas en *Symfony2*

Crear una nueva página en *Symfony2* es un sencillo proceso de dos pasos:

- *Crea una ruta*: Una ruta define la *URL* de tu página (por ejemplo `/sobre`) y especifica un controlador (el cual es una función *PHP*) que *Symfony2* debe ejecutar cuando la *URL* de una petición entrante coincida con el patrón de la ruta;
- *Crea un controlador*: Un controlador es una función *PHP* que toma la petición entrante y la transforma en el objeto `Respuesta` de *Symfony2* que es devuelto al usuario.

Nos encanta este enfoque simple porque coincide con la forma en que funciona la Web. Cada interacción en la Web se inicia con una petición *HTTP*. El trabajo de la aplicación simplemente es interpretar la petición y devolver la respuesta *HTTP* adecuada.

Symfony2 sigue esta filosofía y te proporciona las herramientas y convenios para mantener organizada tu aplicación a medida que crece en usuarios y complejidad.

¿Suena bastante simple? ¡Démonos una zambullida!

2.4.1 La página "¡Hola *Symfony*!"

Vamos a empezar con una aplicación derivada del clásico "¡Hola Mundo!". Cuando hayamos terminado, el usuario podrá recibir un saludo personal (por ejemplo, "Hola *Symfony*") al ir a la siguiente *URL*:

```
http://localhost/app_dev.php/hola/Symfony
```

En realidad, serás capaz de sustituir `Symfony` con cualquier otro nombre al cual darle la bienvenida. Para crear la página, sigue el simple proceso de dos pasos.

Nota: La guía asume que ya has descargado *Symfony2* y configurado tu servidor web. En la *URL* anterior se supone que `localhost` apunta al directorio `web`, de tu nuevo proyecto *Symfony2*. Para información más detallada sobre este proceso, consulta *Instalando Symfony2* (Página 53).

Antes de empezar: Crea el paquete

Antes de empezar, tendrás que crear un `bundle` (*paquete* en adelante). En *Symfony2*, un *paquete* es como un complemento (o plugin, para los puristas), salvo que todo el código de tu aplicación debe vivir dentro de un paquete.

Un paquete no es más que un directorio que alberga todo lo relacionado con una función específica, incluyendo clases *PHP*, configuración, e incluso hojas de estilo y archivos de *Javascript* (consulta *El sistema de paquetes* (Página 64)).

Para crear un paquete llamado `AcmeHolaBundle` (el paquete de ejemplo que vamos a construir en este capítulo), ejecuta la siguiente orden y sigue las instrucciones en pantalla (usa todas las opciones predeterminadas):

```
php app/console generate:bundle --namespace=Acme/HolaBundle --format=yml
```

Detrás del escenario, se crea un directorio para el paquete en `src/Acme/HolaBundle`. Además agrega automáticamente una línea al archivo `app/AppKernel.php` para registrar el paquete en el núcleo:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Acme\HolaBundle\AcmeHolaBundle(),
    );
    // ...

    return $bundles;
}
```

Ahora que ya está configurado el paquete, puedes comenzar a construir tu aplicación dentro del paquete.

Paso 1: Creando la ruta

De manera predeterminada, el archivo de configuración de enrutado en una aplicación *Symfony2* se encuentra en `app/config/routing.yml`. Al igual que toda la configuración en *Symfony2*, fuera de la caja también puedes optar por utilizar *XML* o *PHP* para configurar tus rutas.

Si te fijas en el archivo de enrutado principal, verás que *Symfony* ya ha agregado una entrada al generar el `AcmeHolaBundle`:

- *YAML*

  ```
  # app/config/routing.yml
  AcmeHolaBundle:
      resource: "@AcmeHolaBundle/Resources/config/routing.yml"
      prefix:   /
  ```

- *XML*

  ```
  <!-- app/config/routing.xml -->
  <?xml version="1.0" encoding="UTF-8" ?>

  <routes xmlns="http://symfony.com/schema/routing"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

      <import resource="@AcmeHolaBundle/Resources/config/routing.xml" prefix="/" />
  </routes>
  ```

- *PHP*

  ```
  // app/config/routing.php
  use Symfony\Component\Routing\RouteCollection;
  use Symfony\Component\Routing\Route;

  $coleccion = new RouteCollection();
  $coleccion->addCollection(
      $cargador->import('@AcmeHolaBundle/Resources/config/routing.php'),
      '/',
  );

  return $coleccion;
  ```

Esta entrada es bastante básica: le dice a *Symfony* que cargue la configuración de enrutado del archivo `Resources/config/routing.yml` que reside en el interior del `AcmeHolaBundle`. Esto significa que co-

locas la configuración de enrutado directamente en `app/config/routing.yml` u organizas tus rutas a través de tu aplicación, y las importas desde aquí.

Ahora que el archivo `routing.yml` es importado desde el paquete, añade la nueva ruta que define la *URL* de la página que estás a punto de crear:

- *YAML*

```yaml
# src/Acme/HolaBundle/Resources/config/routing.yml
hola:
    pattern:  /hola/{nombre}
    defaults: { _controller: AcmeHolaBundle:Hola:index }
```

- *XML*

```xml
<!-- src/Acme/HolaBundle/Resources/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="hola" pattern="/hola/{nombre}">
        <default key="_controller">AcmeHolaBundle:Hola:index</default>
    </route>
</routes>
```

- *PHP*

```php
// src/Acme/HolaBundle/Resources/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('hola', new Route('/hola/{nombre}', array(
    '_controller' => 'AcmeHolaBundle:Hola:index',
)));

return $coleccion;
```

La ruta se compone de dos piezas básicas: el `patrón`, que es la *URL* con la que esta ruta debe coincidir, y un arreglo `defaults`, que especifica el controlador que se debe ejecutar. La sintaxis del marcador de posición en el patrón (`{nombre}`) es un comodín. Significa que `/hola/Ryan`, `/hola/Fabien` o cualquier otra URI similar coincidirá con esta ruta. El parámetro marcador de posición `{nombre}` también se pasará al controlador, de manera que podemos utilizar su valor para saludar personalmente al usuario.

Nota: El sistema de enrutado tiene muchas más características para crear estructuras *URI* flexibles y potentes en tu aplicación. Para más detalles, consulta el capítulo *Enrutando* (Página 81).

Paso 2: Creando el controlador

Cuando una *URL* como `/hola/Ryan` es manejada por la aplicación, la ruta `hola` corresponde con el controlador `AcmeHolaBundle:Hola:index` el cual es ejecutado por la plataforma. El segundo paso del proceso de creación de páginas es precisamente la creación de ese controlador.

El controlador - `AcmeHolaBundle:Hola:index` es el *nombre lógico* del controlador, el cual se asigna al método `indexAction` de una clase *PHP* llamada `Acme\HolaBundle\Controller\Hola`. Empieza creando este archivo dentro de tu `AcmeHolaBundle`:

```
// src/Acme/HolaBundle/Controller/HolaController.php
namespace Acme\HolaBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class HolaController
{
}
```

En realidad, el controlador no es más que un método *PHP* que tú creas y *Symfony* ejecuta. Aquí es donde el código utiliza la información de la petición para construir y preparar el recurso solicitado. Salvo en algunos casos avanzados, el producto final de un controlador siempre es el mismo: un objeto `Respuesta` de *Symfony2*.

Crea el método `indexAction` que *Symfony* ejecutará cuando concuerde la ruta `hola`:

```
// src/Acme/HolaBundle/Controller/HolaController.php

// ...
class HolaController
{
    public function indexAction($nombre)
    {
        return new Response('<html><body>Hola '.$nombre.'!</body></html>');
    }
}
```

El controlador es simple: este crea un nuevo objeto `Respuesta`, cuyo primer argumento es el contenido que se debe utilizar para la respuesta (una pequeña página *HTML* en este ejemplo).

¡Enhorabuena! Después de crear solamente una ruta y un controlador ¡ya tienes una página completamente funcional! Si todo lo has configurado correctamente, la aplicación debe darte la bienvenida:

```
http://localhost/app_dev.php/hola/Ryan
```

Truco: También puedes ver tu aplicación en el *entorno* (Página 69) "prod" visitando:

```
http://localhost/app.php/hola/Ryan
```

Si se produce un error, probablemente sea porque necesitas vaciar la caché ejecutando:

```
php app/console cache:clear --env=prod --no-debug
```

Un opcional, pero frecuente, tercer paso en el proceso es crear una plantilla.

Nota: Los controladores son el punto de entrada principal a tu código y un ingrediente clave en la creación de páginas. Puedes encontrar mucho más información en el capítulo *Controlador* (Página 71).

Paso opcional 3: Creando la plantilla

Las plantillas te permiten mover toda la presentación (por ejemplo, código *HTML*) a un archivo separado y reutilizar diferentes partes del diseño de la página. En vez de escribir el código *HTML* dentro del controlador, en su lugar reproduce una plantilla:

```
1 // src/Acme/HolaBundle/Controller/HolaController.php
2 namespace Acme\HolaBundle\Controller;
3
```

```
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6  class HolaController extends Controller
7  {
8      public function indexAction($nombre)
9      {
10         return $this->render('AcmeHolaBundle:Hola:index.html.twig', array('nombre' => $nombre));
11
12         // reproduce una plantilla PHP en su lugar
13         // return $this->render('AcmeHolaBundle:Hola:index.html.php', array('nombre' => $nombre));
14     }
15 }
```

Nota: Para poder usar el método `render()`, debes extender la clase `Symfony\Bundle\FrameworkBundle\Controller\Controller` (documentación de la *API*: `Symfony\Bundle\FrameworkBundle\Controller\Controller`), la cual añade atajos para tareas comunes dentro de los controladores. Esto se hace en el ejemplo anterior añadiendo la declaración `use` en la línea 4 y luego extendiéndo el `Controlador` en la línea 6.

El método `render()` crea un objeto `Respuesta` poblado con el contenido propuesto, y reproduce la plantilla. Como cualquier otro controlador, en última instancia vas a devolver ese objeto `Respuesta`.

Ten en cuenta que hay dos ejemplos diferentes para procesar la plantilla. De forma predeterminada, *Symfony2* admite dos diferentes lenguajes de plantillas: las clásicas plantillas *PHP* y las breves pero poderosas plantillas Twig. No te espantes - eres libre de optar por una o, incluso, ambas en el mismo proyecto.

El controlador procesa la plantilla `AcmeHolaBundle:Hola:index.html.twig`, utilizando la siguiente convención de nomenclatura:

> **NombrePaquete**:**NombreControlador**:**NombrePlantilla**

Este es el nombre *lógico* de la plantilla, el cual se asigna a una ubicación física usando la siguiente convención.

> **/ruta/a/NombrePaquete/**Resources/views/**NombreControlador/NombrePlantilla**

En este caso, `AcmeHolaBundle` es el nombre del paquete, `Hola` es el controlador e `index.html.twig` la plantilla:

- *Twig*

```
1  {# src/Acme/HolaBundle/Resources/views/Hola/index.html.twig #}
2  {% extends '::base.html.twig' %}
3
4  {% block cuerpo %}
5      Hola {{ nombre }}!
6  {% endblock %}
```

- *PHP*

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/index.html.php -->
<?php $view->extend('::base.html.php') ?>

Hola <?php echo $view->escape($nombre) ?>!
```

Veamos la situación a través de la plantilla *Twig* línea por línea:

- *línea 2*: La etiqueta `extends` define una plantilla padre. La plantilla define explícitamente un archivo con el diseño dentro del cual será colocada.

- *línea 4*: La etiqueta `block` dice que todo el interior se debe colocar dentro de un bloque llamado `body`. Como verás, es responsabilidad de la plantilla padre (`base.html.twig`) reproducir, en última instancia, el bloque llamado `body`.

La plantilla padre, `::base.html.twig`, omite ambas porciones de su nombre tanto **NombrePaquete** como **NombreControlador** (de ahí los dobles dos puntos (`::`) al principio). Esto significa que la plantilla vive fuera de cualquier paquete, en el directorio `app`:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block titulo %}¡Bienvenido!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
        {% block cuerpo %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->salida('title', '¡Bienvenido!') ?></title>
        <?php $view['slots']->salida('stylesheets') ?>
        <link rel="shortcut icon" href="<?php echo $view['assets']->getUrl('favicon.ico') ?>" />
    </head>
    <body>
        <?php $view['slots']->salida('_content') ?>
        <?php $view['slots']->salida('stylesheets') ?>
    </body>
</html>
```

El archivo de la plantilla base define el diseño *HTML* y reproduce el bloque `body` que definiste en la plantilla `index.html.twig`. Además reproduce el bloque `título`, el cual puedes optar por definirlo en la plantilla `index.html.twig`. Dado que no has definido el bloque `título` en la plantilla derivada, el valor predeterminado es "¡Bienvenido!".

Las plantillas son una poderosa manera de reproducir y organizar el contenido de tu página. Una plantilla puede reproducir cualquier cosa, desde el marcado *HTML*, al código *CSS*, o cualquier otra cosa que el controlador posiblemente tenga que devolver.

En el ciclo de vida del manejo de una petición, el motor de plantillas simplemente es una herramienta opcional. Recuerda que el objetivo de cada controlador es devolver un objeto `Respuesta`. Las plantillas son una poderosa, pero opcional, herramienta para crear el contenido de ese objeto `Respuesta`.

2.4.2 La estructura de directorios

Después de unas cortas secciones, ya entiendes la filosofía detrás de la creación y procesamiento de páginas en *Symfony2*. También has comenzado a ver cómo están estructurados y organizados los proyectos *Symfony2*. Al final de esta

sección, sabrás dónde encontrar y colocar diferentes tipos de archivos y por qué.

Aunque totalmente flexible, por omisión, cada *aplicación Symfony* tiene la misma estructura de directorios básica y recomendada:

- `app/`: Este directorio contiene la configuración de la aplicación;
- `src/`: Todo el código *PHP* del proyecto se almacena en este directorio;
- `vendor/`: Por convención aquí se coloca cualquier biblioteca de terceros;
- `web/`: Este es el directorio web raíz y contiene todos los archivos de acceso público;

El directorio web

El directorio raíz del servidor web, es el hogar de todos los archivos públicos y estáticos tales como imágenes, hojas de estilo y archivos *JavaScript*. También es el lugar donde vive cada *controlador frontal*:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$nucleo = new AppKernel('prod', false);
$nucleo->loadClassCache();
$nucleo->handle(Request::createFromGlobals())->send();
```

El archivo del controlador frontal (`app.php` en este ejemplo) es el archivo *PHP* que realmente se ejecuta cuando utilizas una aplicación *Symfony2* y su trabajo consiste en utilizar una clase del núcleo, `AppKernel`, para arrancar la aplicación.

Truco: Tener un controlador frontal significa que se utilizan diferentes y más flexibles direcciones *URL* que en una aplicación *PHP* típica. Cuando usamos un controlador frontal, las direcciones *URL* se formatean de la siguiente manera:

```
http://localhost/app.php/hola/Ryan
```

El controlador frontal, `app.php`, se ejecuta y la *URL* "interna": `/hola/Ryan` es ruteada internamente con la configuración de enrutado. Al utilizar las reglas `mod_rewrite` de Apache, puedes forzar la ejecución del archivo `app.php` sin necesidad de especificarlo en la *URL*:

```
http://localhost/hola/Ryan
```

Aunque los controladores frontales son esenciales en el manejo de cada petición, rara vez los tendrás que modificar o incluso pensar en ellos. Los vamos a mencionar brevemente de nuevo en la sección de Entornos (Página 69).

El directorio aplicación (`app`)

Como vimos en el controlador frontal, la clase `AppKernel` es el punto de entrada principal de la aplicación y es la responsable de toda la configuración. Como tal, se almacena en el directorio `app/`.

Esta clase debe implementar dos métodos que definen todo lo que *Symfony* necesita saber acerca de tu aplicación. Ni siquiera tienes que preocuparte de estos métodos durante el arranque - *Symfony* los llena por ti con parámetros predeterminados.

- `registerBundles()`: Devuelve una matriz con todos los paquetes necesarios para ejecutar la aplicación (consulta *El sistema de paquetes* (Página 64));

- `registerContainerConfiguration()`: Carga el archivo de configuración de recursos principal de la aplicación (consulta la sección Configurando la aplicación (Página 67));

En el desarrollo del día a día, generalmente vas a utilizar el directorio `app/` para modificar la configuración y los archivos de enrutado en el directorio `app/config/` (consulta la sección Configurando la aplicación (Página 67)). Este también contiene el directorio caché de la aplicación (`app/cache`), un directorio de registro (`app/logs`) y un directorio para archivos de recursos a nivel de aplicación, tal como plantillas (`app/Resources`). Aprenderás más sobre cada uno de estos directorios en capítulos posteriores.

Cargando automáticamente

Al arrancar *Symfony*, un archivo especial - `app/autoload.php` - es incluido. Este archivo es responsable de configurar el cargador automático, el cual cargará automáticamente los archivos de tu aplicación desde el directorio `src/` y librerías de terceros del directorio `vendor/`.
Gracias al cargador automático, nunca tendrás que preocuparte de usar declaraciones `include` o `require`. En cambio, *Symfony2* utiliza el espacio de nombres de una clase para determinar su ubicación e incluir automáticamente el archivo en el instante en que necesitas una clase.
El cargador automático ya está configurado para buscar en el directorio `src/` cualquiera de tus clases *PHP*. Para que trabaje la carga automática, el nombre de la clase y la ruta del archivo deben seguir el mismo patrón:

```
Nombre de clase:
    Acme\HolaBundle\Controller\HolaController
Ruta:
    src/Acme/HolaBundle/Controller/HolaController.php
```

Típicamente, la única vez que tendrás que preocuparte por el archivo `app/autoload.php` es cuando estás incluyendo una nueva biblioteca de terceros en el directorio `vendor/`. Para más información sobre la carga automática, consulta *Cómo cargar clases automáticamente* (Página 407).

El directorio fuente (`src`)

En pocas palabras, el directorio `src/` contiene todo el código real (código *PHP*, plantillas, archivos de configuración, estilo, etc.) que impulsa a *tu* aplicación. De hecho, cuando desarrollas, la gran mayoría de tu trabajo se llevará a cabo dentro de uno o más paquetes creados en este directorio.

Pero, ¿qué es exactamente un *paquete*?

2.4.3 El sistema de paquetes

Un paquete es similar a un complemento en otro software, pero aún mejor. La diferencia clave es que en *Symfony2* **todo** es un paquete, incluyendo tanto la funcionalidad básica de la plataforma como el código escrito para tu aplicación. Los paquetes son ciudadanos de primera clase en *Symfony2*. Esto te proporciona la flexibilidad para utilizar las características preconstruidas envasadas en paquetes de terceros o para distribuir tus propios paquetes. Además, facilita la selección y elección de las características por habilitar en tu aplicación y optimizarlas en la forma que desees.

Nota: Si bien, aquí vamos a cubrir lo básico, hay un capítulo dedicado completamente al tema de los *paquetes* (Página 346).

Un paquete simplemente es un conjunto estructurado de archivos en un directorio que implementa una sola característica. Puedes crear un `BlogBundle`, un `ForoBundle` o un paquete para gestionar usuarios (muchos de ellos ya existen como paquetes de código abierto). Cada directorio contiene todo lo relacionado con esa característica, incluyendo archivos *PHP*, plantillas, hojas de estilo, archivos *Javascript*, pruebas y cualquier otra cosa necesaria. Cada aspecto de una característica existe en un paquete y cada característica vive en un paquete.

Una aplicación se compone de paquetes tal como está definido en el método `registerBundles()` de la clase `AppKernel`:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

Con el método `registerBundles()`, tienes el control total sobre cuales paquetes utiliza tu aplicación (incluyendo los paquetes del núcleo de *Symfony*).

Truco: Un paquete puede vivir en *cualquier lugar* siempre y cuando *Symfony2* lo pueda cargar automáticamente (vía el autocargador configurado en `app/autoload.php`).

Creando un paquete

La edición estándar de *Symfony* viene con una práctica tarea que crea un paquete totalmente funcional para ti. Por supuesto, la creación manual de un paquete también es muy fácil.

Para mostrarte lo sencillo que es el sistema de paquetes, vamos a crear y activar un nuevo paquete llamado `AcmePruebaBundle`.

Truco: La parte `Acme` es sólo un nombre ficticio que debes sustituir por un "proveedor" que represente tu nombre u organización (por ejemplo, `ABCPruebaBundle` por alguna empresa llamada `ABC`).

En primer lugar, crea un directorio `src/Acme/PruebaBundle/` y añade un nuevo archivo llamado `AcmePruebaBundle.php`:

```
// src/Acme/PruebaBundle/AcmePruebaBundle.php
namespace Acme\PruebaBundle;

    use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmePruebaBundle extends Bundle
```

```
{
}
```

Truco: El nombre `AcmePruebaBundle` sigue las *convenciones de nomenclatura de paquetes* (Página 347) estándar. También puedes optar por acortar el nombre del paquete simplemente a `PruebaBundle` al nombrar esta clase `PruebaBundle` (y el nombre del archivo `PruebaBundle.php`).

Esta clase vacía es la única pieza que necesitamos para crear nuestro nuevo paquete. Aunque comúnmente está vacía, esta clase es poderosa y se puede utilizar para personalizar el comportamiento del paquete.

Ahora que hemos creado nuestro paquete, tenemos que activarlo a través de la clase `AppKernel`:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...

        // registra tus paquetes
        new Acme\PruebaBundle\AcmePruebaBundle(),
    );
    // ...

    return $bundles;
}
```

Y si bien `AcmePruebaBundle` aún no hace nada, está listo para utilizarlo.

Y aunque esto es bastante fácil, *Symfony* también proporciona una interfaz de línea de ordenes para generar una estructura de paquete básica:

```
php app/console generate:bundle --namespace=Acme/PruebaBundle
```

Esto genera el esqueleto del paquete con un controlador básico, la plantilla y recursos de enrutado que se pueden personalizar. Aprenderás más sobre la línea de ordenes de las herramientas de *Symfony2* más tarde.

Truco: Cuando quieras crear un nuevo paquete o uses un paquete de terceros, siempre asegúrate de habilitar el paquete en `registerBundles()`. Cuando usas la orden `generate:bundle`, hace esto para ti.

Estructura de directorios de un paquete

La estructura de directorios de un paquete es simple y flexible. De forma predeterminada, el sistema de paquetes sigue una serie de convenciones que ayudan a mantener el código consistente entre todos los paquetes *Symfony2*. Echa un vistazo a `AcmeHolaBundle`, ya que contiene algunos de los elementos más comunes de un paquete:

- `Controller/` Contiene los controladores del paquete (por ejemplo, `HolaController.php`);
- `Resources/config/` Contiene la configuración, incluyendo la configuración de enrutado (por ejemplo, `routing.yml`);
- `Resources/views/` Contiene las plantillas organizadas por nombre de controlador (por ejemplo, `Hola/index.html.twig`);
- `Resources/public/` Contiene recursos web (imágenes, hojas de estilo, etc.) y es copiado o enlazado simbólicamente al directorio `web/` del proyecto vía la orden de consola inicial `assets:install`;
- `Tests/` Tiene todas las pruebas para el paquete.

Un paquete puede ser tan pequeño o tan grande como la característica que implementa. Este contiene sólo los archivos que necesita y nada más.

A medida que avances en el libro, aprenderás cómo persistir objetos a una base de datos, crear y validar formularios, crear traducciones para tu aplicación, escribir pruebas y mucho más. Cada uno de estos tiene su propio lugar y rol dentro del paquete.

2.4.4 Configurando la aplicación

La aplicación consiste de una colección de paquetes que representan todas las características y capacidades de tu aplicación. Cada paquete se puede personalizar a través de archivos de configuración escritos en *YAML*, *XML* o *PHP*. De forma predeterminada, el archivo de configuración principal vive en el directorio `app/config/` y se llama `config.yml`, `config.xml` o `config.php` en función del formato que prefieras:

- *YAML*

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

framework:
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    session:
        default_locale: %locale%
        auto_start:     true

# Configuración Twig
twig:
    debug:            %kernel.debug%
    strict_variables: %kernel.debug%

# ...
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="parameters.ini" />
    <import resource="security.yml" />
</imports>

<framework:config charset="UTF-8" secret="%secret%">
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <framework:form />
    <framework:csrf-protection />
    <framework:validation annotations="true" />
    <framework:templating assets-version="SomeVersionScheme">
        <framework:engine id="twig" />
    </framework:templating>
    <framework:session default-locale="%locale%" auto-start="true" />
</framework:config>
```

```
<!-- Configuración Twig -->
<twig:config debug="%kernel.debug%" strict-variables="%kernel.debug%" />

<!-- ... -->
```

- *PHP*

```
$this->import('parameters.ini');
$this->import('security.yml');

$contenedor->loadFromExtension('framework', array(
    'secret'          => '%secret%',
    'charset'         => 'UTF-8',
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
    'form'            => array(),
    'csrf-protection' => array(),
    'validation'      => array('annotations' => true),
    'templating'      => array(
        'engines' => array('twig'),
        #'assets_version' => "SomeVersionScheme",
    ),
    'session' => array(
        'default_locale' => "%locale%",
        'auto_start'     => true,
    ),
));

// Configuración Twig
$contenedor->loadFromExtension('twig', array(
    'debug'            => '%kernel.debug%',
    'strict_variables' => '%kernel.debug%',
));

// ...
```

Nota: Aprenderás exactamente cómo cargar cada archivo/formato en la siguiente sección Entornos (Página 69).

Cada entrada de nivel superior como `framework` o `twig` define la configuración de un paquete específico. Por ejemplo, la clave `framework` define la configuración para el núcleo de *Symfony* `FrameworkBundle` e incluye la configuración de enrutado, plantillas, y otros sistemas del núcleo.

Por ahora, no te preocupes por las opciones de configuración específicas de cada sección. El archivo de configuración viene con parámetros predeterminados. A medida que leas y explores más cada parte de *Symfony2*, aprenderás sobre las opciones de configuración específicas de cada característica.

Formatos de configuración

A lo largo de los capítulos, todos los ejemplos de configuración muestran los tres formatos (*YAML*, *XML* y *PHP*). Cada uno tiene sus propias ventajas y desventajas. Tú eliges cual utilizar:

- *YAML*: Sencillo, limpio y fácil de leer;
- *XML*: Más poderoso que YAML a veces y es compatible con el autocompletado del IDE;
- *PHP*: Muy potente, pero menos fácil de leer que los formatos de configuración estándar.

2.4.5 Entornos

Una aplicación puede funcionar en diversos entornos. Los diferentes entornos comparten el mismo código *PHP* (aparte del controlador frontal), pero usan diferente configuración. Por ejemplo, un entorno de desarrollo `dev` registrará las advertencias y errores, mientras que un entorno de producción `prod` sólo registra los errores. Algunos archivos se vuelven a generar en cada petición en el entorno `dev` (para mayor comodidad de los desarrolladores), pero se memorizan en caché en el entorno `prod`. Todos los entornos viven juntos en la misma máquina y ejecutan la misma aplicación.

Un proyecto *Symfony2* generalmente comienza con tres entornos (`dev`, `test` y `prod`), aunque la creación de nuevos entornos es fácil. Puedes ver tu aplicación en diferentes entornos con sólo cambiar el controlador frontal en tu navegador. Para ver la aplicación en el entorno `dev`, accede a la aplicación a través del controlador frontal de desarrollo:

```
http://localhost/app_dev.php/hola/Ryan
```

Si deseas ver cómo se comportará tu aplicación en el entorno de producción, en su lugar, llama al controlador frontal `prod`:

```
http://localhost/app.php/hola/Ryan
```

Puesto que el entorno `prod` está optimizado para velocidad, la configuración, el enrutado y las plantillas *Twig* se compilan en clases *PHP* simples y se guardan en caché. Cuando cambies las vistas en el entorno `prod`, tendrás que borrar estos archivos memorizados en caché y así permitir su reconstrucción:

```
php app/console cache:clear --env=prod --no-debug
```

Nota: Si abres el archivo `web/app.php`, encontrarás que está configurado explícitamente para usar el entorno `prod`:

```
$kernel = new AppKernel('prod', false);
```

Puedes crear un nuevo controlador frontal para un nuevo entorno copiando el archivo y cambiando `prod` por algún otro valor.

Nota: El entorno `test` se utiliza cuando se ejecutan pruebas automáticas y no se puede acceder directamente a través del navegador. Consulta el capítulo *Probando* (Página 137) para más detalles.

Configurando entornos

La clase `AppKernel` es responsable de cargar realmente el archivo de configuración de tu elección:

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

Ya sabes que la extensión `.yml` se puede cambiar a `.xml` o `.php` si prefieres usar *XML* o *PHP* para escribir tu configuración. Además, observa que cada entorno carga su propio archivo de configuración. Considera el archivo de configuración para el entorno `dev`.

- *YAML*

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

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

# ...
```

- *XML*

```xml
<!-- app/config/config_dev.xml -->
<imports>
    <import resource="config.xml" />
</imports>

<framework:config>
    <framework:router resource="%kernel.root_dir%/config/routing_dev.xml" />
    <framework:profiler only-exceptions="false" />
</framework:config>

<!-- ... -->
```

- *PHP*

```php
// app/config/config_dev.php
$cargador->import('config.php');

$contenedor->loadFromExtension('framework', array(
    'router'   => array('resource' => '%kernel.root_dir%/config/routing_dev.php'),
    'profiler' => array('only-exceptions' => false),
));

// ...
```

La clave `imports` es similar a una declaración `include` PHP y garantiza que en primer lugar se carga el archivo de configuración principal (`config.yml`). El resto del archivo de configuración predeterminado aumenta el registro de eventos y otros ajustes conducentes a un entorno de desarrollo.

Ambos entornos `prod` y `test` siguen el mismo modelo: cada uno importa el archivo de configuración básico y luego modifica sus valores de configuración para adaptarlos a las necesidades específicas del entorno. Esto es sólo una convención, pero te permite reutilizar la mayor parte de tu configuración y personalizar sólo piezas puntuales entre entornos.

2.4.6 Resumen

¡Enhorabuena! Ahora has visto todos los aspectos fundamentales de *Symfony2* y afortunadamente descubriste lo fácil y flexible que puede ser. Y si bien aún *hay muchas* características por venir, asegúrate de tener en cuenta los siguientes puntos básicos:

- La creación de una página es un proceso de tres pasos que involucran una **ruta**, un **controlador** y (opcionalmente) una **plantilla**.
- Cada proyecto contiene sólo unos cuantos directorios principales: `web/` (recursos web y controladores frontales), `app/` (configuración), `src/` (tus paquetes), y `vendor/` (código de terceros) (también hay un directorio `bin/` que se utiliza para ayudarte a actualizar las bibliotecas de proveedores);

- Cada característica en *Symfony2* (incluyendo el núcleo de la plataforma *Symfony2*) está organizada en un *paquete*, el cual es un conjunto estructurado de archivos para esa característica;
- La **configuración** de cada paquete vive en el directorio `app/config` y se puede especificar en *YAML*, *XML* o *PHP*;
- Cada **entorno** es accesible a través de un diferente controlador frontal (por ejemplo, `app.php` y `app_dev.php`) y carga un archivo de configuración diferente.

A partir de aquí, cada capítulo te dará a conocer más y más potentes herramientas y conceptos avanzados. Cuanto más sepas sobre *Symfony2*, tanto más apreciarás la flexibilidad de su arquitectura y el poder que te proporciona para desarrollar aplicaciones rápidamente.

2.5 Controlador

Un controlador es una función *PHP* que tú creas, misma que toma información desde la petición *HTTP* y construye una respuesta *HTTP* y la devuelve (como un objeto `Respuesta` de *Symfony2*). La respuesta podría ser una página *HTML*, un documento *XML*, una matriz *JSON* serializada, una imagen, una redirección, un error 404 o cualquier otra cosa que se te ocurra. El controlador contiene toda la lógica arbitraria que *tu aplicación necesita* para reproducir el contenido de la página.

Para ver lo sencillo que es esto, echemos un vistazo a un controlador de *Symfony2* en acción. El siguiente controlador reproducirá una página que simplemente imprime `¡Hola, mundo!`:

```
use Symfony\Component\HttpFoundation\Response;

public function holaAction()
{
    return new Response('¡Hola mundo!');
}
```

El objetivo de un controlador siempre es el mismo: crear y devolver un objeto `Respuesta`. Por el camino, este puede leer la información de la petición, cargar un recurso de base de datos, enviar un correo electrónico, o fijar información en la sesión del usuario. Pero en todos los casos, el controlador eventualmente devuelve el objeto `Respuesta` que será entregado al cliente.

¡No hay magia y ningún otro requisito del cual preocuparse! Aquí tienes unos cuantos ejemplos comunes:

- *Controlador A* prepara un objeto `Respuesta` que reproduce el contenido de la página principal del sitio.
- *Controlador B* lee el parámetro `ficha` de la petición para cargar una entrada del blog de la base de datos y crear un objeto `Respuesta` mostrando ese blog. Si la `ficha` no se puede encontrar en la base de datos, crea y devuelve un objeto `Respuesta` con un código de estado 404.
- *Controlador C* procesa la información presentada en un formulario de contacto. Este lee la información del formulario desde la petición, guarda la información del contacto en la base de datos y envía mensajes de correo electrónico con la información de contacto al administrador del sitio web. Por último, crea un objeto `Respuesta` que redirige el navegador del cliente desde el formulario de contacto a la página de *"agradecimiento"*.

2.5.1 Ciclo de vida de la petición, controlador, respuesta

Cada petición manejada por un proyecto *Symfony2* pasa por el mismo ciclo de vida básico. La plataforma se encarga de las tareas repetitivas y, finalmente, ejecuta el controlador, que contiene el código personalizado de tu aplicación:

1. Cada petición es manejada por un único archivo controlador frontal (por ejemplo, `app.php` o `app_dev.php`) el cual es responsable de arrancar la aplicación;

2. El `Enrutador` lee la información de la petición (por ejemplo, la *URI*), encuentra una ruta que coincida con esa información, y lee el parámetro `_controller` de la ruta;
3. El controlador de la ruta encontrada es ejecutado y el código dentro del controlador crea y devuelve un objeto `Respuesta`;
4. Las cabeceras *HTTP* y el contenido del objeto `Respuesta` se envían de vuelta al cliente.

La creación de una página es tan fácil como crear un controlador (#3) y hacer una ruta que vincula una *URI* con ese controlador (#2).

Nota: Aunque nombrados de manera similar, un "controlador frontal" es diferente de los "controladores" vamos a hablar acerca de eso en este capítulo. Un controlador frontal es un breve archivo *PHP* que vive en tu directorio web raíz y a través del cual se dirigen todas las peticiones. Una aplicación típica tendrá un controlador frontal de producción (por ejemplo, `app.php`) y un controlador frontal de desarrollo (por ejemplo, `app_dev.php`). Probablemente nunca necesites editar, ver o preocuparte por los controladores frontales en tu aplicación.

2.5.2 Un controlador sencillo

Mientras que un controlador puede ser cualquier ejecutable *PHP* (una función, un método en un objeto o un `Cierre`), en *Symfony2*, un controlador suele ser un único método dentro de un objeto controlador. Los controladores también se conocen como *acciones*.

```
1  // src/Acme/HolaBundle/Controller/HolaController.php
2
3  namespace Acme\HolaBundle\Controller;
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HolaController
7  {
8      public function indexAction($nombre)
9      {
10       return new Response('<html><body>Hola '.$nombre.'!</body></html>');
11     }
12 }
```

Truco: Ten en cuenta que el *controlador* es el método `indexAction`, que vive dentro de una *clase controlador* (`HolaController`). No te dejes confundir por la nomenclatura: una *clase controlador* simplemente es una conveniente forma de agrupar varios controladores/acciones. Generalmente, la clase controlador albergará varios controladores (por ejemplo, `updateAction`, `deleteAction`, etc.).

Este controlador es bastante sencillo, pero vamos a revisarlo línea por línea:

- En la *línea 3*: *Symfony2* toma ventaja de la funcionalidad del espacio de nombres de *PHP 5.3* para el espacio de nombres de la clase del controlador completa. La palabra clave `use` importa la clase `Respuesta`, misma que nuestro controlador debe devolver.
- En la *línea 6*: el nombre de clase es la concatenación del nombre de la clase controlador (es decir `Hola`) y la palabra `Controller`. Esta es una convención que proporciona consistencia a los controladores y permite hacer referencia sólo a la primera parte del nombre (es decir, `Hola`) en la configuración del enrutador.
- En la *línea 8*: Cada acción en una clase controlador prefijada con `Action` y referida en la configuración de enrutado por el nombre de la acción (`index`). En la siguiente sección, crearás una ruta que asigna un *URI* a esta acción. Aprenderás cómo los marcadores de posición de la ruta (`{nombre}`) se convierten en argumentos para el método de acción (`$nombre`).

- En la *línea 10*: el controlador crea y devuelve un objeto `Respuesta`.

2.5.3 Asignando una *URI* a un controlador

El nuevo controlador devuelve una página *HTML* simple. Para realmente ver esta página en tu navegador, necesitas crear una ruta, la cual corresponda a un patrón de *URL* específico para el controlador:

- *YAML*

```
# app/config/routing.yml
hola:
    pattern:      /hola/{nombre}
    defaults:     { _controller: AcmeHolaBundle:Hola:index }
```

- *XML*

```
<!-- app/config/routing.xml -->
<route id="hola" pattern="/hola/{nombre}">
    <default key="_controller">AcmeHolaBundle:Hola:index</default>
</route>
```

- *PHP*

```
// app/config/routing.php
$coleccion->add('hola', new Route('/hola/{nombre}', array(
    '_controller' => 'AcmeHolaBundle:Hola:index',
)));
```

Yendo ahora a `/hola/ryan` se ejecuta el controlador `HolaController::indexAction()` y pasa `ryan` a la variable `$nombre`. Crear una "página" significa simplemente que debes crear un método controlador y una ruta asociada.

Observa la sintaxis utilizada para referirse al controlador: `AcmeHolaBundle:Hola:index`. *Symfony2* utiliza una flexible notación de cadena para referirse a diferentes controladores. Esta es la sintaxis más común y le dice *Symfony2* que busque una clase controlador llamada `HolaController` dentro de un paquete llamado `AcmeHolaBundle`. Entonces ejecuta el método `indexAction()`.

Para más detalles sobre el formato de cadena utilizado para referir a diferentes controladores, consulta el *Patrón de nomenclatura para controladores* (Página 93).

Nota: Este ejemplo coloca la configuración de enrutado directamente en el directorio `app/config/`. Una mejor manera de organizar tus rutas es colocar cada ruta en el paquete al que pertenece. Para más información sobre este tema, consulta *Incluyendo recursos de enrutado externos* (Página 94).

Truco: Puedes aprender mucho más sobre el sistema de enrutado en el *capítulo de enrutado* (Página 81).

Parámetros de ruta como argumentos para el controlador

Ya sabes que el parámetro `_controller` en `AcmeHolaBundle:Hola:index` se refiere al método `HolaController::indexAction()` que vive dentro del paquete `AcmeHolaBundle`. Lo más interesante de esto son los argumentos que se pasan a este método:

```
<?php
// src/Acme/HolaBundle/Controller/HolaController.php
```

```
namespace Acme\HolaBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HolaController extends Controller
{
    public function indexAction($nombre)
    {
      // ...
    }
}
```

El controlador tiene un solo argumento, `$nombre`, el cual corresponde al parámetro `{nombre}` de la ruta coincidente (`ryan` en nuestro ejemplo). De hecho, cuando ejecutas tu controlador, *Symfony2* empareja cada argumento del controlador con un parámetro de la ruta coincidente. Tomemos el siguiente ejemplo:

- *YAML*

  ```
  # app/config/routing.yml
  hola:
      pattern:      /hola/{nombre_de_pila}/{apellido}
      defaults:     { _controller: AcmeHolaBundle:Hola:index, color: verde }
  ```

- *XML*

  ```
  <!-- app/config/routing.xml -->
  <route id="hola" pattern="/hola/{nombre_de_pila}/{apellido}">
      <default key="_controller">AcmeHolaBundle:Hola:index</default>
      <default key="color">verde</default>
  </route>
  ```

- *PHP*

  ```
  // app/config/routing.php
  $coleccion->add('hola', new Route('/hola/{nombre_de_pila}/{apellido}', array(
      '_controller' => 'AcmeHolaBundle:Hola:index',
      'color'       => 'verde',
  )));
  ```

El controlador para esto puede tomar varios argumentos:

```
public function indexAction($nombre_de_pila, $apellido, $color)
{
    // ...
}
```

Ten en cuenta que ambas variables marcadoras de posición (`{nombre_de_pila}`, `{apellido}`) así como la variable predeterminada `color` están disponibles como argumentos en el controlador. Cuando una ruta corresponde, las variables marcadoras de posición se combinan con `defaults` para hacer que una matriz esté disponible para tu controlador.

Asignar parámetros de ruta a los argumentos del controlador es fácil y flexible. Ten muy en cuenta las siguientes pautas mientras desarrollas.

- **El orden de los argumentos del controlador no tiene importancia**

  Symfony2 es capaz de igualar los nombres de los parámetros de la ruta con los nombres de las variables en la firma del método controlador. En otras palabras, se da cuenta de que el parámetro `{apellido}` coincide con el argumento `$apellido`. Los argumentos del controlador se pueden reorganizar completamente y aún así siguen funcionando perfectamente:

```
public function indexAction($apellido, $color, $nombre_de_pila)
{
    // ..
}
```

- **Cada argumento requerido del controlador debe coincidir con un parámetro de enrutado**

  Lo siguiente lanzará una `RuntimeException` porque no hay ningún parámetro `foo` definido en la ruta:

  ```
  public function indexAction($nombre_de_pila, $apellido, $color, $foo)
  {
      // ..
  }
  ```

  Sin embargo, hacer que el argumento sea opcional, está perfectamente bien. El siguiente ejemplo no lanzará una excepción:

  ```
  public function indexAction($nombre_de_pila, $apellido, $color, $foo = 'bar')
  {
      // ..
  }
  ```

- **No todos los parámetros de enrutado deben ser argumentos en tu controlador**

  Si, por ejemplo, `apellido` no es tan importante para tu controlador, lo puedes omitir por completo:

  ```
  public function indexAction($nombre_de_pila, $color)
  {
      // ..
  }
  ```

Truco: Además, todas las rutas tienen un parámetro especial `_route`, el cual es igual al nombre de la ruta con la que fue emparejado (por ejemplo, `hola`). Aunque no suele ser útil, igualmente está disponible como un argumento del controlador.

La `Petición` como argumento para el controlador

Para mayor comodidad, también puedes hacer que *Symfony* pase el objeto `Petición` como un argumento a tu controlador. Esto especialmente es conveniente cuando trabajas con formularios, por ejemplo:

```
use Symfony\Component\HttpFoundation\Request;

public function updateAction(Request $peticion)
{
    $formulario = $this->createForm(...);

    $formulario->bindRequest($peticion);
    // ...
}
```

2.5.4 La clase base del controlador

Para mayor comodidad, *Symfony2* viene con una clase base `Controller` que te ayuda en algunas de las tareas más comunes del controlador y proporciona acceso a cualquier recurso que tu clase controlador pueda necesitar. Al extender esta clase `Controller`, puedes tomar ventaja de varios métodos ayudantes.

Agrega la instrucción `use` en lo alto de la clase `Controller` y luego modifica `HolaController` para extenderla:

```
// src/Acme/HolaBundle/Controller/HolaController.php

namespace Acme\HolaBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HolaController extends Controller
{
    public function indexAction($nombre)
    {
      return new Response('<html><body>Hola '.$nombre.'!</body></html>');
    }
}
```

Esto, en realidad no cambia nada acerca de cómo funciona el controlador. En la siguiente sección, aprenderás acerca de los métodos ayudantes que la clase base del controlador pone a tu disposición. Estos métodos sólo son atajos para utilizar la funcionalidad del núcleo de *Symfony2* que está a nuestra disposición, usando o no la clase base `Controller`. Una buena manera de ver la funcionalidad del núcleo en acción es buscar en la misma clase `Symfony\Bundle\FrameworkBundle\Controller\Controller`.

Truco: Extender la clase base es *opcional* en *Symfony*; esta contiene útiles atajos, pero no es obligatorio. También puedes extender la clase `Symfony\Component\DependencyInjection\ContainerAware`. El objeto contenedor del servicio será accesible a través de la propiedad `container`.

Nota: Puedes definir tus *Controladores como Servicios* (Página 278).

2.5.5 Tareas comunes del controlador

A pesar de que un controlador puede hacer prácticamente cualquier cosa, la mayoría de los controladores se encargarán de las mismas tareas básicas una y otra vez. Estas tareas, tal como redirigir, procesar plantillas y acceder a servicios básicos, son muy fáciles de manejar en *Symfony2*.

Redirigiendo

Si deseas redirigir al usuario a otra página, utiliza el método `redirect()`:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('portada'));
}
```

El método `generateUrl()` es sólo una función auxiliar que genera la *URL* de una determinada ruta. Para más información, consulta el capítulo *Enrutando* (Página 81).

Por omisión, el método `redirect()` produce una redirección 302 (temporal). Para realizar una redirección 301 (permanente), modifica el segundo argumento:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('portada'), 301);
}
```

Truco: El método `redirect()` simplemente es un atajo que crea un objeto `Respuesta` que se especializa en redirigir a los usuarios. Es equivalente a:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

return new RedirectResponse($this->generateUrl('portada'));
```

Reenviando

También, fácilmente puedes redirigir internamente hacia a otro controlador con el método `forward()`. En lugar de redirigir el navegador del usuario, este hace una subpetición interna, y pide el controlador especificado. El método `forward()` devuelve el objeto `Respuesta`, el cual es devuelto desde el controlador:

```
public function indexAction($nombre)
{
    $respuesta = $this->forward('AcmeHolaBundle:Hola:maravillosa', array(
        'nombre'  => $nombre,
        'color' => 'verde'
    ));

    // alguna modificación adicional a la respuesta o la devuelve directamente

    return $respuesta;
}
```

Ten en cuenta que el método `forward()` utiliza la misma representación de cadena del controlador utilizada en la configuración de enrutado. En este caso, la clase controlador de destino será `HolaController` dentro de algún `AcmeHolaBundle`. La matriz pasada al método convierte los argumentos en el controlador resultante. Esta misma interfaz se utiliza al incrustar controladores en las plantillas (consulta *Incrustando controladores* (Página 106)). El método del controlador destino debe tener un aspecto como el siguiente:

```
public function maravillosaAction($nombre, $color)
{
    // ... crea y devuelve un objeto Response
}
```

Y al igual que al crear un controlador para una ruta, el orden de los argumentos para `maravillosaAction` no tiene la menor importancia. *Symfony2* empareja las claves nombre con el índice (por ejemplo, `nombre`) con el argumento del método (por ejemplo, `$nombre`). Si cambias el orden de los argumentos, *Symfony2* todavía pasará el valor correcto a cada variable.

Truco: Al igual que otros métodos del `Controller` base, el método `forward` sólo es un atajo para la funcionalidad del núcleo de *Symfony2*. Puedes redirigir directamente por medio del servicio `http_kernel`. Un método `forward` devuelve un objeto `Respuesta`:

```
$httpKernel = $this->contenedor->get('http_kernel');
$respuesta = $httpKernel->forward('AcmeHolaBundle:Hola:maravillosa', array(
    'nombre'  => $nombre,
    'color' => 'verde',
));
```

Procesando plantillas

Aunque no es un requisito, la mayoría de los controladores en última instancia, reproducen una plantilla que es responsable de generar el código *HTML* (u otro formato) para el controlador. El método `renderView()` procesa una plantilla y devuelve su contenido. Puedes usar el contenido de la plantilla para crear un objeto `Respuesta`:

```
$contenido = $this->renderView('AcmeHolaBundle:Hola:index.html.twig', array('nombre' => $nombre));

return new Response($contenido);
```

Incluso puedes hacerlo en un solo paso con el método `render()`, el cual devuelve un objeto `Respuesta` con el contenido de la plantilla:

```
return $this->render('AcmeHolaBundle:Hola:index.html.twig', array('nombre' => $nombre));
```

En ambos casos, se reproducirá la plantilla `Resources/views/Hola/index.html.twig` dentro del `AcmeHolaBundle`.

El motor de plantillas de *Symfony* se explica con gran detalle en el capítulo *Plantillas* (Página 98).

Truco: El método `renderView` es un atajo para usar el servicio de `plantillas`. El servicio de `plantillas` también se puede utilizar directamente:

```
$plantilla = $this->get('templating');
$contenido = $plantilla->render('AcmeHolaBundle:Hola:index.html.twig', array('nombre' => $nombre));
```

Accediendo a otros servicios

Al extender la clase base del controlador, puedes acceder a cualquier servicio de *Symfony2* a través del método `get()`. Aquí hay varios servicios comunes que puedes necesitar:

```
$peticion = $this->getRequest();

$respuesta = $this->get('response');

$plantilla = $this->get('templating');

$enrutador = $this->get('router');

$cartero = $this->get('mailer');
```

Hay un sinnúmero de servicios disponibles y te animamos a definir tus propios servicios. Para listar todos los servicios disponibles, utiliza la orden `container:debug` de la consola:

```
php app/console container:debug
```

Para más información, consulta el capítulo *Contenedor de servicios* (Página 237).

2.5.6 Gestionando errores y páginas 404

Cuando no se encuentra algo, debes jugar bien con el protocolo *HTTP* y devolver una respuesta 404. Para ello, debes lanzar un tipo de excepción especial. Si estás extendiendo la clase base del controlador, haz lo siguiente:

```
public function indexAction()
{
    $producto = // recupera el objeto desde la base de datos
    if (!$producto) {
        throw $this->createNotFoundException('El producto no existe');
    }

    return $this->render(...);
}
```

El método `createNotFoundException()` crea un objeto `NotFoundHttpException` especial, que en última instancia, desencadena una respuesta *HTTP 404* en el interior de *Symfony*.

Por supuesto, estás en libertad de lanzar cualquier clase de `Excepción` en tu controlador - *Symfony2* automáticamente devolverá una respuesta *HTTP* con código 500.

```
throw new \Exception('¡Algo salió mal!');
```

En todos los casos, se muestra al usuario final una página de error estilizada y a los desarrolladores se les muestra una página de error de depuración completa (cuando se visualiza la página en modo de depuración). Puedes personalizar ambas páginas de error. Para más detalles, lee "*Cómo personalizar páginas de error* (Página 277)" en el recetario.

2.5.7 Gestionando la sesión

Symfony2 proporciona un agradable objeto sesión que puedes utilizar para almacenar información sobre el usuario (ya sea una persona real usando un navegador, un robot o un servicio web) entre las peticiones. De manera predeterminada, *Symfony2* almacena los atributos de una `cookie` usando las sesiones nativas de *PHP*.

Almacenar y recuperar información de la sesión se puede conseguir fácilmente desde cualquier controlador:

```
$sesion = $this->getRequest()->getSession();

// guarda un atributo para reutilizarlo durante una posterior petición del usuario
$sesion->set('foo', 'bar');

// en otro controlador por otra petición
$foo = $sesion->get('foo');

// fija la configuración regional del usuario
$sesion->setLocale('es');
```

Estos atributos se mantendrán en el usuario por el resto de la sesión de ese usuario.

Mensajes flash

También puedes almacenar pequeños mensajes que se pueden guardar en la sesión del usuario para exactamente una petición adicional. Esto es útil cuando procesas un formulario: deseas redirigir y proporcionar un mensaje especial que aparezca en la *siguiente* petición. Este tipo de mensajes se conocen como mensajes "flash".

Por ejemplo, imagina que estás procesando el envío de un formulario:

```
public function updateAction()
{
    $formulario = $this->createForm(...);

    $formulario->bindRequest($this->getRequest());
    if ($formulario->isValid()) {
```

```
        // hace algún tipo de procesamiento

        $this->get('session')->setFlash('aviso', '¡Tus cambios se han guardado!');

        return $this->redirect($this->generateUrl(...));
    }

    return $this->render(...);
}
```

Después de procesar la petición, el controlador establece un mensaje flash `aviso` y luego redirige al usuario. El nombre (`aviso`) no es significativo - es lo que estás usando para identificar el tipo del mensaje.

En la siguiente acción de la plantilla, podrías utilizar el siguiente código para reproducir el mensaje de `aviso`:

- *Twig*

```
{% if app.session.hasFlash('aviso') %}
    <div class="flash-aviso">
        {{ app.session.flash('aviso') }}
    </div>
{% endif %}
```

- *PHP*

```
<?php if ($view['session']->hasFlash('aviso')): ?>
    <div class="flash-aviso">
        <?php echo $view['session']->getFlash('aviso') ?>
    </div>
<?php endif; ?>
```

Por diseño, los mensajes flash están destinados a vivir por exactamente una petición (estos "desaparecen en un flash"). Están diseñados para utilizarlos a través de redirecciones exactamente como lo hemos hecho en este ejemplo.

2.5.8 El objeto `Respuesta`

El único requisito para un controlador es que devuelva un objeto `Respuesta`. La clase `Symfony\Component\HttpFoundation\Response` es una abstracción *PHP* en torno a la respuesta *HTTP* - el mensaje de texto, relleno con cabeceras *HTTP* y el contenido que se envía de vuelta al cliente:

```
// crea una simple respuesta con un código de estado 200 (el predeterminado)
$respuesta = new Response('Hola '.$nombre, 200);

// crea una respuesta JSON con un código de estado 200
$respuesta = new Response(json_encode(array('nombre' => $nombre)));
$respuesta->headers->set('Content-Type', 'application/json');
```

Truco: La propiedad `headers` es un objeto `Symfony\Component\HttpFoundation\HeaderBag` con varios métodos útiles para lectura y mutación de las cabeceras del objeto `Respuesta`. Los nombres de las cabeceras están normalizados para que puedas usar `Content-Type` y este sea equivalente a `content-type`, e incluso a `content_type`.

2.5.9 El objeto `Petición`

Además de los valores de los marcadores de posición de enrutado, el controlador también tiene acceso al objeto `Petición` al extender la clase base `Controlador`:

```
$peticion = $this->getRequest();

$peticion->isXmlHttpRequest(); // ¿es una petición Ajax?

$peticion->getPreferredLanguage(array('en', 'es'));

$peticion->query->get('pag'); // consigue un parámetro $_GET

$peticion->request->get('pag'); // consigue un parámetro $_POST
```

Al igual que el objeto `Respuesta`, las cabeceras de la petición se almacenan en un objeto `HeaderBag` y son fácilmente accesibles.

2.5.10 Consideraciones finales

Siempre que creas una página, en última instancia, tendrás que escribir algún código que contenga la lógica para esa página. En *Symfony*, a esto se le llama *controlador*, y es una función *PHP* que puede hacer cualquier cosa que necesites a fin de devolver el objeto `Respuesta` que se entregará al usuario final.

Para facilitarte la vida, puedes optar por extender la clase base `Controller`, la cual contiene atajos a métodos para muchas tareas de control comunes. Por ejemplo, puesto que no deseas poner el código *HTML* en tu controlador, puedes usar el método `render()` para reproducir y devolver el contenido desde una plantilla.

En otros capítulos, veremos cómo puedes usar el controlador para conservar y recuperar objetos desde una base de datos, procesar formularios presentados, manejar el almacenamiento en caché y mucho más.

2.5.11 Aprende más en el recetario

- *Cómo personalizar páginas de error* (Página 277)
- *Cómo definir controladores como servicios* (Página 278)

2.6 Enrutando

Las URL bonitas absolutamente son una necesidad para cualquier aplicación web seria. Esto significa dejar atrás las *URL* feas como `index.php?id_articulo=57` en favor de algo así como `/leer/intro-a-symfony`.

Tener tal flexibilidad es más importante aún. ¿Qué pasa si necesitas cambiar la *URL* de una página de `/blog` a `/noticias`? ¿Cuántos enlaces necesitas cazar y actualizar para hacer el cambio? Si estás utilizando el enrutador de *Symfony*, el cambio es sencillo.

El enrutador de *Symfony2* te permite definir direcciones *URL* creativas que se asignan a diferentes áreas de la aplicación. Al final de este capítulo, serás capaz de:

- Crear rutas complejas asignadas a controladores
- Generar URL dentro de plantillas y controladores
- Cargar recursos enrutando desde el paquete (o en cualquier otro lugar)
- Depurar tus rutas

2.6.1 Enrutador en acción

Una *ruta* es un mapa desde un patrón *URL* hasta un controlador. Por ejemplo, supongamos que deseas adaptar cualquier *URL* como `/blog/mi-post` o `/blog/todo-sobre-symfony` y enviarla a un controlador que puede buscar y reproducir esta entrada del blog. La ruta es simple:

- *YAML*

```
# app/config/routing.yml
blog_show:
    pattern:   /blog/{ficha}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog_show" pattern="/blog/{ficha}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog_show', new Route('/blog/{ficha}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $coleccion;
```

El patrón definido por la ruta `blog_show` actúa como `/blog/*` dónde al comodín se le da el nombre de `ficha`. Para la *URL* `/blog/mi-entrada-del-blog`, la variable `ficha` obtiene un valor de `mi-entrada-del-blog`, que está disponible para usarla en el controlador (sigue leyendo).

El parámetro `_controller` es una clave especial que le dice a *Symfony* qué controlador se debe ejecutar cuando una *URL* coincide con esta ruta. La cadena `_controller` se conoce como el *nombre lógico* (Página 93). Esta sigue un patrón que apunta hacia una clase *PHP* y un método:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme/BlogBundle/Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($ficha)
    {
        $blog = // usa la variable $ficha para consultar la base de datos

        return $this->render('AcmeBlogBundle:Blog:show.html.twig', array(
            'blog' => $blog,
```

```
        ));
    }
}
```

¡Enhorabuena! Acabas de crear tu primera ruta y la conectaste a un controlador. Ahora, cuando visites `/blog/mi-comentario`, el controlador `showAction` será ejecutado y la variable `$ficha` será igual a `mi-comentario`.

Este es el objetivo del enrutador de *Symfony2*: asignar la *URL* de una petición a un controlador. De paso, aprenderás todo tipo de trucos que incluso facilitan la asignación de direcciones *URL* complejas.

2.6.2 Enrutador: bajo el capó

Cuando se hace una petición a tu aplicación, esta contiene una dirección al "recurso" exacto que solicitó el cliente. Esta dirección se conoce como *URL* (o *URI*), y podría ser `/contacto`, `/blog/leeme`, o cualquier otra cosa. Tomemos la siguiente petición *HTTP*, por ejemplo:

```
GET /blog/mi-entrada-del-blog
```

El objetivo del sistema de enrutado de *Symfony2* es analizar esta *URL* y determinar qué controlador se debe ejecutar. Todo el proceso es el siguiente:

1. La petición es manejada por el controlador frontal de *Symfony2* (por ejemplo, `app.php`);
2. El núcleo de *Symfony2* (es decir, el Kernel) pregunta al enrutador que examine la petición;
3. El enrutador busca la *URL* entrante para emparejarla con una ruta específica y devuelve información sobre la ruta, incluyendo el controlador que se debe ejecutar;
4. El núcleo de *Symfony2* ejecuta el controlador, que en última instancia, devuelve un objeto `Respuesta`.

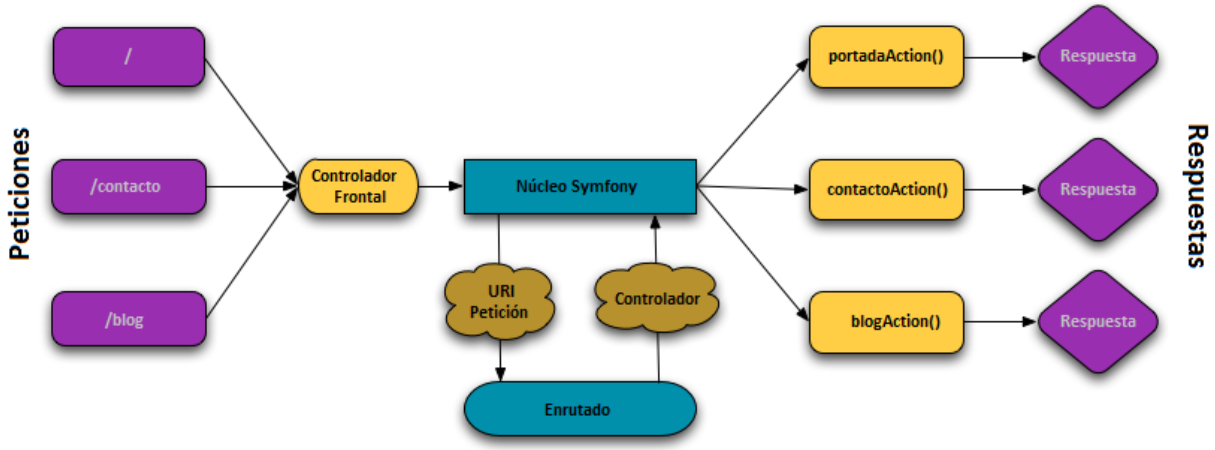


Figura 2.2: La capa del enrutador es una herramienta que traduce la *URL* entrante a un controlador específico a ejecutar.

2.6.3 Creando rutas

Symfony carga todas las rutas de tu aplicación desde un archivo de configuración de enrutado. El archivo usualmente es `app/config/routing.yml`, pero lo puedes configurar para que sea cualquier otro (incluyendo un archivo *XML* o *PHP*) vía el archivo de configuración de la aplicación:

- *YAML*

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

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <!-- ... -->
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('framework', array(
    // ...
    'router'         => array('resource' => '%kernel.root_dir%/config/routing.php'),
));
```

Truco: A pesar de que todas las rutas se cargan desde un solo archivo, es práctica común incluir recursos de enrutado adicionales desde el interior del archivo. Consulta la sección *Incluyendo recursos de enrutado externos* (Página 94) para más información.

2.7 Configuración básica de rutas

Definir una ruta es fácil, y una aplicación típica tendrá un montón de rutas. Una ruta básica consta de dos partes: el `patrón` a coincidir y un arreglo `defaults`:

- *YAML*

```
_bienvenida:
    pattern:   /
    defaults:  { _controller: AcmeDemoBundle:Principal:portada }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="_bienvenida" pattern="/">
        <default key="_controller">AcmeDemoBundle:Principal:portada</default>
    </route>

</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
```

```
$coleccion = new RouteCollection();
$coleccion->add('_bienvenida', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Principal:portada',
)));

return $coleccion;
```

Esta ruta coincide con la página de inicio (`/`) y la asigna al controlador de la página principal `AcmeDemoBundle:Principal:portada`. *Symfony2* convierte la cadena `_controller` en una función *PHP* real y la ejecuta. Este proceso será explicado en breve en la sección *Patrón de nomenclatura para controladores* (Página 93).

2.8 Enrutando con marcadores de posición

Por supuesto, el sistema de enrutado es compatible con rutas mucho más interesantes. Muchas rutas contienen uno o más "comodines" llamados marcadores de posición:

- *YAML*

```
blog_show:
    pattern:   /blog/{ficha}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog_show" pattern="/blog/{ficha}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog_show', new Route('/blog/{ficha}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $coleccion;
```

El patrón coincidirá con cualquier cosa que se vea como `/blog/*`. Aún mejor, el valor coincide con el marcador de posición `{ficha}` que estará disponible dentro de tu controlador. En otras palabras, si la *URL* es `/blog/hola-mundo`, una variable `$ficha`, con un valor de `hola-mundo`, estará disponible en el controlador. Esta se puede usar, por ejemplo, para cargar la entrada en el blog coincidente con esa cadena.

El patrón *no* es, sin embargo, simplemente una coincidencia con `/blog`. Eso es porque, por omisión, todos los marcadores de posición son obligatorios. Esto se puede cambiar agregando un valor marcador de posición al arreglo `defaults`.

2.9 Marcadores de posición obligatorios y opcionales

Para hacer las cosas más emocionantes, añade una nueva ruta que muestre una lista de todas las entradas del 'blog' para la petición imaginaria 'blog':

- *YAML*

```
blog:
    pattern:   /blog
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog', new Route('/blog', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $coleccion;
```

Hasta el momento, esta ruta es tan simple como es posible - no contiene marcadores de posición y sólo coincidirá con la *URL* exacta `/blog`. ¿Pero si necesitamos que esta ruta sea compatible con paginación, donde `/blog/2` muestra la segunda página de las entradas del blog? Actualiza la ruta para que tenga un nuevo marcador de posición `{pag}`:

- *YAML*

```
blog:
    pattern:   /blog/{pag}
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{pag}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog', new Route('/blog/{pag}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $coleccion;
```

Al igual que el marcador de posición `{ficha}` anterior, el valor coincidente con `{pag}` estará disponible dentro de tu controlador. Puedes utilizar su valor para determinar cual conjunto de entradas del blog muestra determinada página.

¡Pero espera! Puesto que los marcadores de posición de forma predeterminada son obligatorios, esta ruta ya no coincidirá con `/blog` simplemente. En su lugar, para ver la página 1 del blog, ¡habrá la necesidad de utilizar la *URL* `/blog/1`! Debido a que esa no es la manera en que se comporta una aplicación web rica, debes modificar la ruta para que el parámetro `{pag}` sea opcional. Esto se consigue incluyendo la colección `defaults`:

- *YAML*

```
blog:
    pattern:   /blog/{pag}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, pag: 1 }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{pag}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="pag">1</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog', new Route('/blog/{pag}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'pag' => 1,
)));

return $coleccion;
```

Agregando `pag` a la clave `defaults`, el marcador de posición `{pag}` ya no es necesario. La *URL* `/blog` coincidirá con esta ruta y el valor del parámetro `pag` se fijará en un `1`. La *URL* `/blog/2` también coincide, dando al parámetro `pag` un valor de `2`. Perfecto.

/blog	{pag} = 1
/blog/1	{pag} = 1
/blog/2	{pag} = 2

2.10 Agregando requisitos

Echa un vistazo a las rutas que hemos creado hasta ahora:

- *YAML*

```
blog:
    pattern:   /blog/{pag}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, pag: 1 }

blog_show:
    pattern:   /blog/{ficha}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{pag}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="pag">1</default>
    </route>

    <route id="blog_show" pattern="/blog/{ficha}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog', new Route('/blog/{pag}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'pag' => 1,
)));

$coleccion->add('blog_show', new Route('/blog/{show}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $coleccion;
```

¿Puedes ver el problema? Ten en cuenta que ambas rutas tienen patrones que coinciden con las *URL* que se parezcan a `/blog/*`. El enrutador de *Symfony* siempre elegirá la **primera** ruta coincidente que encuentre. En otras palabras, la ruta `blog_show` *nunca* corresponderá. En cambio, una *URL* como `/blog/mi-entrada-del-blog` coincidirá con la primera ruta (`blog`) y devolverá un valor sin sentido de `mi-entrada-del-blog` para el parámetro `{pag}`.

URL	ruta	parámetros
/blog/2	blog	{pag} = 2
/blog/mi-entrada-del-blog	blog	{pag} = mi-entrada-del-blog

La respuesta al problema es añadir *requisitos* a la ruta. Las rutas en este ejemplo deben funcionar a la perfección si el patrón `/blog/{pag}` *sólo* concuerda con una URL dónde la parte `{pag}` es un número entero. Afortunadamente,

se puede agregar fácilmente una expresión regular de requisitos para cada parámetro. Por ejemplo:

- *YAML*

```
blog:
    pattern:   /blog/{pag}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, pag: 1 }
    requirements:
        pag:  \d+
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{pag}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="pag">1</default>
        <requirement key="pag">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('blog', new Route('/blog/{pag}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'pag' => 1,
), array(
    'pag' => '\d+',
)));

return $coleccion;
```

El requisito `\d+` es una expresión regular diciendo que el valor del parámetro `{pag}` debe ser un dígito (es decir, un número). La ruta `blog` todavía coincide con una *URL* como `/blog/2` (porque 2 es un número), pero ya no concuerda con una *URL* como `/blog/my-blog-pos` (porque `mi-entrada-del-blog` *no* es un número).

Como resultado, una *URL* como `/blog/mi-entrada-del-blog` ahora coincide correctamente con la ruta `blog_show`.

URL	ruta	parámetros
/blog/2	blog	{pag} = 2
/blog/mi-entrada-del-blog	blog_show	{ficha} = mi-entrada-del-blog

Las primeras rutas siempre ganan

¿Qué significa todo eso de que el orden de las rutas es muy importante? Si la ruta `blog_show` se coloca por encima de la ruta `blog`, la *URL* `/blog/2` coincidiría con `blog_show` en lugar de `blog` ya que el parámetro `{ficha}` de `blog_show` no tiene ningún requisito. Usando el orden adecuado y requisitos claros, puedes lograr casi cualquier cosa.

Puesto que el parámetro `requirements` son expresiones regulares, la complejidad y flexibilidad de cada requisito es totalmente tuya. Supongamos que la página principal de tu aplicación está disponible en dos diferentes idiomas,

según la *URL*:

- *YAML*

```
portada:
    pattern:   /{culture}
    defaults:  { _controller: AcmeDemoBundle:Principal:portada, culture: en }
    requirements:
        culture:  en|es
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="portada" pattern="/{culture}">
        <default key="_controller">AcmeDemoBundle:Principal:portada</default>
        <default key="culture">en</default>
        <requirement key="culture">en|es</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('portada', new Route('/{culture}', array(
    '_controller' => 'AcmeDemoBundle:Principal:portada',
    'culture' => 'en',
), array(
    'culture' => 'en|es',
)));

return $coleccion;
```

Para las peticiones entrantes, la porción `{culture}` de la dirección se compara con la expresión regular `(en|es)`.

/	{culture} = en
/en	{culture} = en
/es	{culture} = es
/fr	*no coincidirá con esta ruta*

2.11 Agregando requisitos de método *HTTP*

Además de la *URL*, también puedes coincidir con el *método* de la petición entrante (es decir, GET, HEAD, POST, PUT, DELETE). Supongamos que tienes un formulario de contacto con dos controladores - uno para mostrar el formulario (en una petición GET) y uno para procesar el formulario una vez presentada (en una petición POST). Esto se puede lograr con la siguiente configuración de ruta:

- *YAML*

```
contacto:
    pattern:  /contacto
```

```
    defaults: { _controller: AcmeDemoBundle:Principal:contacto }
    requirements:
        _method:  GET

contacto_process:
    pattern:  /contacto
    defaults: { _controller: AcmeDemoBundle:Principal:contactoProcess }
    requirements:
        _method:  POST
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="contacto" pattern="/contacto">
        <default key="_controller">AcmeDemoBundle:Principal:contacto</default>
        <requirement key="_method">GET</requirement>
    </route>

    <route id="contact_process" pattern="/contacto">
        <default key="_controller">AcmeDemoBundle:Principal:contactoProcess</default>
        <requirement key="_method">POST</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('contacto', new Route('/contacto', array(
    '_controller' => 'AcmeDemoBundle:Principal:contacto',
), array(
    '_method' => 'GET',
)));

$coleccion->add('contacto_process', new Route('/contacto', array(
    '_controller' => 'AcmeDemoBundle:Principal:contactoProcess',
), array(
    '_method' => 'POST',
)));

return $coleccion;
```

A pesar de que estas dos rutas tienen patrones idénticos (`/contacto`), la primera ruta sólo coincidirá con las peticiones GET y la segunda sólo coincidirá con las peticiones POST. Esto significa que puedes mostrar y enviar el formulario a través de la misma *URL*, mientras usas controladores distintos para las dos acciones.

Nota: Si no especificas el requisito `_method`, la ruta coincidirá con todos los *métodos*.

Al igual que los otros requisitos, el requisito `_method` se analiza como una expresión regular. Para hacer coincidir peticiones `GET` *o* `POST`, puedes utilizar `GET|POST`.

2.12 Ejemplo de enrutado avanzado

En este punto, tienes todo lo necesario para crear una poderosa estructura de enrutado *Symfony*. El siguiente es un ejemplo de cuán flexible puede ser el sistema de enrutado:

- *YAML*

```
articulo_show:
  pattern:  /articulos/{culture}/{year}/{titulo}.{_format}
  defaults: { _controller: AcmeDemoBundle:Articulo:show, _format: html }
  requirements:
      culture:  en|es
      _format:  html|rss
      year:     \d+
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="articulo_show" pattern="/articulos/{culture}/{year}/{titulo}.{_format}">
        <default key="_controller">AcmeDemoBundle:Articulo:show</default>
        <default key="_format">html</default>
        <requirement key="culture">en|es</requirement>
        <requirement key="_format">html|rss</requirement>
        <requirement key="year">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('portada', new Route('/articulos/{culture}/{year}/{titulo}.{_format}', array(
    '_controller' => 'AcmeDemoBundle:Articulo:show',
    '_format' => 'html',
), array(
    'culture' => 'en|es',
    '_format' => 'html|rss',
    'year' => '\d+',
)));

return $coleccion;
```

Como hemos visto, esta ruta sólo coincide si la porción `{culture}` de la *URL* es o bien `en` o `es` y si `{year}` es un número. Esta ruta también muestra cómo puedes utilizar un punto entre los marcadores de posición en lugar de una barra inclinada. Las URL que coinciden con esta ruta podrían ser:

- `/articulos/en/2010/mi-comentario`
- `/articulos/es/2010/mi-comentario.rss`

El parámetro especial de enrutado `_format`

Este ejemplo también resalta el parámetro especial de enrutado `_format`. Cuando se utiliza este parámetro, el valor coincidente se convierte en el "formato de la petición" del objeto `Petición`. En última instancia, el formato de la petición se usa para cosas tales como establecer el `Content-Type` de la respuesta (por ejemplo, un formato de petición `json` se traduce en un `Content-Type` de `application/json`). Este también se puede usar en el controlador para reproducir una plantilla diferente por cada valor de `_format`. El parámetro `_format` es una forma muy poderosa para reproducir el mismo contenido en distintos formatos.

2.13 Parámetros de enrutado especiales

Como hemos visto, cada parámetro de enrutado o valor predeterminado finalmente está disponible como un argumento en el método controlador. Adicionalmente, hay tres parámetros que son especiales: cada uno añade una única pieza de funcionalidad a tu aplicación:

- `_controller`: Como hemos visto, este parámetro se utiliza para determinar qué controlador se ejecuta cuando coincide con la ruta;
- `_format`: Se utiliza para establecer el formato de la petición (*Leer más* (Página 92));
- `_locale`: Se utiliza para establecer la configuración regional en la sesión (*Leer más* (Página 233));

2.13.1 Patrón de nomenclatura para controladores

Cada ruta debe tener un parámetro `_controller`, el cual determina qué controlador se debe ejecutar cuando dicha ruta se empareje. Este parámetro utiliza un patrón de cadena simple llamado el *nombre lógico del controlador*, que *Symfony* asigna a un método y clase *PHP* específico. El patrón consta de tres partes, cada una separada por dos puntos:

paquete:**controlador**:**acción**

Por ejemplo, un valor `_controller` de `AcmeBlogBundle:Blog:show` significa:

Paquete	Clase de controlador	Nombre método
AcmeBlogBundle	BlogController	showAction

El controlador podría tener este aspecto:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($ficha)
    {
        // ...
    }
}
```

Ten en cuenta que *Symfony* añade la cadena `Controller` al nombre de la clase (`Blog` => `BlogController`) y `Action` al nombre del método (`show` => `showAction`).

También podrías referirte a este controlador utilizando su nombre de clase y método completamente cualificado: `Acme\BlogBundle\Controller\BlogController::showAction`. Pero si sigues algunas simples convenciones, el nombre lógico es más conciso y permite mayor flexibilidad.

Nota: Además de utilizar el nombre lógico o el nombre de clase completamente cualificado, *Symfony* es compatible con una tercera forma de referirse a un controlador. Este método utiliza un solo separador de dos puntos (por ejemplo, `service_name:indexAction`) y hace referencia al controlador como un servicio (consulta *Cómo definir controladores como servicios* (Página 278)).

2.13.2 Parámetros de ruta y argumentos del controlador

Los parámetros de ruta (por ejemplo, `{ficha}`) son especialmente importantes porque cada uno se pone a disposición como argumento para el método controlador:

```
public function showAction($ficha)
{
  // ...
}
```

En realidad, toda la colección `defaults` se combina con los valores del parámetro para formar una sola matriz. Cada clave de esa matriz está disponible como un argumento en el controlador.

En otras palabras, por cada argumento de tu método controlador, *Symfony* busca un parámetro de ruta de ese nombre y asigna su valor a ese argumento. En el ejemplo avanzado anterior, cualquier combinación (en cualquier orden) de las siguientes variables se podría utilizar como argumentos para el método `showAction()`:

- `$culture`
- `$year`
- `$titulo`
- `$_format`
- `$_controller`

Dado que los marcadores de posición y los valores de la colección `defaults` se combinan, incluso la variable `$_controller` está disponible. Para una explicación más detallada, consulta *Parámetros de ruta como argumentos para el controlador* (Página 73).

Truco: También puedes utilizar una variable especial `$_route`, que se fija al nombre de la ruta que se emparejó.

2.13.3 Incluyendo recursos de enrutado externos

Todas las rutas se cargan a través de un único archivo de configuración - usualmente `app/config/routing.yml` (consulta Creando rutas (Página 83) más arriba). Por lo general, sin embargo, deseas cargar rutas para otros lugares, como un archivo de enrutado que vive dentro de un paquete. Esto se puede hacer "importando" ese archivo:

- *YAML*

  ```
  # app/config/routing.yml
  acme_hola:
      resource: "@AcmeHolaBundle/Resources/config/routing.yml"
  ```

- *XML*

  ```
  <!-- app/config/routing.xml -->
  <?xml version="1.0" encoding="UTF-8" ?>
  ```

```xml
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHolaBundle/Resources/config/routing.xml" />
</routes>
```

- *PHP*

```php
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;

$coleccion = new RouteCollection();
$coleccion->addCollection($cargador->import("@AcmeHolaBundle/Resources/config/routing.php"));

return $coleccion;
```

Nota: Cuando importas recursos desde *YAML*, la clave (por ejemplo, `acme_hola`) no tiene sentido. Sólo asegúrate de que es única para que no haya otras líneas que reemplazar.

La clave `resource` carga el recurso dado enrutando. En este ejemplo, el recurso es la ruta completa a un archivo, donde la sintaxis contextual del atajo `@AcmeHolaBundle` se resuelve en la ruta a ese paquete. El archivo importado podría tener este aspecto:

- *YAML*

```yaml
 # src/Acme/HolaBundle/Resources/config/routing.yml
acme_hola:
     pattern:  /hola/{nombre}
     defaults: { _controller: AcmeHolaBundle:Hola:index }
```

- *XML*

```xml
<!-- src/Acme/HolaBundle/Resources/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="acme_hola" pattern="/hola/{nombre}">
        <default key="_controller">AcmeHolaBundle:Hola:index</default>
    </route>
</routes>
```

- *PHP*

```php
// src/Acme/HolaBundle/Resources/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('acme_hola', new Route('/Hola/{nombre}', array(
    '_controller' => 'AcmeHolaBundle:Hola:index',
)));

return $coleccion;
```

Las rutas de este archivo se analizan y cargan en la misma forma que el archivo de enrutado principal.

2.14 Prefijando rutas importadas

También puedes optar por proporcionar un "prefijo" para las rutas importadas. Por ejemplo, supongamos que deseas que la ruta `acme_hola` tenga un patrón final de `/admin/hola/{nombre}` en lugar de simplemente `/hola/{nombre}`:

- *YAML*

```
# app/config/routing.yml
acme_hola:
    resource: "@AcmeHolaBundle/Resources/config/routing.yml"
    prefix:   /admin
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHolaBundle/Resources/config/routing.xml" prefix="/admin" />
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;

$coleccion = new RouteCollection();
$coleccion->addCollection($cargador->import("@AcmeHolaBundle/Resources/config/routing.php"), '/a

return $coleccion;
```

La cadena `/admin` ahora se antepondrá al patrón de cada ruta cargada desde el nuevo recurso enrutado.

2.14.1 Visualizando y depurando rutas

Si bien agregar y personalizar rutas, es útil para poder visualizar y obtener información detallada sobre tus rutas. Una buena manera de ver todas las rutas en tu aplicación es a través de la orden de consola `router:debug`. Ejecuta la siguiente orden desde la raíz de tu proyecto.

```
php app/console router:debug
```

Esta orden imprimirá una útil lista de *todas* las rutas configuradas en tu aplicación:

```
portada               ANY       /
contact               GET       /contacto
contact_process       POST      /contacto
articulo_show         ANY       /articulos/{culture}/{year}/{titulo}.{_format}
blog                  ANY       /blog/{pag}
blog_show             ANY       /blog/{ficha}
```

También puedes obtener información muy específica de una sola ruta incluyendo el nombre de la ruta después de la orden:

```
php app/console router:debug articulo_show
```

2.14.2 Generando *URL*

El sistema de enrutado también se debe utilizar para generar direcciones *URL*. En realidad, el enrutado es un sistema bidireccional: asignando la *URL* a un controlador+parámetros y la ruta+parámetros a una *URL*. Los métodos **:method:'Symfony\\Component\\Routing\\Router::match'** y **:method:'Symfony\\Component\\Routing\\Router::generate'** de este sistema bidireccional. Tomando la ruta `blog_show` del ejemplo anterior:

```
$params = $enrutador->match('/blog/mi-entrada-del-blog');
// array('ficha' => 'mi-entrada-del-blog', '_controller' => 'AcmeBlogBundle:Blog:show')

$uri = $enrutador->generate('blog_show', array('ficha' => 'mi-entrada-del-blog'));
// /blog/mi-entrada-del-blog
```

Para generar una *URL*, debes especificar el nombre de la ruta (por ejemplo, `blog_show`) y ningún comodín (por ejemplo, `ficha = mi-entrada-del-blog`) utilizados en el patrón para esa ruta. Con esta información, puedes generar fácilmente cualquier *URL*:

```
class PrincipalController extends Controller
{
    public function showAction($ficha)
    {
      // ...

      $url = $this->get('router')->generate('blog_show', array('ficha' => 'mi-entrada-del-blog'));
    }
}
```

En una sección posterior, aprenderás cómo generar direcciones *URL* desde dentro de plantillas.

2.15 Generando *URL* absolutas

De forma predeterminada, el enrutador va a generar direcciones *URL* relativas (por ejemplo `/blog`). Para generar una *URL* absoluta, sólo tienes que pasar `true` como tercer argumento del método `generate()`:

```
$enrutador->generate('blog_show', array('ficha' => 'mi-entrada-del-blog'), true);
// http://www.ejemplo.com/blog/mi-entrada-del-blog
```

Nota: El host que utiliza al generar una *URL* absoluta es el anfitrión del objeto `Petición` actual. Este, de forma automática, lo detecta basándose en la información del servidor proporcionada por *PHP*. Al generar direcciones *URL* absolutas para archivos desde la línea de ordenes, tendrás que configurar manualmente el anfitrión que desees en el objeto `Petición`:

```
$peticion->headers->set('HOST', 'www.ejemplo.com');
```

2.16 Generando *URL* con cadena de consulta

El método `generate` toma una matriz de valores comodín para generar la *URI*. Pero si pasas adicionales, se añadirán a la *URI* como cadena de consulta:

```
$enrutador->generate('blog', array('pag' => 2, 'categoria' => 'Symfony'));
// /blog/2?categoria=Symfony
```

2.17 Generando *URL* desde una plantilla

El lugar más común para generar una *URL* es dentro de una plantilla cuando creas enlaces entre las páginas de tu aplicación. Esto se hace igual que antes, pero utilizando una función ayudante de plantilla:

- *Twig*

```
<a href="{{ path('blog_show', { 'ficha': 'mi-entrada-del-blog' }) }}">
  Leer esta entrada del blog.
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('blog_show', array('ficha' => 'mi-entrada-del-blog
    Leer esta entrada del blog.
</a>
```

También se pueden generar URL absolutas.

- *Twig*

```
<a href="{{ url('blog_show', { 'ficha': 'mi-entrada-del-blog' }) }}">
  Leer esta entrada del blog.
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('blog_show', array('ficha' => 'mi-entrada-del-blog
    Leer esta entrada del blog.
</a>
```

2.17.1 Resumen

El enrutado es un sistema para asignar la dirección de las peticiones entrantes a la función controladora que se debe llamar para procesar la petición. Este permite especificar ambas direcciones *URL* bonitas y mantiene la funcionalidad de tu aplicación disociada de las direcciones *URL*. El enrutado es un mecanismo de dos vías, lo cual significa que también se debe usar para generar direcciones *URL*.

2.17.2 Aprende más en el recetario

- *Cómo forzar las rutas para utilizar siempre HTTPS* (Página 279)

2.18 Creando y usando plantillas

Como sabes, el *Controlador* (Página 71) es responsable de manejar cada petición entrante en una aplicación *Symfony2*. En realidad, el controlador delega la mayor parte del trabajo pesado a otros lugares para que el código se pueda probar y volver a utilizar. Cuando un controlador necesita generar *HTML*, *CSS* o cualquier otro contenido, que maneje el trabajo fuera del motor de plantillas. En este capítulo, aprenderás cómo escribir potentes plantillas que puedes utilizar para

devolver contenido al usuario, rellenar el cuerpo de correo electrónico y mucho más. Aprenderás métodos abreviados, formas inteligentes para extender las plantillas y cómo reutilizar código de plantilla.

2.18.1 Plantillas

Una plantilla simplemente es un archivo de texto que puede generar cualquier formato basado en texto (*HTML*, *XML*, *CSV*, *LaTeX*...). El tipo de plantilla más familiar es una plantilla *PHP* - un archivo de texto interpretado por *PHP* que contiene una mezcla de texto y código *PHP*:

```
<!DOCTYPE html>
<html>
    <head>
        <title>¡Bienvenido a Symfony!</title>
    </head>
    <body>
        <h1><?php echo $titulo_pag ?></h1>

        <ul id="navegacion">
            <?php foreach ($navegacion as $elemento): ?>
                <li>
                    <a href="<?php echo $elemento->getHref() ?>">
                        <?php echo $elemento->getCaption() ?>
                    </a>
                </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

Pero *Symfony2* contiene un lenguaje de plantillas aún más potente llamado Twig. *Twig* te permite escribir plantillas concisas y fáciles de leer que son más amigables para los diseñadores web y, de varias maneras, más poderosas que las plantillas *PHP*:

```
<!DOCTYPE html>
<html>
    <head>
        <title>¡Bienvenido a Symfony!</title>
    </head>
    <body>
        <h1>{{ titulo_pag }}</h1>

        <ul id="navegacion">
            {% for elemento in navegacion %}
                <li><a href="{{ elemento.href }}">{{ elemento.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Twig define dos tipos de sintaxis especial:

- `{{ ... }}`: "Dice algo": imprime una variable o el resultado de una expresión a la plantilla;
- `{% ... %}`: "Hace algo": una **etiqueta** que controla la lógica de la plantilla; se utiliza para ejecutar declaraciones, como bucles `for`, por ejemplo.

Nota: Hay una tercer sintaxis utilizada para crear comentarios: `{# esto es un comentario #}`. Esta sintaxis se puede utilizar en múltiples líneas como la sintaxis `/* comentario */` equivalente de *PHP*.

Twig también contiene **filtros**, los cuales modifican el contenido antes de reproducirlo. El siguiente fragmento convierte a mayúsculas la variable `titulo` antes de reproducirla:

```
{{ titulo | upper }}
```

Twig viene con una larga lista de etiquetas y filtros que están disponibles de forma predeterminada. Incluso puedes agregar tus propias extensiones a *Twig*, según sea necesario.

Truco: Registrar una extensión de *Twig* es tan fácil como crear un nuevo servicio y etiquetarlo con las *etiquetas* (Página 251) `twig.extension`.

Como verás en toda la documentación, *Twig* también es compatible con funciones y fácilmente puedes añadir nuevas funciones. Por ejemplo, la siguiente función, utiliza una etiqueta `for` estándar y la función `cycle` para imprimir diez etiquetas div, alternando entre clases `par` e `impar`:

```
{% for i in 0..10 %}
  <div class="{{ cycle(['par', 'impar'], i) }}">
    <!-- algún HTML aquí -->
  </div>
{% endfor %}
```

A lo largo de este capítulo, mostraremos las plantillas de ejemplo en ambos formatos *Twig* y *PHP*.

¿Porqué *Twig*?

Las plantillas *Twig* están destinadas a ser simples y no procesar etiquetas PHP. Esto es por diseño: el sistema de plantillas *Twig* está destinado a expresar la presentación, no la lógica del programa. Cuanto más utilices *Twig*, más apreciarás y te beneficiarás de esta distinción. Y, por supuesto, todos los diseñadores web las amarán.
Twig también puede hacer cosas que *PHP* no puede, como heredar verdaderas plantillas (las plantillas *Twig* se compilan hasta clases *PHP* que se heredan unas a otras), controlar los espacios en blanco, restringir un ambiente para prácticas, e incluir funciones y filtros personalizados que sólo afectan a las plantillas. *Twig* contiene características que facilitan la escritura de plantillas y estas son más concisas. Tomemos el siguiente ejemplo, el cual combina un bucle con una declaración `if` lógica:

```
<ul>
    {% for usuario in usuarios %}
        <li>{{ usuario.nombreusuario }}</li>
    {% else %}
        <li>¡No hay usuarios!</li>
    {% endfor %}
</ul>
```

Guardando en caché plantillas *Twig*

Twig es rápido. Cada plantilla *Twig* se compila hasta una clase *PHP* nativa que se reproduce en tiempo de ejecución. Las clases compiladas se encuentran en el directorio `app/cache/{entorno}/twig` (donde `{entorno}` es el entorno, tal como `dev` o `prod`) y, en algunos casos, pueden ser útiles mientras depuras. Consulta la sección *Entornos* (Página 69) para más información sobre los entornos.

Cuando está habilitado el modo `debug` (comúnmente en el entorno `dev`) al realizar cambios a una plantilla *Twig*, esta se vuelve a compilar automáticamente. Esto significa que durante el desarrollo, felizmente, puedes realizar cambios en una plantilla *Twig* e inmediatamente ver las modificaciones sin tener que preocuparte de limpiar ninguna caché.

Cuando el modo `debug` está desactivado (comúnmente en el entorno `prod`), sin embargo, debes borrar el directorio de caché para regenerar las plantillas. Recuerda hacer esto al desplegar tu aplicación.

2.18.2 Plantillas, herencia y diseño

A menudo, las plantillas en un proyecto comparten elementos comunes, como el encabezado, pie de página, barra lateral o más. En *Symfony2*, nos gusta pensar en este problema de forma diferente: una plantilla se puede decorar con otra. Esto funciona exactamente igual que las clases *PHP*: la herencia de plantillas nos permite crear un "diseño" de plantilla base que contiene todos los elementos comunes de tu sitio definidos como **bloques** (piensa en "clases *PHP* con métodos base"). Una plantilla hija puede extender el diseño base y reemplazar cualquiera de sus bloques (piensa en las "subclases *PHP* que sustituyen determinados métodos de su clase padre").

En primer lugar, crea un archivo con tu diseño base:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block titulo %}Aplicación de prueba{% endblock %}</title>
    </head>
    <body>
        <div id="barralateral">
            {% block barralateral %}
            <ul>
                <li><a href="/">Portada</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
            {% endblock %}
        </div>

        <div id="contenido">
            {% block cuerpo %}{% endblock %}
        </div>
    </body>
</html>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->salida('titulo', 'Aplicación de prueba') ?></title>
    </head>
    <body>
        <div id="barralateral">
            <?php if ($view['slots']->has('barralateral')): ?>
                <?php $view['slots']->salida('barralateral') ?>
            <?php else: ?>
                <ul>
                    <li><a href="/">Portada</a></li>
                    <li><a href="/blog">Blog</a></li>
                </ul>
            <?php endif; ?>
```

```
        </div>

        <div id="contenido">
            <?php $view['slots']->salida('body') ?>
        </div>
    </body>
</html>
```

Nota: Aunque la explicación sobre la herencia de plantillas será en términos de *Twig*, la filosofía es la misma entre plantillas *Twig* y *PHP*.

Esta plantilla define el esqueleto del documento *HTML* base de una simple página de dos columnas. En este ejemplo, se definen tres áreas `{% block %}` (`titulo`, `barralateral` y `body`). Una plantilla hija puede sustituir cada uno de los bloques o dejarlos con su implementación predeterminada. Esta plantilla también se podría reproducir directamente. En este caso, los bloques `titulo`, `barralateral` y `body` simplemente mantienen los valores predeterminados usados en esta plantilla.

Una plantilla hija podría tener este aspecto:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block titulo %}Mis interesantes entradas del blog{% endblock %}

{% block cuerpo %}
    {% for entrada in entradas_blog %}
        <h2>{{ entrada.titulo }}</h2>
        <p>{{ entrada.cuerpo }}</p>
    {% endfor %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/index.html.php -->
<?php $view->extend('::base.html.php') ?>

<?php $view['slots']->set('titulo', 'Mis interesantes entradas del blog') ?>

<?php $view['slots']->start('body') ?>
    <?php foreach ($entradas_blog as $entrada): ?>
        <h2><?php echo $entrada->getTitulo() ?></h2>
        <p><?php echo $entrada->getCuerpo() ?></p>
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

Nota: La plantilla padre se identifica mediante una sintaxis de cadena especial (`::base.html.twig`) la cual indica que la plantilla vive en el directorio `app/Resources/views` del proyecto. Esta convención de nomenclatura se explica completamente en *Nomenclatura y ubicación de plantillas* (Página 103).

La clave para la herencia de plantillas es la etiqueta `{% extends %}`. Esto le indica al motor de plantillas que primero evalúe la plantilla base, la cual establece el diseño y define varios bloques. Luego reproduce la plantilla hija, en ese momento, los bloques `titulo` y `body` del padre son reemplazados por los de la hija. Dependiendo del valor de `entradas_blog`, el resultado sería algo como esto:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>Mis interesantes entradas del blog</title>
    </head>
    <body>
        <div id="barralateral">
            <ul>
                <li><a href="/">Portada</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
        </div>

        <div id="contenido">
            <h2>Mi primer mensaje</h2>
            <p>El cuerpo del primer mensaje.</p>

            <h2>Otro mensaje</h2>
            <p>El cuerpo del segundo mensaje.</p>
        </div>
    </body>
</html>
```

Ten en cuenta que como en la plantilla hija no has definido un bloque `barralateral`, en su lugar, se utiliza el valor de la plantilla padre. Una plantilla padre, de forma predeterminada, siempre utiliza una etiqueta `{ % block %}` para el contenido.

Puedes utilizar tantos niveles de herencia como quieras. En la siguiente sección, explicaremos un modelo común de tres niveles de herencia junto con la forma en que se organizan las plantillas dentro de un proyecto *Symfony2*.

Cuando trabajes con la herencia de plantillas, ten en cuenta los siguientes consejos:

- Si utilizas `{ % extends %}` en una plantilla, esta debe ser la primer etiqueta en esa plantilla.
- Mientras más etiquetas `{ % block %}` tengas en tu plantilla base, mejor. Recuerda, las plantillas hijas no tienen que definir todos los bloques de los padres, por lo tanto crea tantos bloques en tus plantillas base como desees y dale a cada uno un valor predeterminado razonable. Mientras más bloques tengan tus plantillas base, más flexible será tu diseño.
- Si te encuentras duplicando contenido en una serie de plantillas, probablemente significa que debes mover el contenido a un `{ % block %}` en una plantilla padre. En algunos casos, una mejor solución podría ser mover el contenido a una nueva plantilla e incluirla con `include` (consulta *Incluyendo otras plantillas* (Página 105)).
- Si necesitas conseguir el contenido de un bloque de la plantilla padre, puedes usar la función `{{ parent() }}`. Esta es útil si deseas añadir contenido a un bloque padre en vez de reemplazarlo por completo:

```
{% block barralateral %}
  <h3>Tabla de contenido</h3>
  ...
  {{ parent() }}
{% endblock %}
```

2.18.3 Nomenclatura y ubicación de plantillas

De forma predeterminada, las plantillas pueden vivir en dos diferentes lugares:

- `app/Resources/views/` El directorio de las `vistas` de la aplicación puede contener toda las plantillas base de la aplicación (es decir, los diseños de tu aplicación), así como plantillas que sustituyen a plantillas de

paquetes (consulta *Sustituyendo plantillas del paquete* (Página 111));

- `ruta/al/paquete/Resources/views/` Cada paquete contiene sus plantillas en su directorio `Resources/views` (y subdirectorios). La mayoría de las plantillas viven dentro de un paquete.

Symfony2 utiliza una sintaxis de cadena **paquete**:**controlador**:**plantilla** para las plantillas. Esto permite diferentes tipos de plantilla, dónde cada una vive en un lugar específico:

- `AcmeBlogBundle:Blog:index.html.twig`: Esta sintaxis se utiliza para especificar una plantilla para una página específica. Las tres partes de la cadena, cada una separada por dos puntos (`:`), significan lo siguiente:
  - `AcmeBlogBundle`: (*paquete*) la plantilla vive dentro de `AcmeBlogBundle` (por ejemplo, `src/Acme/BlogBundle`);
  - `Blog`: (*controlador*) indica que la plantilla vive dentro del subdirectorio `Blog` de `Resources/views`;
  - `index.html.twig`: (*plantilla*) el nombre real del archivo es `index.html.twig`.

  Suponiendo que `AcmeBlogBundle` vive en `src/Acme/BlogBundle`, la ruta final para el diseño debería ser *src/Acme/BlogBundle/Resources/views/Blog/index.html.twig'*.
- `AcmeBlogBundle::base.html.twig`: Esta sintaxis se refiere a una plantilla base que es específica para `AcmeBlogBundle`. Puesto que falta la porción central, "controlador", (por ejemplo, `Blog`), la plantilla vive en `Resources/views/base.html.twig` dentro de `AcmeBlogBundle`.
- `::base.html.twig`: Esta sintaxis se refiere a una plantilla o diseño base de la aplicación. Observa que la cadena comienza con dobles dos puntos (`::`), lo cual significa que faltan ambas porciones *paquete* y *controlador*. Esto significa que la plantilla no se encuentra en ningún paquete, sino en el directorio raíz de la aplicación `app/Resources/views/`.

En la sección *Sustituyendo plantillas del paquete* (Página 111), encontrarás cómo puedes sustituir cada plantilla que vive dentro de `AcmeBlogBundle`, por ejemplo, colocando una plantilla del mismo nombre en el directorio `app/Resources/AcmeBlog/views/`. Esto nos da el poder para sustituir plantillas de cualquier paquete de terceros.

Truco: Esperemos que la sintaxis de nomenclatura de plantilla te resulte familiar - es la misma convención de nomenclatura utilizada para referirse al *Patrón de nomenclatura para controladores* (Página 93).

Sufijo de plantilla

El formato **paquete**:**controlador**:**plantilla** de cada plantilla, especifica *dónde* se encuentra el archivo de plantilla. Cada nombre de plantilla también cuenta con dos extensiones que especifican el *formato* y *motor* de esa plantilla.

- **AcmeBlogBundle:Blog:index.html.twig** - formato *HTML*, motor *Twig*
- **AcmeBlogBundle:Blog:index.html.php** - formato *HTML*, motor *PHP*
- **AcmeBlogBundle:Blog:index.css.twig** - formato *CSS*, motor *Twig*

De forma predeterminada, cualquier plantilla *Symfony2* se puede escribir en *Twig* o PHP, y la última parte de la extensión (por ejemplo `.twig` o `.php`) especifica cuál de los dos *motores* se debe utilizar. La primera parte de la extensión, (por ejemplo `.html`, `.css`, etc.) es el formato final que la plantilla debe generar. A diferencia del motor, el cual determina cómo analiza *Symfony2* la plantilla, esta simplemente es una táctica de organización utilizada en caso de que el mismo recurso se tenga que reproducir como *HTML* (`index.html.twig`), XML (`index.xml.twig`), o cualquier otro formato. Para más información, lee la sección *Formato de plantillas* (Página 114).

Nota: Los "motores" disponibles se pueden configurar e incluso agregar nuevos motores. Consulta *Configuración de plantillas* (Página 110) para más detalles.

2.18.4 Etiquetas y ayudantes

Ya entendiste los conceptos básicos de las plantillas, cómo son denominadas y cómo utilizar la herencia en plantillas. Las partes más difíciles ya quedaron atrás. En esta sección, aprenderás acerca de un amplio grupo de herramientas disponibles para ayudarte a realizar las tareas de plantilla más comunes, como la inclusión de otras plantillas, enlazar páginas e incluir imágenes.

Symfony2 viene con varias etiquetas *Twig* especializadas y funciones que facilitan la labor del diseñador de la plantilla. En PHP, el sistema de plantillas extensible ofrece un sistema de *ayudantes* que proporciona funciones útiles en el contexto de la plantilla.

Ya hemos visto algunas etiquetas integradas en *Twig* (`{ % block %}` y `{ % extends %}`), así como un ejemplo de un ayudante *PHP* (consulta `$view['slot']`). Aprendamos un poco más...

Incluyendo otras plantillas

A menudo querrás incluir la misma plantilla o fragmento de código en varias páginas diferentes. Por ejemplo, en una aplicación con "artículos de noticias", el código de la plantilla que muestra un artículo se puede utilizar en la página de detalles del artículo, en una página que muestra los artículos más populares, o en una lista de los últimos artículos.

Cuando necesites volver a utilizar un trozo de código *PHP*, normalmente mueves el código a una nueva clase o función *PHP*. Lo mismo es cierto para las plantillas. Al mover el código de la plantilla a su propia plantilla, este se puede incluir en cualquier otra plantilla. En primer lugar, crea la plantilla que tendrás que volver a usar.

- *Twig*

```
{# src/Acme/ArticuloBundle/Resources/views/Articulo/articuloDetalles.html.twig #}
<h1>{{ articulo.titulo }}</h1>
<h3 class="lineapor">por {{ articulo.nombreAutor }}</h3>

<p>
  {{ articulo.cuerpo }}
</p>
```

- *PHP*

```
<!-- src/Acme/ArticuloBundle/Resources/views/Articulo/articuloDetalles.html.php -->
<h2><?php echo $articulo->getTitulo() ?></h2>
<h3 class="lineapor">por <?php echo $articulo->getNombreAutor() ?></h3>

<p>
  <?php echo $articulo->getCuerpo() ?>
</p>
```

Incluir esta plantilla en cualquier otra plantilla es sencillo:

- *Twig*

```
{# src/Acme/ArticuloBundle/Resources/Articulo/lista.html.twig #}
{% extends 'AcmeArticuloBundle::base.html.twig' %}

{% block cuerpo %}
    <h1>Artículos recientes<h1>

    {% for articulo in articulos %}
        {% include 'AcmeArticuloBundle:Articulo:articuloDetalles.html.twig' with {'articulo': ar
    {% endfor %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/ArticuloBundle/Resources/Articulo/lista.html.php -->
<?php $view->extend('AcmeArticuloBundle::base.html.php') ?>

<?php $view['slots']->start('body') ?>
    <h1>Artículos recientes</h1>

    <?php foreach ($articulos as $articulo): ?>
        <?php echo $view->render('AcmeArticuloBundle:Articulo:articuloDetalles.html.php', array(
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

La plantilla se incluye con la etiqueta `{% include %}`. Observa que el nombre de la plantilla sigue la misma convención típica. La plantilla `articuloDetalles.html.twig` utiliza una variable `articulo`. Esta es proporcionada por la plantilla `lista.html.twig` utilizando la orden `with`.

Truco: La sintaxis `{'articulo': articulo}` es la sintaxis estándar de *Twig* para asignar `hash` (es decir, una matriz con claves nombradas). Si tuviéramos que pasar varios elementos, se vería así: `{'foo': foo, 'bar': bar}`.

Incrustando controladores

En algunos casos, es necesario hacer algo más que incluir una simple plantilla. Supongamos que en tu diseño tienes una barra lateral, la cual contiene los tres artículos más recientes. Recuperar los tres artículos puede incluir consultar la base de datos o realizar otra pesada lógica que no se puede hacer desde dentro de una plantilla.

La solución es simplemente insertar el resultado de un controlador en tu plantilla entera. En primer lugar, crea un controlador que reproduzca un cierto número de artículos recientes:

```
// src/Acme/ArticuloBundle/Controller/ArticuloController.php

class ArticuloController extends Controller
{
    public function arcticulosRecientesAction($max = 3)
    {
        // hace una llamada a la base de datos u otra lógica para obtener los "$max" artículos más re
        $articulos = ...;

        return $this->render('AcmeArticuloBundle:Articulo:listaRecientes.html.twig', array('articulos
    }
}
```

La plantilla `listaRecientes` es perfectamente clara:

- *Twig*

```
{# src/Acme/ArticuloBundle/Resources/views/Articulo/listaRecientes.html.twig #}
{% for articulo in articulos %}
  <a href="/articulo/{{ articulo.ficha }}">
      {{ articulo.titulo }}
  </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticuloBundle/Resources/views/Articulo/listaRecientes.html.php -->
<?php foreach ($articulos in $articulo): ?>
    <a href="/articulo/<?php echo $articulo->getFicha() ?>">
        <?php echo $articulo->getTitulo() ?>
    </a>
<?php endforeach; ?>
```

Nota: Ten en cuenta que en este ejemplo hemos falsificado y codificado la *URL* del artículo (por ejemplo `/articulo/ficha`). Esta es una mala práctica. En la siguiente sección, aprenderás cómo hacer esto correctamente.

Para incluir el controlador, tendrás que referirte a él utilizando la sintaxis de cadena estándar para controladores (es decir, **paquete**:**controlador**:**acción**):

- *Twig*

```
{# app/Resources/views/base.html.twig #}
...

<div id="barralateral">
    {% render "AcmeArticuloBundle:Articulo:articulosRecientes" with {'max': 3} %}
</div>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
...

<div id="barralateral">
    <?php echo $view['actions']->render('AcmeArticuloBundle:Articulo:articulosRecientes', array(
</div>
```

Cada vez que te encuentres necesitando una variable o una pieza de información a la que una plantilla no tiene acceso, considera reproducir un controlador. Los controladores se ejecutan rápidamente y promueven la buena organización y reutilización de código.

Enlazando páginas

La creación de enlaces a otras páginas en tu aplicación es uno de los trabajos más comunes de una plantilla. En lugar de codificar las direcciones *URL* en las plantillas, utiliza la función `path` de *Twig* (o el ayudante `router` en *PHP*) para generar direcciones *URL* basadas en la configuración de enrutado. Más tarde, si deseas modificar la *URL* de una página en particular, todo lo que tienes que hacer es cambiar la configuración de enrutado, las plantillas automáticamente generarán la nueva *URL*.

En primer lugar, crea el enlace a la página "*_bienvenida*", la cual es accesible a través de la siguiente configuración de enrutado:

- *YAML*

```
_bienvenida:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Bienvenida:index }
```

- *XML*

```
<route id="_bienvenida" pattern="/">
    <default key="_controller">AcmeDemoBundle:Bienvenida:index</default>
</route>
```

- *PHP*

```
$coleccion = new RouteCollection();
$coleccion->add('_bienvenida', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Bienvenida:index',
)));

return $coleccion;
```

Para enlazar a la página, sólo tienes que utilizar la función `path` de *Twig* y referir la ruta:

- *Twig*

```
<a href="{{ path('_bienvenida') }}">Portada</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('_bienvenida') ?>">Portada</a>
```

Como era de esperar, esto genera la *URL* `/`. Vamos a ver cómo funciona esto con una ruta más complicada:

- *YAML*

```
articulo_show:
    pattern:  /articulo/{ficha}
    defaults: { _controller: AcmeArticuloBundle:Articulo:show }
```

- *XML*

```
<route id="articulo_show" pattern="/articulo/{ficha}">
    <default key="_controller">AcmeArticuloBundle:Articulo:show</default>
</route>
```

- *PHP*

```
$coleccion = new RouteCollection();
$coleccion->add('articulo_show', new Route('/articulo/{ficha}', array(
    '_controller' => 'AcmeArticuloBundle:Articulo:show',
)));

return $coleccion;
```

En este caso, es necesario especificar el nombre de la ruta (`articulo_show`) y un valor para el parámetro `{ficha}`. Usando esta ruta, vamos a volver a la plantilla `listaRecientes` de la sección anterior y enlazar los artículos correctamente:

- *Twig*

```
{# src/Acme/ArticuloBundle/Resources/views/Articulo/listaRecientes.html.twig #}
{% for articulo in articulos %}
  <a href="{{ path('articulo_show', { 'ficha': articulo.ficha }) }}">
      {{ articulo.titulo }}
  </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticuloBundle/Resources/views/Articulo/listaRecientes.html.php -->
<?php foreach ($articulos in $articulo): ?>
    <a href="<?php echo $view['router']->generate('articulo_show', array('ficha' => $articulo->g
        <?php echo $articulo->getTitulo() ?>
```

```
        </a>
    <?php endforeach; ?>
```

Truco: También puedes generar una *URL* absoluta utilizando la función `url` de *Twig*:

```
<a href="{{ url('_bienvenida') }}">Portada</a>
```

Lo mismo se puede hacer en plantillas *PHP* pasando un tercer argumento al método `generate()`:

```
<a href="<?php echo $view['router']->generate('_bienvenida', array(), true) ?>">Portada</a>
```

Enlazando activos

Las plantillas también se refieren comúnmente a imágenes, *JavaScript*, hojas de estilo y otros activos. Por supuesto, puedes codificar la ruta de estos activos (por ejemplo `/images/logo.png`), pero *Symfony2* ofrece una opción más dinámica a través de la función `assets` de *Twig*:

- *Twig*

  ```
  <img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

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

- *PHP*

  ```
  <img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="Symfony!" />

  <link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/c
  ```

El propósito principal de la función `asset` es hacer más portátil tu aplicación. Si tu aplicación vive en la raíz de tu anfitrión (por ejemplo, http://ejemplo.com), entonces las rutas reproducidas deben ser `/images/logo.png`. Pero si tu aplicación vive en un subdirectorio (por ejemplo, http://ejemplo.com/mi_aplic), cada ruta de activo debe reproducir el subdirectorio (por ejemplo `/mi_aplic/images/logo.png`). La función `asset` se encarga de esto determinando cómo se está utilizando tu aplicación y generando las rutas correctas en consecuencia.

2.18.5 Incluyendo hojas de estilo y *JavaScript* en *Twig*

Ningún sitio estaría completo sin incluir archivos de *JavaScript* y hojas de estilo. En *Symfony*, la inclusión de estos activos se maneja elegantemente, aprovechando la herencia de plantillas de *Symfony*.

Truco: Esta sección te enseñará la filosofía detrás de la inclusión de activos como hojas de estilo y *Javascript* en *Symfony*. *Symfony* también empaca otra biblioteca, llamada `assetic`, la cual sigue esta filosofía, pero te permite hacer cosas mucho más interesantes con esos activos. Para más información sobre el uso de `assetic` consulta *Cómo utilizar Assetic para gestionar activos* (Página 281).

Comienza agregando dos bloques a la plantilla base que mantendrá tus activos: uno llamado `stylesheet` dentro de la etiqueta `head` y otro llamado `javascript` justo por encima de la etiqueta de cierre `body`. Estos bloques deben contener todas las hojas de estilo y archivos *Javascript* que necesitas en tu sitio:

```
{# 'app/Resources/views/base.html.twig' #}
<html>
    <head>
        {# ... #}
```

```
        {% block stylesheets %}
            <link href="{{ asset('/css/main.css') }}" type="text/css" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('/js/main.js') }}" type="text/javascript"></script>
        {% endblock %}
    </body>
</html>
```

¡Eso es bastante fácil! Pero ¿y si es necesario incluir una hoja de estilo extra o archivos *Javascript* desde una plantilla hija? Por ejemplo, supongamos que tienes una página de contacto y necesitas incluir una hoja de estilo `contacto.css` *sólo* en esa página. Desde dentro de la plantilla de la página de contacto, haz lo siguiente:

```
{# src/Acme/DemoBundle/Resources/views/Contacto/contacto.html.twig #}
{# extends '::base.html.twig' #}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('/css/contacto.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# ... #}
```

En la plantilla hija, sólo tienes que reemplazar el bloque `stylesheet` y poner tu nueva etiqueta de hoja de estilo dentro de ese bloque. Por supuesto, debido a que deseas añadir al contenido del bloque padre (y no *cambiarlo* en realidad), debes usar la función `parent()` de *Twig* para incluir todo, desde el bloque `stylesheet` de la plantilla base.

El resultado final es una página que incluye ambas hojas de estilo `main.css` y `contacto.css`.

2.18.6 Configurando y usando el servicio `plantilla`

El corazón del sistema de plantillas en *Symfony2* es el `motor` de plantillas. Este objeto especial es el encargado de reproducir las plantillas y devolver su contenido. Cuando reproduces una plantilla en un controlador, por ejemplo, en realidad estás usando el motor del servicio de plantillas. Por ejemplo:

```
return $this->render('AcmeArticuloBundle:Articulo:index.html.twig');
```

es equivalente a

```
$motor = $this->contenedor->get('templating');
$contenido = $motor->render('AcmeArticuloBundle:Articulo:index.html.twig');

return $respuesta = new Response($contenido);
```

El motor de plantillas (o "servicio") está configurado para funcionar automáticamente al interior de *Symfony2*. Por supuesto, puedes configurar más en el archivo de configuración de la aplicación:

- *YAML*

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

```
    # ...
    templating: { engines: ['twig'] }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:templating>
    <framework:engine id="twig" />
</framework:templating>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig'),
    ),
));
```

Disponemos de muchas opciones de configuración y están cubiertas en el *Apéndice Configurando* (Página 443).

Nota: En el motor de `twig` es obligatorio el uso del `webprofiler` (así como muchos otros paquetes de terceros).

2.18.7 Sustituyendo plantillas del paquete

La comunidad de *Symfony2* se enorgullece de crear y mantener paquetes de alta calidad (consulta Symfony2Bundles.org) para una gran cantidad de diferentes características. Una vez que utilizas un paquete de terceros, probablemente necesites redefinir y personalizar una o más de sus plantillas.

Supongamos que hemos incluido el paquete imaginario `AcmeBlogBundle` de código abierto en el proyecto (por ejemplo, en el directorio `src/Acme/BlogBundle`). Y si bien estás muy contento con todo, deseas sustituir la página "lista" del blog para personalizar el marcado específicamente para tu aplicación. Al excavar en el controlador del `Blog` de `AcmeBlogBundle`, encuentras lo siguiente:

```
public function indexAction()
{
    $blogs = // cierta lógica para recuperar las entradas

    $this->render('AcmeBlogBundle:Blog:index.html.twig', array('blogs' => $blogs));
}
```

Al reproducir `AcmeBlogBundle:Blog:index.html.twig`, en realidad *Symfony2* busca la plantilla en dos diferentes lugares:

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

Para sustituir la plantilla del paquete, sólo tienes que copiar la plantilla `index.html.twig` del paquete a `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (el directorio `app/Resources/AcmeBlogBundle` no existe, por lo tanto tendrás que crearlo). Ahora eres libre de personalizar la plantilla para tu aplicación.

Esta lógica también aplica a las plantillas base del paquete. Supongamos también que cada plantilla en `AcmeBlogBundle` hereda de una plantilla base llamada `AcmeBlogBundle::base.html.twig`. Al igual que antes, *Symfony2* buscará la plantilla en los dos siguientes lugares:

1. app/Resources/AcmeBlogBundle/views/base.html.twig
2. src/Acme/BlogBundle/Resources/views/base.html.twig

Una vez más, para sustituir la plantilla, sólo tienes que copiarla desde el paquete a app/Resources/AcmeBlogBundle/views/base.html.twig. Ahora estás en libertad de personalizar esta copia como mejor te parezca.

Si retrocedes un paso, verás que *Symfony2* siempre empieza a buscar una plantilla en el directorio app/Resources/{NOMBRE_PAQUETE}/views/. Si la plantilla no existe allí, continúa buscando dentro del directorio Resources/views del propio paquete. Esto significa que todas las plantillas del paquete se pueden sustituir colocándolas en el subdirectorio app/Resources correcto.

Sustituyendo plantillas del núcleo

Puesto que la plataforma *Symfony2* en sí misma sólo es un paquete, las plantillas del núcleo se pueden sustituir de la misma manera. Por ejemplo, el núcleo de TwigBundle contiene una serie de de plantillas de "excepción" y "error" diferentes que puedes sustituir copiando cada uno del directorio Resources/views/Exception del TwigBundle a, adivinaste, el directorio app/Resources/TwigBundle/views/Exception.

2.18.8 Herencia de tres niveles

Una manera común de usar la herencia es utilizar un enfoque de tres niveles. Este método funciona a la perfección con los tres diferentes tipos de plantillas que acabamos de cubrir:

- Crea un archivo app/Resources/views/base.html.twig que contenga el diseño principal para tu aplicación (como en el ejemplo anterior). Internamente, esta plantilla se llama ::base.html.twig;
- Crea una plantilla para cada "sección" de tu sitio. Por ejemplo, un AcmeBlogBundle, tendría una plantilla llamada AcmeBlogBundle::base.html.twig que sólo contiene los elementos específicos de la sección blog;

```
{# src/Acme/BlogBundle/Resources/views/base.html.twig #}
{% extends '::base.html.twig' %}

{% block cuerpo %}
    <h1>Aplicación Blog</h1>

    {% block contenido %}{% endblock %}
{% endblock %}
```

- Crea plantillas individuales para cada página y haz que cada una extienda la plantilla de la sección adecuada. Por ejemplo, la página "index" se llama algo parecido a AcmeBlogBundle:Blog:index.html.twig y lista las entradas del blog real.

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends 'AcmeBlogBundle::base.html.twig' %}

{% block contenido %}
    {% for entrada in entradas_blog %}
        <h2>{{ entrada.titulo }}</h2>
        <p>{{ entrada.cuerpo;/p>
    {% endfor %}
{% endblock %}
```

Ten en cuenta que esta plantilla extiende la plantilla de la sección - (AcmeBlogBundle::base.html.twig), que a su vez, extiende el diseño base de la aplicación (::base.html.twig). Este es el modelo común de la herencia de tres niveles.

Cuando construyas tu aplicación, podrás optar por este método o, simplemente, hacer que cada plantilla de página extienda directamente la plantilla base de tu aplicación (por ejemplo, `{% extends '::base.html.twig' %}`). El modelo de tres plantillas es un método de las buenas prácticas utilizadas por los paquetes de proveedores a fin de que la plantilla base de un paquete se pueda sustituir fácilmente para extender correctamente el diseño base de tu aplicación.

2.18.9 Mecanismo de escape

Cuando generas *HTML* a partir de una plantilla, siempre existe el riesgo de que una variable de plantilla pueda producir *HTML* involuntario o código peligroso de lado del cliente. El resultado es que el contenido dinámico puede romper el código *HTML* de la página resultante o permitir a un usuario malicioso realizar un ataque de Explotación de vulnerabilidades del sistema (*Cross Site Scripting XSS*). Considera este ejemplo clásico:

- *Twig*

  ```
  Hola {{ nombre }}
  ```

- *PHP*

  ```
  Hola <?php echo $nombre ?>!
  ```

Imagina que el usuario introduce el siguiente código como su nombre:

```
<script>alert('hola!')</script>
```

Sin ningún tipo de mecanismo de escape, la plantilla resultante provocaría que aparezca un cuadro de alerta *JavaScript*:

```
Hola <script>alert('hola!')</script>
```

Y aunque esto parece inofensivo, si un usuario puede llegar hasta aquí, ese mismo usuario también debe ser capaz de escribir código *JavaScript* malicioso que realice acciones dentro de la zona segura de un usuario legítimo sin saberlo.

La respuesta al problema es el mecanismo de escape. Con el mecanismo de escape, reproduces la misma plantilla sin causar daño alguno, y, literalmente, imprimes en pantalla la etiqueta `script`:

```
Hola &lt;script&gt;alert(&#39;holae&#39;)&lt;/script&gt;
```

Twig y los sistemas de plantillas *PHP* abordan el problema de diferentes maneras. Si estás utilizando *Twig*, el mecanismo de escape por omisión está activado y tu aplicación está protegida. En *PHP*, el mecanismo de escape no es automático, lo cual significa que, de ser necesario, necesitas escapar todo manualmente.

Mecanismo de escape en *Twig*

Si estás utilizando las plantillas de *Twig*, entonces el mecanismo de escape está activado por omisión. Esto significa que estás protegido fuera de la caja de las consecuencias no intencionales del código presentado por los usuarios. De forma predeterminada, el mecanismo de escape asume que el contenido se escapó para salida *HTML*.

En algunos casos, tendrás que desactivar el mecanismo de escape cuando estás reproduciendo una variable de confianza y marcado que no se debe escapar. Supongamos que los usuarios administrativos están autorizados para escribir artículos que contengan código *HTML*. De forma predeterminada, *Twig* debe escapar el cuerpo del artículo. Para reproducirlo normalmente, agrega el filtro `raw`: `{{ articulo.cuerpo | raw }}`.

También puedes desactivar el mecanismo de escape dentro de una área `{% block %}` o para una plantilla completa. Para más información, consulta la documentación del Mecanismo de escape *Twig*.

Mecanismo de escape en *PHP*

El mecanismo de escape no es automático cuando utilizas plantillas *PHP*. Esto significa que a menos que escapes una variable expresamente, no estás protegido. Para utilizar el mecanismo de escape, usa el método especial de la vista `escape()`:

```
Hola <?php echo $view->escape($nombre) ?>
```

De forma predeterminada, el método `escape()` asume que la variable se está reproduciendo en un contexto *HTML* (y por tanto la variable se escapa para que sea *HTML* seguro). El segundo argumento te permite cambiar el contexto. Por ejemplo, para mostrar algo en una cadena *JavaScript*, utiliza el contexto `js`:

```
var miMsj = 'Hola <?php echo $view->escape($nombre, 'js') ?>';
```

2.18.10 Formato de plantillas

Las plantillas son una manera genérica para reproducir contenido en *cualquier* formato. Y aunque en la mayoría de los casos debes utilizar plantillas para reproducir contenido *HTML*, una plantilla fácilmente puede generar *JavaScript*, *CSS*, *XML* o cualquier otro formato que puedas soñar.

Por ejemplo, el mismo "recurso" a menudo se reproduce en varios formatos diferentes. Para reproducir una página índice de artículos en formato XML, basta con incluir el formato en el nombre de la plantilla:

- *nombre de plantilla XML*: `AcmeArticuloBundle:Articulo:index.xml.twig`
- *nombre de archivo XML*: `index.xml.twig`

En realidad, esto no es más que una convención de nomenclatura y la plantilla realmente no se reproduce de manera diferente en función de ese formato.

En muchos casos, posiblemente desees permitir que un solo controlador reproduzca múltiples formatos basándose en el "formato de la petición". Por esa razón, un patrón común es hacer lo siguiente:

```
public function indexAction()
{
    $formato = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$formato.'.twig');
}
```

El `getRequestFormat` en el objeto `Petición` por omisión es *HTML*, pero lo puedes devolver en cualquier otro formato basándote en el formato solicitado por el usuario. El formato de la petición muy frecuentemente es gestionado por el ruteador, donde puedes configurar una ruta para que `/contacto` establezca el formato `html` de la petición, mientras que `/contacto.xml` establezca al formato `xml`. Para más información, consulta el *ejemplo avanzado en el capítulo de Enrutado* (Página 92).

Para crear enlaces que incluyan el parámetro de formato, agrega una clave `_format` en el parámetro `hash`:

- *Twig*

  ```
  <a href="{{ path('articulo_show', {'id': 123, '_format': 'pdf'}) }}">
      Versión PDF
  </a>
  ```

- *PHP*

  ```
  <a href="<?php echo $view['router']->generate('articulo_show', array('id' => 123, '_format' => '
      Versión PDF
  </a>
  ```

2.18.11 Consideraciones finales

El motor de plantillas de *Symfony* es una poderosa herramienta que puedes utilizar cada vez que necesites generar contenido de presentación en *HTML*, *XML* o cualquier otro formato. Y aunque las plantillas son una manera común de generar contenido en un controlador, su uso no es obligatorio. El objeto `Petición` devuelto por un controlador se puede crear usando o sin usar una plantilla:

```
// crea un objeto Respuesta donde el contenido reproduce la plantilla
$respuesta = $this->render('AcmeArticuloBundle:Articulo:index.html.twig');

// crea un objeto Response cuyo contenido es texto simple
$respuesta = new Response('contenido respuesta');
```

El motor de plantillas de *Symfony* es muy flexible y de manera predeterminada hay dos diferentes reproductores de plantilla disponibles: las tradicionales plantillas *PHP* y las elegantes y potentes plantillas *Twig*. Ambas apoyan una jerarquía de plantillas y vienen empacadas con un rico conjunto de funciones ayudantes capaces de realizar las tareas más comunes.

En general, el tema de las plantillas se debe pensar como una poderosa herramienta que está a tu disposición. En algunos casos, posiblemente no necesites reproducir una plantilla, y en *Symfony2*, eso está absolutamente bien.

2.18.12 Aprende más en el recetario

- *Cómo usar plantillas PHP en lugar de Twig* (Página 403)
- *Cómo personalizar páginas de error* (Página 277)

2.19 Bases de datos y *Doctrine* ("El modelo")

Seamos realistas, una de las tareas más comunes y desafiantes para cualquier aplicación consiste en la persistencia y la lectura de la información hacia y desde una base de datos. Afortunadamente, *Symfony* viene integrado con Doctrine, una biblioteca, cuyo único objetivo es dotarte de poderosas herramientas para facilitarte esto. En este capítulo, aprenderás la filosofía básica detrás de *Doctrine* y verás lo fácil que puede ser trabajar con una base de datos.

Nota: *Doctrine* está totalmente desconectado de *Symfony* y utilizarlo es opcional. Este capítulo trata acerca del *ORM Doctrine*, el cual te permite asignar objetos a una base de datos relacional (tal como *MySQL*, *PostgreSQL* o *Microsoft SQL*). Si prefieres utilizar las consultas de base de datos en bruto, es fácil, y se explica en el artículo "*Cómo utiliza Doctrine la capa DBAL* (Página 304)" del recetario.

También puedes persistir tus datos en MongoDB utilizando la biblioteca *ODM* de *Doctrine*. Para más información, lee la documentación en "*DoctrineMongoDBBundle* (Página 601)".

2.19.1 Un ejemplo sencillo: un producto

La forma más fácil de entender cómo funciona *Doctrine* es verla en acción. En esta sección, configuraremos tu base de datos, crearemos un objeto `Producto`, lo persistiremos en la base de datos y lo recuperaremos de nuevo.

Código del ejemplo

Si quieres seguir el ejemplo de este capítulo, crea el paquete `AcmeTiendaBundle` ejecutando la orden:

```
php app/console generate:bundle --namespace=Acme/GuardaBundle
```

Configurando la base de datos

Antes de comenzar realmente, tendrás que configurar tu información de conexión a la base de datos. Por convención, esta información se suele configurar en un archivo `app/config/parameters.ini`:

```
;app/config/parameters.ini
[parameters]
    database_driver   = pdo_mysql
    database_host     = localhost
    database_name     = test_project
    database_user     = root
    database_password = password
```

Nota: Definir la configuración a través de `parameters.ini` sólo es una convención. Los parámetros definidos en este archivo son referidos en el archivo de configuración principal al configurar *Doctrine*:

```
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
```

Al separar la información de la base de datos en un archivo independiente, puedes guardar fácilmente diferentes versiones del archivo en cada servidor. También puedes almacenar fácilmente la configuración de la base de datos (o cualquier otra información sensible) fuera de tu proyecto, posiblemente dentro de tu configuración de Apache, por ejemplo. Para más información, consulta *Cómo configurar parámetros externos en el contenedor de servicios* (Página 329).

Ahora que *Doctrine* conoce acerca de tu base de datos, posiblemente tenga que crear la base de datos para ti:

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

Creando una clase Entidad

Supongamos que estás construyendo una aplicación donde necesitas mostrar tus productos. Sin siquiera pensar en *Doctrine* o en una base de datos, ya sabes que necesitas un objeto `Producto` para representar los productos. Crea esta clase en el directorio `Entity` de tu paquete `AcmeGuardaBundle`:

```
// src/Acme/GuardaBundle/Entity/Producto.php
namespace Acme\GuardaBundle\Entity;

class Producto
{
    protected $nombre;

    protected $precio;
```

```
    protected $descripcion;
}
```

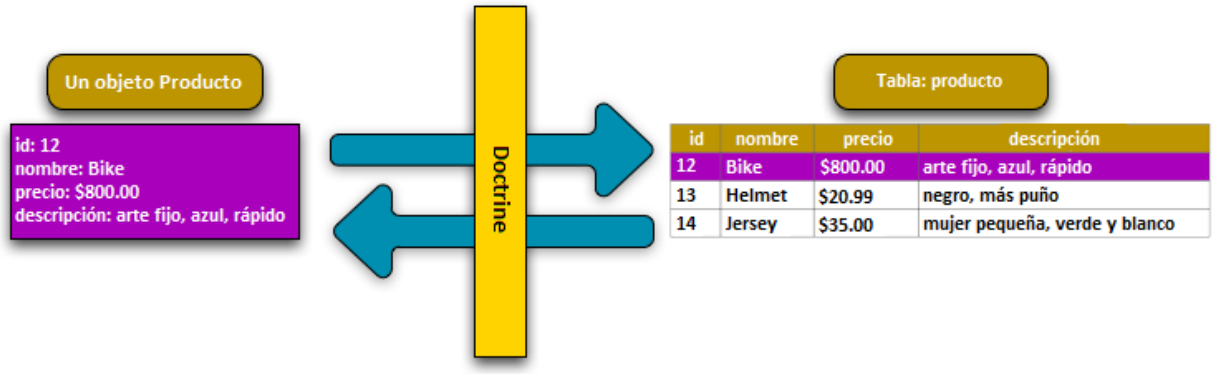
La clase - a menudo llamada “entidad”, es decir, *una clase básica que contiene datos* - es simple y ayuda a cumplir con el requisito del negocio de productos que necesita tu aplicación. Sin embargo, esta clase no se puede guardar en una base de datos - es sólo una clase *PHP* simple.

Truco: Una vez que aprendas los conceptos de *Doctrine*, puedes dejar que *Doctrine* cree por ti la entidad para esta clase:

```
php app/console doctrine:generate:entity --entity="AcmeGuardaBundle:Producto" --fields="nombre:string
```

Agregando información de asignación

Doctrine te permite trabajar con bases de datos de una manera mucho más interesante que solo recuperar filas de una tabla basada en columnas en una matriz. En cambio, *Doctrine* te permite persistir *objetos* completos a la base de datos y recuperar objetos completos desde la base de datos. Esto funciona asignando una clase *PHP* a una tabla de la base de datos, y las propiedades de esa clase *PHP* a las columnas de la tabla:



Para que *Doctrine* sea capaz de hacer esto, sólo hay que crear “metadatos”, o la configuración que le dice a *Doctrine* exactamente cómo debe *asignar* la clase `Producto` y sus propiedades a la base de datos. Estos metadatos se pueden especificar en una variedad de formatos diferentes, incluyendo *YAML*, XML o directamente dentro de la clase `Producto` a través de anotaciones:

Nota: Un paquete sólo puede aceptar un formato para definir metadatos. Por ejemplo, no es posible mezclar metadatos para la clase Entidad definidos en YAML con definidos en anotaciones *PHP*.

- *Annotations*

```
// src/Acme/GuardaBundle/Entity/Producto.php
namespace Acme\GuardaBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="producto")
 */
class Producto
```

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

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

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $precio;

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

- *YAML*

```
# src/Acme/GuardaBundle/Resources/config/doctrine/Producto.orm.yml
Acme\GuardaBundle\Entity\Producto:
    type: entity
    table: producto
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        nombre:
            type: string
            length: 100
        precio:
            type: decimal
            scale: 2
        descripcion:
            type: text
```

- *XML*

```
<!-- src/Acme/GuardaBundle/Resources/config/doctrine/Producto.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">

    <entity name="Acme\GuardaBundle\Entity\Producto" table="producto">
        <id name="id" type="integer" column="id">
            <generator strategy="AUTO" />
        </id>
        <field name="nombre" column="nombre" type="string" length="100" />
        <field name="precio" column="precio" type="decimal" scale="2" />
        <field name="descripcion" column="descripcion" type="text" />
```

```
        </entity>
    </doctrine-mapping>
```

Truco: El nombre de la tabla es opcional y si la omites, será determinada automáticamente basándose en el nombre de la clase entidad.

Doctrine te permite elegir entre una amplia variedad de tipos de campo diferentes, cada uno con sus propias opciones. Para obtener información sobre los tipos de campo disponibles, consulta la sección *Referencia de tipos de campo Doctrine* (Página 135).

Ver También:

También puedes consultar la Documentación de asignación básica de *Doctrine* para todos los detalles sobre la información de asignación. Si utilizas anotaciones, tendrás que prefijar todas las anotaciones con `ORM\` (por ejemplo, `ORM\Column(..)`), lo cual no se muestra en la documentación de *Doctrine*. También tendrás que incluir la declaración `use Doctrine\ORM\Mapping as ORM;`, la cual *importa* las anotaciones prefijas `ORM`.

Prudencia: Ten cuidado de que tu nombre de clase y propiedades no estén asignados a un área protegida por palabras clave de `SQL` (tal como `group` o `user`). Por ejemplo, si el nombre de clase de tu entidad es `group`, entonces, de manera predeterminada, el nombre de la tabla será `group`, lo cual provocará un error `SQL` en algunos motores. Consulta la Documentación de palabras clave SQL reservadas para que sepas cómo escapar correctamente estos nombres.

Nota: Cuando utilizas otra biblioteca o programa (es decir, Doxygen) que utiliza anotaciones, debes colocar la anotación `@IgnoreAnnotation` en la clase para indicar que se deben ignorar las anotaciones *Symfony*.

Por ejemplo, para evitar que la anotación `@fn` lance una excepción, añade lo siguiente:

```
/**
 * @IgnoreAnnotation("fn")
 */
class Producto
```

Generando captadores y definidores

A pesar de que *Doctrine* ahora sabe cómo persistir un objeto `Producto` a la base de datos, la clase en sí realmente no es útil todavía. Puesto que `Producto` es sólo una clase *PHP* regular, es necesario crear métodos captadores y definidores (por ejemplo, `getNombre()`, `setNombre()`) para poder acceder a sus propiedades (ya que las propiedades son `protegidas`). Afortunadamente, *Doctrine* puede hacer esto por ti con la siguiente orden:

```
php app/console doctrine:generate:entities Acme/GuardaBundle/Entity/Producto
```

Esta orden se asegura de que se generen todos los captadores y definidores para la clase `Producto`. Esta es una orden segura - la puedes ejecutar una y otra vez: sólo genera captadores y definidores que no existen (es decir, no sustituye métodos existentes).

Prudencia: La orden `doctrine:generate:entities` guarda una copia de seguridad del `Producto.php` original llamada `Producto.php~`. En algunos casos, la presencia de este archivo puede provocar un error "No se puede redeclarar la clase". La puedes quitar sin problemas.

También puedes generar todas las entidades conocidas (es decir, cualquier clase *PHP* con información de asignación *Doctrine*) de un paquete o un espacio de nombres completo:

```
php app/console doctrine:generate:entities AcmeStoreBundle
php app/console doctrine:generate:entities Acme
```

Nota: A *Doctrine* no le importa si tus propiedades son `protegidas` o `privadas`, o si una propiedad tiene o no una función captadora o definidora. Aquí, los captadores y definidores se generan sólo porque los necesitarás para interactuar con tu objeto *PHP*.

Creando tablas/esquema de la base de datos

Ahora tienes una clase `Producto` utilizable con información de asignación de modo que *Doctrine* sabe exactamente cómo persistirla. Por supuesto, en tu base de datos aún no tienes la tabla `Producto` correspondiente. Afortunadamente, *Doctrine* puede crear automáticamente todas las tablas de la base de datos necesarias para cada entidad conocida en tu aplicación. Para ello, ejecuta:

```
php app/console doctrine:schema:update --force
```

Truco: En realidad, esta orden es increíblemente poderosa. Esta compara cómo se *debe* ver tu base de datos (en base a la información de asignación de tus entidades) con la forma en que *realmente* se ve, y genera las declaraciones `SQL` necesarias para *actualizar* la base de datos a donde debe estar. En otras palabras, si agregas una nueva propiedad asignando metadatos a `Producto` y ejecutas esta tarea de nuevo, vas a generar la declaración `alter table` necesaria para añadir la nueva columna a la tabla `Producto` existente.

Una forma aún mejor para tomar ventaja de esta funcionalidad es a través de las *migraciones* (Página 597), las cuales te permiten generar estas instrucciones `SQL` y almacenarlas en las clases de la migración, mismas que puedes ejecutar sistemáticamente en tu servidor en producción con el fin de seguir la pista y migrar el esquema de la base de datos segura y fiablemente.

Tu base de datos ahora cuenta con una tabla `producto` completamente funcional, con columnas que coinciden con los metadatos que has especificado.

Persistiendo objetos a la base de datos

Ahora que tienes asignada una entidad `producto` y la tabla `Producto` correspondiente, estás listo para persistir los datos a la base de datos. Desde el interior de un controlador, esto es bastante fácil. Agrega el siguiente método al `DefaultController` del paquete:

```
1  // src/Acme/GuardaBundle/Controller/DefaultController.php
2  use Acme\GuardaBundle\Entity\Producto;
3  use Symfony\Component\HttpFoundation\Response;
4  // ...
5
6  public function createAction()
7  {
8      $producto = new Producto();
9      $producto->setNombre('A Foo Bar');
10     $producto->setPrecio('19.99');
11     $producto->setDescription('Lorem ipsum dolor');
12
13     $em = $this->getDoctrine()->getEntityManager();
14     $em->persist($producto);
15     $em->flush();
16
```

```
17      return new Response('Id de producto '.$producto->getId().' creado.');
18  }
```

Nota: Si estás siguiendo este ejemplo, tendrás que crear una ruta que apunte a esta acción para verla trabajar.

Vamos a recorrer este ejemplo:

- **líneas 8-11** En esta sección, creas una instancia y trabajas con el objeto `$producto` como con cualquier otro objeto PHP normal;
- **Línea 13** Esta línea recupera un objeto *gestor de entidad* de *Doctrine*, el cual es responsable de manejar el proceso de persistir y recuperar objetos hacia y desde la base de datos;
- **Línea 14** El método `persist()` dice a *Doctrine* que "maneje" el objeto `$producto`. Esto en realidad no provoca una consulta que se deba introducir en la base de datos (todavía).
- **Línea 15** Cuando se llama al método `flush()`, *Doctrine* examina todos los objetos que está gestionando para ver si necesitan persistirse en la base de datos. En este ejemplo, el objeto `$producto` aún no se ha persistido, por lo tanto el gestor de la entidad ejecuta una consulta `INSERT` y crea una fila en la tabla `Producto`.

Nota: De hecho, ya que *Doctrine* es consciente de todas tus entidades gestionadas, cuando se llama al método `flush()`, calcula el conjunto de cambios y ejecuta la(s) consulta(s) más eficiente(s) posible(s). Por ejemplo, si persistes un total de 100 objetos `Producto` y, posteriormente llamas a `flush()`, *Doctrine* creará una *sola* declaración preparada y la volverá a utilizar para cada inserción. Este patrón se conoce como *Unidad de trabajo*, y se usa porque es rápido y eficiente.

Al crear o actualizar objetos, el flujo de trabajo siempre es el mismo. En la siguiente sección, verás cómo *Doctrine* es lo suficientemente inteligente como para emitir automáticamente una consulta `UPDATE` si ya existe el registro en la base de datos.

Truco: *Doctrine* proporciona una biblioteca que te permite cargar mediante programación los datos de prueba en tu proyecto (es decir, "accesorios"). Para más información, consulta *DoctrineFixturesBundle* (Página 592).

Recuperando objetos desde la base de datos

Recuperar un objeto desde la base de datos es aún más fácil. Por ejemplo, supongamos que has configurado una ruta para mostrar un `Producto` específico en función del valor de su `identificador`:

```
public function showAction($id)
{
    $producto = $this->getDoctrine()
        ->getRepository('AcmeGuardaBundle:Producto')
        ->find($id);

    if (!$producto) {
        throw $this->createNotFoundException('Ningún producto encontrado con id '.$id);
    }

    // Hace algo, como pasar el objeto $producto a una plantilla
}
```

Al consultar por un determinado tipo de objeto, siempre utilizas lo que se conoce como "repositorio". Puedes pensar en un repositorio como una clase *PHP*, cuyo único trabajo consiste en ayudarte a buscar las entidades de una determinada clase. Puedes acceder al objeto repositorio de una clase de entidad a través de:

```
$repositorio = $this->getDoctrine()
    ->getRepository('AcmeGuardaBundle:Producto');
```

Nota: La cadena `AcmeGuardaBundle:Producto` es un método abreviado que puedes utilizar en cualquier lugar de *Doctrine* en lugar del nombre de clase completo de la entidad (es decir, `Acme\GuardaBundle\Entity\Producto`). Mientras que tu entidad viva bajo el espacio de nombres `Entity` de tu paquete, esto debe funcionar.

Una vez que tengas tu repositorio, tienes acceso a todo tipo de útiles métodos:

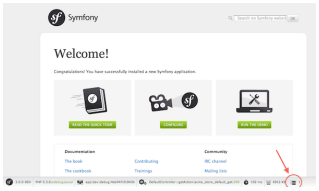
```
// consulta por la clave principal (por lo general "id")
$producto = $repositorio->find($id);

// nombres de método dinámicos para buscar basándose en un valor de columna
$producto = $repositorio->findOneById($id);
$producto = $repositorio->findOneByName('foo');

// recupera *todos* los productos
$productos = $repositorio->findAll();

// busca un grupo de productos basándose en el valor de una columna arbitraria
$productos = $repositorio->findByPrice(19.99);
```

Nota: Por supuesto, también puedes realizar consultas complejas, acerca de las cuales aprenderás más en la sección *Consultando objetos* (Página 123).

También puedes tomar ventaja de los útiles métodos `findBy` y `findOneBy` para recuperar fácilmente los objetos basándote en varias condiciones:

```
// consulta por un producto que coincide en nombre y precio
$producto = $repositorio->findOneBy(array('nombre' => 'foo', 'precio' => 19.99));

// pregunta por todos los productos en que coincide el nombre, ordenados por precio
$producto = $repositorio->findBy(
    array('nombre' => 'foo'),
    array('precio', 'ASC')
);
```

Truco: Cuando reproduces una página, puedes ver, en la esquina inferior derecha de la barra de herramientas de depuración web, cuántas consultas se realizaron.



Si haces clic en el icono, se abrirá el generador de perfiles, mostrando las consultas exactas que se hicieron.

Actualizando un objeto

Una vez que hayas extraído un objeto de *Doctrine*, actualizarlo es relativamente fácil. Supongamos que tienes una ruta que asigna un identificador de producto a una acción de actualización de un controlador:

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();
    $producto = $em->getRepository('AcmeGuardaBundle:Producto')->find($id);

    if (!$producto) {
        throw $this->createNotFoundException('Ningún producto encontrado con id '.$id);
    }

    $producto->setNombre('¡Nuevo nombre de producto!');
    $em->flush();

    return $this->redirect($this->generateUrl('portada'));
}
```

La actualización de un objeto únicamente consiste en tres pasos:

1. Recuperar el objeto desde *Doctrine*;
2. Modificar el objeto;
3. Invocar a `flush()` en el gestor de la entidad

Ten en cuenta que no es necesario llamar a `$em->persist($producto)`. Recuerda que este método simplemente dice a *Doctrine* que maneje o "vea" el objeto `$producto`. En este caso, ya que recuperaste desde *Doctrine* el objeto `$producto`, este ya está gestionado.

Eliminando un objeto

Eliminar un objeto es muy similar, pero requiere una llamada al método `remove()` del gestor de la entidad:

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

Como es de esperar, el método `remove()` notifica a *Doctrine* que deseas eliminar la entidad de la base de datos. La consulta `DELETE` real, sin embargo, no se ejecuta realmente hasta que se invoca al método `flush()`.

2.19.2 Consultando objetos

Ya has visto cómo el objeto repositorio te permite ejecutar consultas básicas sin ningún trabajo:

```
$repositorio->find($id);

$repositorio->findOneByName('Foo');
```

Por supuesto, *Doctrine* también te permite escribir consultas más complejas utilizando el lenguaje de consulta *Doctrine* (`DQL` por *Doctrine Query Language*). `DQL` es similar a `SQL`, excepto que debes imaginar que estás consultando por uno o más objetos de una clase entidad (por ejemplo, `Producto`) en lugar de consultar por filas de una tabla (por ejemplo, `productos`).

Al consultar en *Doctrine*, tienes dos opciones: escribir consultas *Doctrine* puras o utilizar el generador de consultas de *Doctrine*.

Consultando objetos con DQL

Imagina que deseas consultar los productos, pero sólo quieres devolver los productos que cuestan más de 19.99, ordenados del más barato al más caro. Desde el interior de un controlador, haz lo siguiente:

```
$em = $this->getDoctrine()->getEntityManager();
$consulta = $em->createQuery(
    'SELECT p FROM AcmeGuardaBundle:Producto p WHERE p.precio > :precio ORDER BY p.precio ASC'
)->setParameter('precio', '19.99');

$productos = $consulta->getResult();
```

Si te sientes cómodo con SQL, entonces debes sentir a DQL muy natural. La mayor diferencia es que necesitas pensar en términos de "objetos" en lugar de filas de una base de datos. Por esta razón, seleccionas *from* AcmeGuardaBundle:Producto y luego lo apodas p.

El método getResult() devuelve una matriz de resultados. Si estás preguntando por un solo objeto, en su lugar puedes utilizar el método getSingleResult():

```
$producto = $consulta->getSingleResult();
```

Prudencia: El método getSingleResult() lanza una excepción Doctrine\ORM\NoResultException si no se devuelven resultados y una Doctrine\ORM\NonUniqueResultException si se devuelve *más* de un resultado. Si utilizas este método, posiblemente tengas que envolverlo en un bloque try-catch y asegurarte de que sólo se devuelve uno de los resultados (si estás consultando sobre algo que sea viable podrías regresar más de un resultado):

```
$consulta = $em->createQuery('SELECT ....')
    ->setMaxResults(1);

try {
    $producto = $consulta->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $producto = null;
}
// ...
```

La sintaxis DQL es increíblemente poderosa, permitiéndote fácilmente unir entidades (el tema de las *relaciones* (Página 127) se describe más adelante), agrupar, etc. Para más información, consulta la documentación oficial de Doctrine Query Language.

Configurando parámetros

Toma nota del método `setParameter()`. Cuando trabajes con *Doctrine*, siempre es buena idea establecer los valores externos como "marcadores de posición", cómo se hizo en la consulta anterior:

```
... WHERE p.precio > :precio ...
```

Entonces, puedes establecer el valor del marcador de posición `precio` llamando al método `setParameter()`:

```
->setParameter('precio', '19.99')
```

Utilizar parámetros en lugar de colocar los valores directamente en la cadena de consulta, se hace para prevenir ataques de inyección `SQL` y *siempre* se debe hacer. Si estás utilizando varios parámetros, puedes establecer simultáneamente sus valores usando el método `setParameters()`:

```
->setParameters(array(
    'precio' => '19.99',
    'nombre'  => 'Foo',
))
```

Usando el generador de consultas de *Doctrine*

En lugar de escribir las consultas directamente, también puedes usar el `QueryBuilder` de *Doctrine* para hacer el mismo trabajo con una agradable interfaz orientada a objetos. Si usas un IDE, también puedes tomar ventaja del autocompletado a medida que escribes los nombres de método. Desde el interior de un controlador:

```
$repositorio = $this->getDoctrine()
    ->getRepository('AcmeGuardaBundle:Producto');

$consulta = $repositorio->createQueryBuilder('p')
    ->where('p.precio > :precio')
    ->setParameter('precio', '19.99')
    ->orderBy('p.precio', 'ASC')
    ->getQuery();

$productos = $consulta->getResult();
```

El objeto `QueryBuilder` contiene todos los métodos necesarios para construir tu consulta. Al invocar al método `getQuery()`, el generador de consultas devuelve un objeto `Query` normal, el cual es el mismo objeto que construiste directamente en la sección anterior.

Para más información sobre el generador de consultas de *Doctrine*, consulta la documentación del Generador de consultas de *Doctrine*.

Repositorio de clases personalizado

En las secciones anteriores, comenzamos a construir y utilizar consultas más complejas desde el interior de un controlador. Con el fin de aislar, probar y volver a usar estas consultas, es buena idea crear una clase repositorio personalizada para tu entidad y agregar métodos con tu lógica de consulta allí.

Para ello, agrega el nombre de la clase del repositorio a la definición de asignación.

- *Annotations*

  ```
  // src/Acme/GuardaBundle/Entity/Producto.php
  namespace Acme\GuardaBundle\Entity;
  ```

```
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Acme\GuardaBundle\Repository\ProductoRepository")
 */
class Producto
{
    //...
}
```

- *YAML*

```
# src/Acme/GuardaBundle/Resources/config/doctrine/Producto.orm.yml
Acme\GuardaBundle\Entity\Producto:
    type: entity
    repositoryClass: Acme\GuardaBundle\Repository\ProductoRepository
    # ...
```

- *XML*

```
<!-- src/Acme/GuardaBundle/Resources/config/doctrine/Producto.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\GuardaBundle\Entity\Producto"
            repository-class="Acme\GuardaBundle\Repository\ProductoRepository">
            <!-- ... -->
    </entity>
</doctrine-mapping>
```

Doctrine puede generar la clase repositorio por ti ejecutando la misma orden usada anteriormente para generar los métodos captadores y definidores omitidos:

```
php app/console doctrine:generate:entities Acme
```

A continuación, agrega un nuevo método - `findAllOrderedByName()` - a la clase repositorio recién generada. Este método debe consultar por todas las entidades `Producto`, ordenadas alfabéticamente.

```
// src/Acme/GuardaBundle/Repository/ProductoRepository.php
namespace Acme\GuardaBundle\Repository;

use Doctrine\ORM\EntityRepository;

class ProductoRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery('SELECT p FROM AcmeGuardaBundle:Producto p ORDER BY p.nombre ASC')
            ->getResult();
    }
}
```

Truco: Puedes acceder al gestor de la entidad a través de `$this->getEntityManager()` desde el interior del repositorio.

Puedes utilizar este nuevo método al igual que los métodos de búsqueda predeterminados del repositorio:

```
$em = $this->getDoctrine()->getEntityManager();
$productos = $em->getRepository('AcmeGuardaBundle:Producto')
            ->findAllOrderedByName();
```

Nota: Al utilizar una clase repositorio personalizada, todavía tienes acceso a los métodos de búsqueda predeterminados como `find()` y `findAll()`.

2.19.3 Entidad relaciones/asociaciones

Supongamos que los productos en tu aplicación pertenecen exactamente a una "categoría". En este caso, necesitarás un objeto `Categoría` y una manera de relacionar un objeto `Producto` a un objeto `Categoría`. Empieza por crear la entidad `Categoría`. Ya sabemos que tarde o temprano tendrás que persistir la clase a través de *Doctrine*, puedes dejar que *Doctrine* cree la clase para ti.

```
php app/console doctrine:generate:entity --entity="AcmeGuardaBundle:Categoria" --fields="nombre:strir
```

Esta tarea genera la entidad `Categoría` para ti, con un campo `id`, un campo `Nombre` y las funciones captadoras y definidoras asociadas.

Relación con la asignación de metadatos

Para relacionar las entidades `Categoría` y `Producto`, empieza por crear una propiedad `productos` en la clase `Categoría`:

```
// src/Acme/GuardaBundle/Entity/Categoria.php
// ...
use Doctrine\Common\Collections\ArrayCollection;

class Categoria
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Producto", mappedBy="categoria")
     */
    protected $productos;

    public function __construct()
    {
        $this->productos = new ArrayCollection();
    }
}
```

En primer lugar, ya que un objeto `Categoría` debe relacionar muchos objetos `Producto`, agregamos un arreglo como propiedad `Productos` para contener los objetos `Producto`. Una vez más, esto no se hace porque lo necesite *Doctrine*, sino porque tiene sentido en la aplicación para que cada `Categoría` mantenga una gran variedad de objetos `Producto`.

Nota: El código de el método `__construct()` es importante porque *Doctrine* requiere que la propiedad `$productos` sea un objeto `ArrayCollection`. Este objeto se ve y actúa casi *exactamente* como una matriz, pero tiene cierta flexibilidad. Si esto te hace sentir incómodo, no te preocupes. Sólo imagina que es una `matriz` y estarás bien.

A continuación, ya que cada clase `Producto` se puede referir exactamente a un objeto `Categoría`, podrías desear agregar una propiedad `$categoria` a la clase `Producto`:

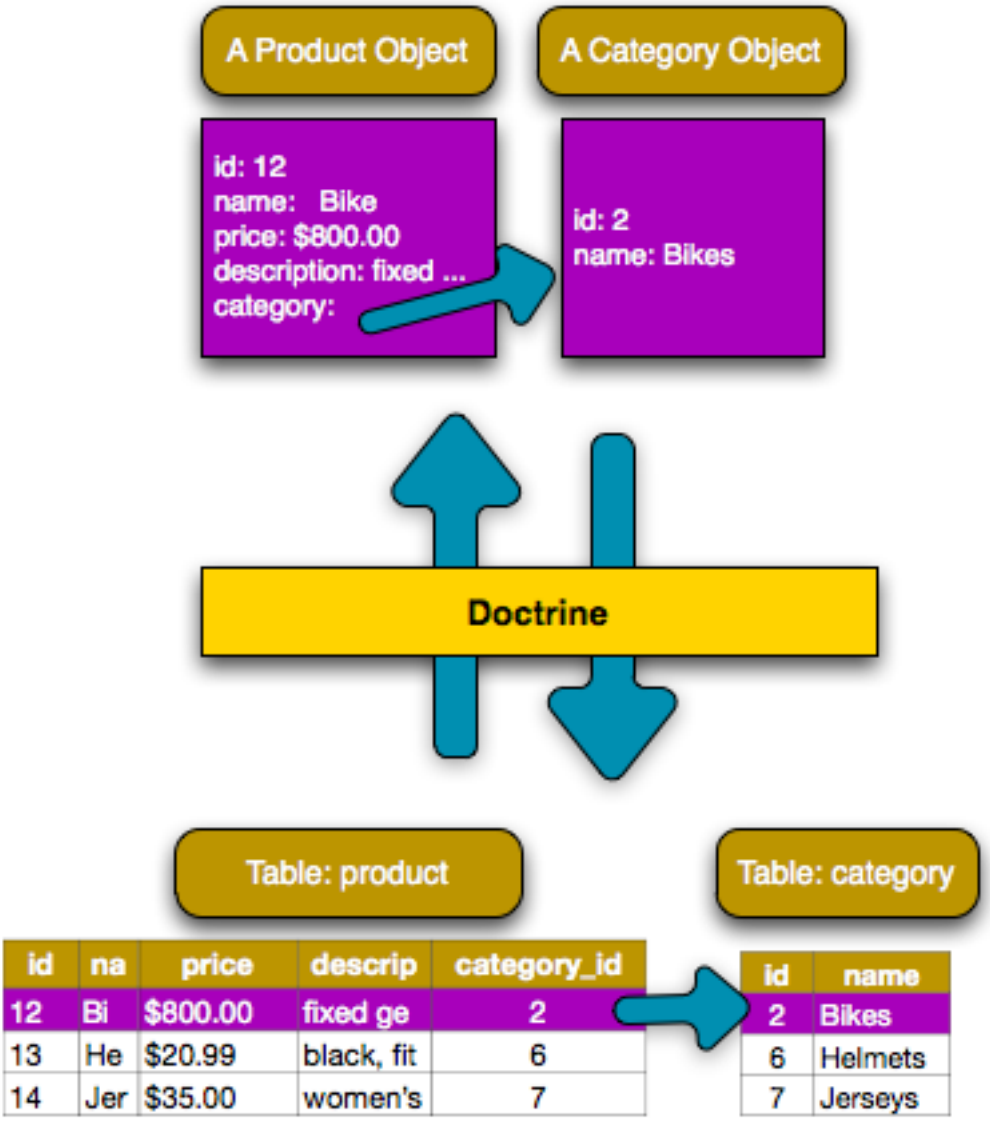
```
// src/Acme/GuardaBundle/Entity/Producto.php
// ...

class Producto
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Categoria", inversedBy="productos")
     * @ORM\JoinColumn(name="categoria_id", referencedColumnName="id")
     */
    protected $categoria;
}
```

Por último, ahora que hemos agregado una nueva propiedad a ambas clases `Categoría` y `Producto`, le informamos a *Doctrine* que genere por ti los métodos captadores y definidores omitidos:

```
php app/console doctrine:generate:entities Acme
```

No hagas caso de los metadatos de *Doctrine* por un momento. Ahora tienes dos clases - `Categoría` y `Producto` con una relación natural de uno a muchos. La clase `Categoría` tiene una matriz de objetos `Producto` y el objeto `producto` puede contener un objeto `Categoría`. En otras palabras - hemos construido tus clases de una manera que tiene sentido para tus necesidades. El hecho de que los datos se tienen que persistir en una base de datos, siempre es secundario.

Ahora, veamos los metadatos sobre la propiedad `$categoria` en la clase `Producto`. Esta información le dice a *Doctrine* que la clase está relacionada con `Categoría` y que debe guardar el `id` del registro de la categoría en un campo `categoria_id` que vive en la tabla `producto`. En otras palabras, el objeto `Categoría` relacionado se almacenará en la propiedad `$categoria`, pero tras bambalinas, *Doctrine* deberá persistir esta relación almacenando el valor del `id` de la categoría en una columna `categoria_id` de la tabla `producto`.



Los metadatos sobre la propiedad `$productos` del objeto `Categoría` son menos importantes, y simplemente dicen a *Doctrine* que vea la propiedad `Producto.categoria` para averiguar cómo se asigna la relación.

Antes de continuar, asegúrate de decirle a *Doctrine* que agregue la nueva tabla `Categoría`, la columna `producto.categoria_id` y la nueva clave externa:

```
php app/console doctrine:schema:update --force
```

Nota: Esta tarea sólo la deberías utilizar durante el desarrollo. Para un método más robusto de actualizar tu base de datos de producción sistemáticamente, lee sobre las :doc:*migraciones de Doctrine </bundles/DoctrineFixturesBundle/index>*.

Guardando entidades relacionadas

Ahora, vamos a ver el código en acción. Imagina que estás dentro de un controlador:

```
// ...
use Acme\GuardaBundle\Entity\Categoria;
use Acme\GuardaBundle\Entity\Producto;
use Symfony\Component\HttpFoundation\Response;
// ...

class DefaultController extends Controller
{
    public function creaProductoAction()
    {
        $categoria = new Categoria();
        $categoria->setNombre('Productos principales');

        $producto = new Producto();
        $producto->setNombre('Foo');
        $producto->setPrecio(19.99);
        // relaciona este producto con la categoría
        $producto->setCategoria($categoria);

        $em = $this->getDoctrine()->getEntityManager();
        $em->persist($categoria);
        $em->persist($producto);
        $em->flush();

        return new Response(
            'Producto con id: '.$producto->getId().' e id de categoría: '.$categoria->getId().' creac
        );
    }
}
```

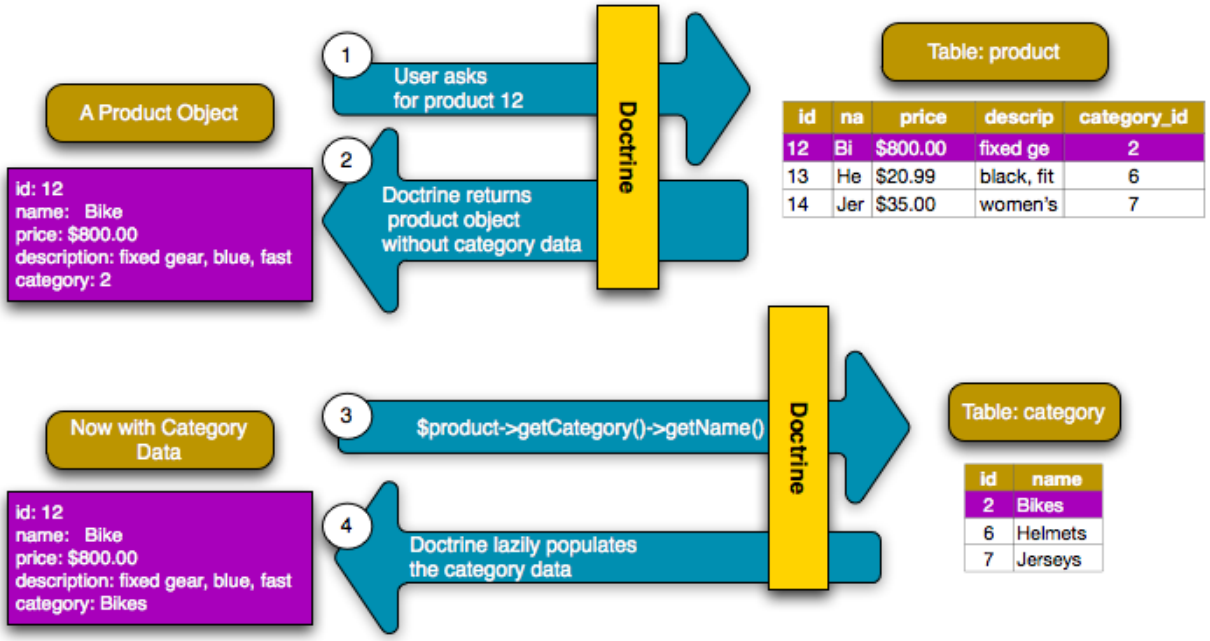
Ahora, se agrega una sola fila a las tablas `categoría` y `producto`. La columna `producto.categoria_id` para el nuevo producto se ajusta a algún `id` de la nueva categoría. *Doctrine* gestiona la persistencia de esta relación para ti.

Recuperando objetos relacionados

Cuando necesites recuperar objetos asociados, tu flujo de trabajo se ve justo como lo hacías antes. En primer lugar, buscas un objeto `$producto` y luego accedes a su `Categoría` asociada:

```
public function showAction($id)
{
    $producto = $this->getDoctrine()
        ->getRepository('AcmeGuardaBundle:Producto')
        ->find($id);

    $nombreCategoria = $producto->getCategoria()->getNombre();

    // ...
}
```

En este ejemplo, primero consultas por un objeto `Producto` basándote en el `id` del producto. Este emite una consulta *solo* para los datos del producto e hidrata al objeto `$producto` con esos datos. Más tarde, cuando llames a `$producto->getCategoria()->getNombre()`, *Doctrine* silenciosamente hace una segunda consulta para encontrar la `Categoría` que está relacionada con este `Producto`. Entonces, prepara el objeto `$categoria` y te lo devuelve.



Lo importante es el hecho de que tienes fácil acceso a la categoría relacionada con el producto, pero, los datos de la categoría realmente no se recuperan hasta que pides la categoría (es decir, trata de "cargarlos de manera diferida").

También puedes consultar en la dirección contraria:

```
public function showProductoAction($id)
{
    $categoria = $this->getDoctrine()
        ->getRepository('AcmeGuardaBundle:Categoria')
        ->find($id);

    $productos = $categoria->getProductos();

    // ...
}
```

En este caso, ocurre lo mismo: primero consultas por un único objeto `Categoría`, y luego *Doctrine* hace una segunda consulta para recuperar los objetos `Producto` relacionados, pero sólo una vez/si le preguntas por ellos (es decir, cuando invoques a `->getProductos()`). La variable `$productos` es una matriz de todos los objetos `Producto` relacionados con el objeto `Categoría` propuesto a través de sus valores `categoria_id`.

Relaciones y clases sustitutas

Esta "carga diferida" es posible porque, cuando sea necesario, *Doctrine* devuelve un objeto "sustituto" en lugar del verdadero objeto. Veamos de nuevo el ejemplo anterior:

```
$producto = $this->getDoctrine()
    ->getRepository('AcmeGuardaBundle:Producto')
    ->find($id);

$categoria = $producto->getCategoria();

// prints "Proxies\AcmeGuardaBundleEntityCategoriaProxy"
echo get_class($categoria);
```

Este objeto sustituto extiende al verdadero objeto `Categoría`, y se ve y actúa exactamente igual que él. La diferencia es que, al usar un objeto sustituto, *Doctrine* puede retrasar la consulta de los datos reales de la `Categoría` hasta que realmente se necesitan esos datos (por ejemplo, hasta que se invoque a `$categoria->getNombre()`).
Las clases sustitutas las genera *Doctrine* y se almacenan en el directorio cache. Y aunque probablemente nunca te des cuenta de que tu objeto `$categoria` en realidad es un objeto sustituto, es importante tenerlo en cuenta. En la siguiente sección, al recuperar simultáneamente los datos del producto y la categoría (a través de una *unión*), *Doctrine* devolverá el *verdadero* objeto `Categoría`, puesto que nada se tiene que cargar de forma diferida.

Uniendo registros relacionados

En los ejemplos anteriores, se realizaron dos consultas - una para el objeto original (por ejemplo, una `Categoría`) y otra para el/los objetos relacionados (por ejemplo, los objetos `Producto`).

Truco: Recuerda que puedes ver todas las consultas realizadas durante una petición a través de la barra de herramientas de depuración web.

Por supuesto, si sabes por adelantado que necesitas tener acceso a los objetos, puedes evitar la segunda consulta emitiendo una unión en la consulta original. Agrega el siguiente método a la clase `ProductoRepository`:

```
// src/Acme/GuardaBundle/Repository/ProductoRepository.php

public function findOneByIdJoinedToCategory($id)
{
    $consulta = $this->getEntityManager()
        ->createQuery('
            SELECT p, c FROM AcmeGuardaBundle:Producto p
            JOIN p.categoria c
            WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $consulta->getSingleResult();
    } catch (\Doctrine\ORM\NoResultException $e) {
        return null;
    }
}
```

Ahora, puedes utilizar este método en el controlador para consultar un objeto `Producto` y su correspondiente `Categoría` con una sola consulta:

```
public function showAction($id)
{
    $producto = $this->getDoctrine()
        ->getRepository('AcmeGuardaBundle:Producto')
        ->findOneByIdJoinedToCategory($id);

    $categoria = $producto->getCategoria();

    // ...
}
```

Más información sobre asociaciones

Esta sección ha sido una introducción a un tipo común de relación entre entidades, la relación uno a muchos. Para obtener detalles más avanzados y ejemplos de cómo utilizar otros tipos de relaciones (por ejemplo, `uno a uno`, `muchos a muchos`), consulta la sección Asignando asociaciones de la documentación de *Doctrine*.

Nota: Si estás utilizando anotaciones, tendrás que prefijar todas las anotaciones con `ORM\` (por ejemplo, `ORM\UnoAMuchos`), lo cual no se refleja en la documentación de *Doctrine*. También tendrás que incluir la declaración `use Doctrine\ORM\Mapping as ORM;`, la cual *importa* las anotaciones prefijas `ORM`.

2.19.4 Configurando

Doctrine es altamente configurable, aunque probablemente nunca tendrás que preocuparte de la mayor parte de sus opciones. Para más información sobre la configuración de *Doctrine*, consulta la sección *Doctrine* del *Manual de referencia* (Página 447).

2.19.5 Ciclo de vida de las retrollamadas

A veces, es necesario realizar una acción justo antes o después de insertar, actualizar o eliminar una entidad. Este tipo de acciones se conoce como "ciclo de vida" de las retrollamadas, ya que son métodos retrollamados que necesitas ejecutar durante las diferentes etapas del ciclo de vida de una entidad (por ejemplo, cuando la entidad es insertada, actualizada, eliminada, etc.)

Si estás utilizando anotaciones para los metadatos, empieza por permitir el ciclo de vida de las retrollamadas. Esto no es necesario si estás usando *YAML* o XML para tu asignación:

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Producto
{
    // ...
}
```

Ahora, puedes decir a *Doctrine* que ejecute un método en cualquiera de los eventos del ciclo de vida disponibles. Por ejemplo, supongamos que deseas establecer una columna de fecha `creado` a la fecha actual, sólo cuando se persiste por primera vez la entidad (es decir, se inserta):

- *Annotations*

```
/**
 * @ORM\prePersist
 */
public function setValorCreado()
{
    $this->creado = new \DateTime();
}
```

- *YAML*

```
# src/Acme/GuardaBundle/Resources/config/doctrine/Producto.orm.yml
Acme\GuardaBundle\Entity\Producto:
    type: entity
    # ...
    lifecycleCallbacks:
        prePersist: [ setValorCreado ]
```

- *XML*

```
<!-- src/Acme/GuardaBundle/Resources/config/doctrine/Producto.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\GuardaBundle\Entity\Producto">
            <!-- ... -->
            <lifecycle-callbacks>
                <lifecycle-callback type="prePersist" method="setValorCreado" />
            </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

Nota: En el ejemplo anterior se supone que has creado y asignado una propiedad `creado` (no mostrada aquí).

Ahora, justo antes de persistir la primer entidad, *Doctrine* automáticamente llamará a este método y establecerá el campo `creado` a la fecha actual.

Esto se puede repetir en cualquiera de los otros eventos del ciclo de vida, los cuales incluyen a:

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`
- `postUpdate`
- `postLoad`
- `loadClassMetadata`

Para más información sobre que significan estos eventos y ciclo de vida de las retrollamadas en general, consulta la sección Eventos del ciclo de vida en la documentación de *Doctrine*.

Ciclo de vida de retrollamada y escuchas de eventos

Observa que el método `setValorCreado()` no recibe argumentos. Este siempre es el caso del ciclo de vida de las retrollamadas y es intencional: el ciclo de vida de las retrollamadas debe ser un método sencillo que se ocupe de transformar los datos internos de la entidad (por ejemplo, estableciendo un campo a creado/actualizado, generar un valor ficticio).
Si necesitas hacer alguna tarea más pesada - como llevar el registro de eventos o enviar un correo electrónico - debes registrar una clase externa como un escucha o suscriptor de eventos y darle acceso a todos los recursos que necesites. Para más información, consulta *Registrando escuchas y suscriptores de eventos* (Página 300).

2.19.6 Extensiones *Doctrine*: `Timestampable`, `Sluggable`, etc.

Doctrine es bastante flexible, y dispone de una serie de extensiones de terceros que te permiten realizar fácilmente tareas repetitivas y comunes en tus entidades. Estas incluyen cosas tales como `Sluggable`, `Timestampable`, `registrable`, `traducible` y `Tree`.

Para más información sobre cómo encontrar y utilizar estas extensiones, ve el artículo sobre el uso de *extensiones comunes *Doctrine** (Página 300).

2.19.7 Referencia de tipos de campo *Doctrine*

Doctrine dispone de una gran cantidad de tipos de campo. Cada uno de estos asigna un tipo de dato PHP a un tipo de columna específica en cualquier base de datos que estés utilizando. Los siguientes tipos son compatibles con *Doctrine*:

- **Cadenas**
  - `string` (usado para cadenas cortas)
  - `text` (usado para cadenas grandes)
- **Números**
  - `integer`
  - `smallint`
  - `bigint`
  - `decimal`
  - `float`
- **Fechas y horas** (usa un objeto DateTime para estos campos en *PHP*)
  - `date`
  - `time`
  - `datetime`
- **Otros tipos**
  - `booleanos`
  - `object` (serializado y almacenado en un campo `CLOB`)
  - `array` (serializado y almacenado en un campo `CLOB`)

Para más información, consulta la sección Asignando tipos en la documentación de *Doctrine*.

Opciones de campo

Cada campo puede tener un conjunto de opciones aplicables. Las opciones disponibles incluyen `type` (el predeterminado es `string`), `name`, `lenght`, `unique` y `nullable`. Aquí tienes unos cuantos ejemplos de anotaciones:

```
/**
 * Un campo cadena con longitud de 255 que no puede ser nulo
 * (reflejando los valores predeterminados para las opciones "tipo", "longitud" y "nulo")
 *
 * @ORM\Column()
 */
protected $nombre;

/**
 * Un campo cadena de longitud 150 que persiste a una columna
 * "direccion_correo" y tiene un índice único.
 *
 * @ORM\Column(name="direccion_correo", unique="true", length="150")
 */
protected $correo;
```

Nota: Hay algunas opciones más que no figuran en esta lista. Para más detalles, consulta la sección Asignando propiedades de la documentación de *Doctrine*.

2.19.8 Ordenes de consola

La integración del *ORM* de *Doctrine2* ofrece varias ordenes de consola bajo el espacio de nombres `doctrine`. Para ver la lista de ordenes puedes ejecutar la consola sin ningún tipo de argumento:

```
php app/console
```

Mostrará una lista de ordenes disponibles, muchas de las cuales comienzan con el prefijo `doctrine:`. Puedes encontrar más información sobre cualquiera de estas ordenes (o cualquier orden de *Symfony*) ejecutando la orden `help`. Por ejemplo, para obtener detalles acerca de la tarea `doctrine:database:create`, ejecuta:

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

Algunas tareas notables o interesantes son:

- `doctrine:ensure-production-settings` - comprueba si el entorno actual está configurado de manera eficiente para producción. Esta siempre se debe ejecutar en el entorno `prod`:

  ```
  php app/console doctrine:ensure-production-settings --env=prod
  ```

- `doctrine:mapping:import` - permite a *Doctrine* llevar a cabo una introspección a una base de datos existente y crear información de asignación. Para más información, consulta *Cómo generar entidades de una base de datos existente* (Página 302).
- `doctrine:mapping:info` - te dice todas las entidades de las que *Doctrine* es consciente y si hay algún error básico con la asignación.
- `doctrine:query:dql` y `doctrine:query:sql` - te permiten ejecutar `DQL` o consultas `SQL` directamente desde la línea de ordenes.

Nota: Para poder cargar accesorios a tu base de datos, en su lugar, necesitas tener instalado el paquete `DoctrineFixturesBundle`. Para aprender cómo hacerlo, lee el artículo en la documentación de "*DoctrineFixturesBundle* (Página 592)".

2.19.9 Resumen

Con *Doctrine*, puedes centrarte en tus objetos y la forma en que son útiles en tu aplicación y luego preocuparte por su persistencia en la base de datos. Esto se debe a que *Doctrine* te permite utilizar cualquier objeto PHP para almacenar los datos y se basa en la información de asignación de metadatos para asignar los datos de un objeto a una tabla particular de la base de datos.

Y aunque *Doctrine* gira en torno a un concepto simple, es increíblemente poderosa, lo cual te permite crear consultas complejas y suscribirte a los eventos que te permiten realizar diferentes acciones conforme los objetos recorren su ciclo de vida en la persistencia.

Para más información acerca de *Doctrine*, ve la sección *Doctrine* del *recetario* (Página 275), que incluye los siguientes artículos:

- *DoctrineFixturesBundle* (Página 592)
- *Extensiones Doctrine: Timestampable: Sluggable, Translatable, etc.* (Página 300)

2.20 Probando

Cada vez que escribes una nueva línea de código, potencialmente, también, añades nuevos errores. Las pruebas automatizadas deben tener todo cubierto y esta guía muestra cómo escribir pruebas unitarias y funcionales para tu aplicación *Symfony2*.

2.20.1 Plataforma de pruebas

En *Symfony2* las pruebas dependen en gran medida de *PHPUnit*, tus buenas prácticas, y algunos convenios. Esta parte no documenta *PHPUnit* en sí mismo, pero si aún no lo conoces, puedes leer su excelente documentación.

Nota: *Symfony2* trabaja con *PHPUnit* 3.5.11 o posterior.

La configuración predeterminada de *PHPUnit* busca pruebas bajo el subdirectorio `Tests/` de tus paquetes:

```
<!-- app/phpunit.xml.dist -->

<phpunit bootstrap="../src/autoload.php">
    <testsuites>
        <testsuite name="Proyecto banco de pruebas">
            <directory>../src/*/*Bundle/Tests</directory>
        </testsuite>
    </testsuites>

    ...
</phpunit>
```

La ejecución del banco de pruebas para una determinada aplicación es muy sencilla:

```
# especifica la configuración del directorio en la línea de ordenes
$ phpunit -c app/

# o ejecuta phpunit desde el directorio de la aplicación
$ cd app/
$ phpunit
```

Truco: La cobertura de código se puede generar con la opción `--coverage-html`.

2.20.2 Pruebas unitarias

Escribir pruebas unitarias en *Symfony2* no es diferente a escribir pruebas unitarias *PHPUnit* normales. Por convención, recomendamos replicar la estructura de directorios de tu paquete bajo el subdirectorio `Tests/`. Por lo tanto, para escribir las pruebas para la clase `'Acme\HolaBundle\Model\Articulo` en el archivo `Acme/HolaBundle/Tests/Model/ArticuloTest.php`.

En una prueba unitaria, el autocargado se activa automáticamente a través del archivo `src/autoload.php` (tal como está configurado por omisión en el archivo `phpunit.xml.dist`).

Correr las pruebas de un determinado archivo o directorio también es muy fácil:

```
# corre todas las pruebas del controlador
$ phpunit -c app src/Acme/HolaBundle/Tests/Controller/

# ejecuta todas las pruebas del modelo
$ phpunit -c app src/Acme/HolaBundle/Tests/Model/

# ejecuta las pruebas para la clase Articulo
$ phpunit -c app src/Acme/HolaBundle/Tests/Model/ArticuloTest.php

# ejecuta todas las pruebas del paquete entero
$ phpunit -c app src/Acme/HolaBundle/
```

2.20.3 Pruebas funcionales

Las pruebas funcionales verifican la integración de las diferentes capas de una aplicación (desde el enrutado hasta la vista). Ellas no son diferentes de las pruebas unitarias en cuanto a *PHPUnit* se refiere, pero tienen un flujo de trabajo muy específico:

- Hacer una petición;
- Probar la respuesta;
- Hacer clic en un enlace o enviar un formulario;
- Probar la respuesta;
- Enjuagar y repetir.

Las peticiones, clics y los envíos los hace un cliente que sabe cómo hablar con la aplicación. Para acceder a este cliente, tus pruebas necesitan ampliar la clase `WebTestCase` de *Symfony2*. La Edición estándar de *Symfony2* proporciona una sencilla prueba funcional para `DemoController` que dice lo siguiente:

```
// src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php
namespace Acme\DemoBundle\Tests\Controller;
```

```
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DemoControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $cliente = static::createClient();

        $impulsor = $cliente->request('GET', '/demo/hola/Fabien');

        $this->assertTrue($impulsor->filter('html:contains("Hola Fabien")')->count() > 0);
    }
}
```

El método `createClient()` devuelve un cliente vinculado a la aplicación actual:

```
$impulsor = $cliente->request('GET', '/demo/hola/Fabien');
```

El método `request()` devuelve un objeto `Impulsor` que puedes utilizar para seleccionar elementos en la respuesta, hacer clic en enlaces, y enviar formularios.

Truco: El `Impulsor` sólo se puede utilizar si el contenido de la respuesta es un documento *XML* o *HTML*. Para otro tipo de contenido, consigue el contenido de la respuesta con `$cliente->getResponse()->getContent()`.

Puedes configurar el `content-type` de la petición a `JSON` añadiendo `HTTP_CONTENT_TYPE => application/json`.

Truco: La firma completa del método `request()` es la siguiente:

```
request($metodo,
    $uri,
    array $parametros = array(),
    array $archivos = array(),
    array $servidor = array(),
    $contenido = null,
    $cambiaHistoria = true
)
```

Haz clic en un enlace seleccionándolo primero con el `Impulsor` utilizando una expresión XPath o un selector *CSS*, luego utiliza el cliente para hacer clic en él:

```
$enlace = $impulsor->filter('a:contains("Bienvenido")')->eq(1)->link();

$impulsor = $cliente->click($enlace);
```

El envío de un formulario es muy similar; selecciona un botón del formulario, opcionalmente sustituye algunos valores del formulario, y envía el formulario correspondiente:

```
$formulario = $impulsor->selectButton('submit')->form();

// sustituye algunos valores
$formulario['nombre'] = 'Lucas';

// envía el formulario
$impulsor = $cliente->submit($formulario);
```

Cada campo del `formulario` tiene métodos especializados en función de su tipo:

```
// llena un campo de texto (input)
$formulario['nombre'] = 'Lucas';

// selecciona una opción o un botón de radio
$formulario['country']->select('Francia');

// marca una casilla de verificación (checkbox)
$formulario['like_symfony']->tick();

// carga un archivo
$formulario['photo']->upload('/ruta/a/lucas.jpg');
```

En lugar de cambiar un campo a la vez, también puedes pasar una matriz de valores al método `submit()`:

```
$impulsor = $cliente->submit($formulario, array(
    'nombre'       => 'Lucas',
    'country'      => 'Francia',
    'like_symfony' => true,
    'photo'        => '/ruta/a/lucas.jpg',
));
```

Ahora que puedes navegar fácilmente a través de una aplicación, utiliza las aserciones para probar que en realidad hace lo que se espera. Utiliza el `Impulsor` para hacer aserciones sobre el DOM:

```
// Afirma que la respuesta concuerda con un determinado selector CSS.
$this->assertTrue($impulsor->filter('h1')->count() > 0);
```

O bien, prueba contra el contenido de la respuesta directamente si lo que deseas es acertar que el contenido contiene algún texto, o si la respuesta no es un documento *XML/HTML*:

```
$this->assertRegExp('/Hola Fabien/', $cliente->getResponse()->getContent());
```

Útiles aserciones

Después de algún tiempo, te darás cuenta de que siempre escribes el mismo tipo de aserciones. Para empezar más rápido, aquí está una lista de las aserciones más comunes y útiles:

```
// Afirma que la respuesta concuerda con un determinado selector CSS.
$this->assertTrue($impulsor->filter($selector)->count() > 0);

// Afirma que la respuesta concuerda n veces con un determinado selector CSS.
$this->assertEquals($count, $impulsor->filter($selector)->count());

// Afirma que la cabecera de la respuesta tiene un valor dado.
$this->assertTrue($cliente->getResponse()->headers->contains($key, $valor));

// Afirma que el contenido de la respuesta concuerda con una expresión regular.
$this->assertRegExp($regexp, $cliente->getResponse()->getContent());

// Acierta el código de estado de la respuesta.
$this->assertTrue($cliente->getResponse()->isSuccessful());
$this->assertTrue($cliente->getResponse()->isNotFound());
$this->assertEquals(200, $cliente->getResponse()->getStatusCode());

// Afirma que el código de estado de la respuesta es una redirección.
$this->assertTrue($cliente->getResponse()->isRedirect('google.com'));
```

2.20.4 El Cliente de pruebas

El Cliente de prueba simula un cliente HTTP tal como un navegador.

Nota: El Cliente de prueba está basado en los componentes `BrowserKit` e `Impulsor`.

Haciendo peticiones

El cliente sabe cómo hacer peticiones a una aplicación *Symfony2*:

```
$impulsor = $cliente->request('GET', '/hola/Fabien');
```

El método `request()` toma el método *HTTP* y una *URL* como argumentos y devuelve una instancia de `Impulsor`.

Utiliza el rastreador Impulsor para encontrar los elementos del DOM en la respuesta. Puedes utilizar estos elementos para hacer clic en los enlaces y presentar formularios:

```
$enlace = $impulsor->selectLink('Ve a algún lugar...')->link();
$impulsor = $cliente->click($enlace);

$formulario = $impulsor->selectButton('validate')->form();
$impulsor = $cliente->submit($formulario, array('nombre' => 'Fabien'));
```

Ambos métodos `click()` y `submit()` devuelven un objeto `Impulsor`. Estos métodos son la mejor manera para navegar por una aplicación que esconde un montón de detalles. Por ejemplo, al enviar un formulario, este detecta automáticamente el método *HTTP* y la *URL* del formulario, te da una buena *API* para subir archivos, y combinar los valores presentados con el formulario predeterminado, y mucho más.

Truco: Aprenderás más sobre los objetos `Link` y `Form` más adelante en la sección Impulsor.

Pero también puedes simular el envío de formularios y peticiones complejas con argumentos adicionales del método `request()`:

```
// envía el formulario
$cliente->request('POST', '/submit', array('nombre' => 'Fabien'));

// envía un formulario con un campo para subir un archivo
use Symfony\Component\HttpFoundation\File\UploadedFile;

$foto = new UploadedFile('/ruta/a/foto.jpg', 'foto.jpg', 'image/jpeg', 123);
// o
$foto = array('nombre_temp' => '/ruta/a/foto.jpg', 'nombre' => 'foto.jpg', 'type' => 'image/jpeg', 's

$cliente->request('POST', '/submit', array('nombre' => 'Fabien'), array('foto' => $foto));

// especifica las cabeceras HTTP
$cliente->request('DELETE', '/post/12', array(), array(), array('PHP_AUTH_USER' => 'username', 'PHP_A
```

Truco: Los envíos de formulario se simplifican en gran medida usando un objeto rastreador (crawler- ve más adelante).

Cuando una petición devuelve una respuesta de redirección, el cliente la sigue automáticamente. Este comportamiento se puede cambiar con el método `followRedirects()`:

```
$cliente->followRedirects(false);
```

Cuando el cliente no sigue los cambios de dirección, puedes forzar el cambio de dirección con el método `followRedirect()`:

```
$impulsor = $cliente->followRedirect();
```

Por último pero no menos importante, puedes hacer que cada petición se ejecute en su propio proceso PHP para evitar efectos secundarios cuando se trabaja con varios clientes en el mismo archivo:

```
$cliente->insulate();
```

Navegando

El cliente es compatible con muchas operaciones que se pueden hacer en un navegador real:

```
$cliente->back();
$cliente->forward();
$cliente->reload();

// Limpia todas las cookies y el historial
$cliente->restart();
```

Accediendo a objetos internos

Si utilizas el cliente para probar tu aplicación, posiblemente quieras acceder a los objetos internos del cliente:

```
$history   = $cliente->getHistory();
$cookieJar = $cliente->getCookieJar();
```

También puedes obtener los objetos relacionados con la última petición:

```
$peticion  = $cliente->getRequest();
$respuesta = $cliente->getResponse();
$impulsor  = $cliente->getCrawler();
```

Si tus peticiones no son aisladas, también puedes acceder al `Contenedor` y al `kernel`:

```
$contenedor = $cliente->getContainer();
$kernel    = $cliente->getKernel();
```

Accediendo al contenedor

Es altamente recomendable que una prueba funcional sólo pruebe la respuesta. Sin embargo, bajo ciertas circunstancias muy raras, posiblemente desees acceder a algunos objetos internos para escribir aserciones. En tales casos, puedes acceder al contenedor de inyección de dependencias:

```
$contenedor = $cliente->getContainer();
```

Ten en cuenta que esto no tiene efecto si aíslas el cliente o si utilizas una capa HTTP.

Truco: Si la información que necesitas comprobar está disponible desde el generador de perfiles, úsalo en su lugar.

Accediendo a los datos del perfil

Para afirmar los datos recogidos por el generador de perfiles, puedes conseguir el perfil de la petición actual de la siguiente manera:

```
$perfil = $cliente->getProfile();
```

Redirigiendo

De manera predeterminada, el cliente no sigue las redirecciones *HTTP*, por lo tanto puedes conseguir y analizar la respuesta antes de redirigir. Una vez que quieras redirigir al cliente, invoca al método `followRedirect()`:

```
// hace algo para emitir la redirección (por ejemplo, llenar un formulario)

// sigue la redirección
$impulsor = $cliente->followRedirect();
```

Si deseas que el cliente siempre sea redirigido automáticamente, puedes llamar al método `followRedirects()`:

```
$cliente->followRedirects();

$impulsor = $cliente->request('GET', '/');

// sigue todas las redirecciones

// configura de nuevo al Cliente para redirigirlo manualmente
$cliente->followRedirects(false);
```

2.20.5 El `Impulsor`

Cada vez que hagas una petición con el cliente devolverá una instancia del `Impulsor`. Este nos permite recorrer documentos *HTML*, seleccionar nodos, encontrar enlaces y formularios.

Creando una instancia de `Impulsor`

Cuando haces una petición con un Cliente, automáticamente se crea una instancia del `Impulsor`. Pero puedes crear el tuyo fácilmente:

```
use Symfony\Component\DomCrawler\Crawler;

$impulsor = new Crawler($html, $url);
```

El constructor toma dos argumentos: el segundo es la *URL* que se utiliza para generar las direcciones absolutas para los enlaces y formularios, el primero puede ser cualquiera de los siguientes:

- Un documento *HTML*;
- Un documento *XML*;
- Una instancia de `DOMDocument`;
- Una instancia de `DOMNodeList`;
- Una instancia de `DOMNode`;
- Un arreglo de todos los elementos anteriores.

Después de crearla, puedes agregar más nodos:

Método	Descripción
`addHTMLDocument()`	Un documento *HTML*
`addXMLDocument()`	Un documento *XML*
`addDOMDocument()`	Una instancia de `DOMDocument`
`addDOMNodeList()`	Una instancia de `DOMNodeList`
`addDOMNode()`	Una instancia de `DOMNode`
`addNodes()`	Un arreglo de los elementos anteriores
`add()`	Acepta cualquiera de los elementos anteriores

Recorriendo

Como jQuery, el `Impulsor` tiene métodos para recorrer el DOM de un documento *HTML/XML*:

Método	Descripción
`filter('h1')`	Nodos que concuerdan con el selector *CSS*
`filterXpath('h1')`	Nodos que concuerdan con la expresión XPath
`eq(1)`	Nodo para el índice especificado
`first()`	Primer nodo
`last()`	Último nodo
`siblings()`	Hermanos
`nextAll()`	Los siguientes hermanos
`previousAll()`	Los hermanos precedentes
`parents()`	Nodos padre
`children()`	Hijos
`reduce($lambda)`	Nodos para los que el ejecutable no devuelve falso

Puedes limitar iterativamente tu selección de nodo encadenando llamadas a métodos, ya que cada método devuelve una nueva instancia del `Impulsor` para los nodos coincidentes:

```
$impulsor
    ->filter('h1')
    ->reduce(function ($nodo, $i)
    {
        if (!$nodo->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Truco: Utiliza la función `count()` para obtener el número de nodos almacenados en un `Impulsor`: `count($impulsor)`

Extrayendo información

El `Impulsor` puede extraer información de los nodos:

```
// Devuelve el valor atributo del primer nodo
$impulsor->attr('class');

// Devuelve el valor del nodo para el primer nodo
$impulsor->text();
```

```
// Extrae una arreglo de atributos de todos los nodos (_text devuelve el valor del nodo)
$impulsor->extract(array('_text', 'href'));

// Ejecuta una lambda por cada nodo y devuelve una matriz de resultados
$datos = $impulsor->each(function ($nodo, $i)
{
    return $nodo->getAttribute('href');
});
```

Enlaces

Puedes seleccionar enlaces con los métodos de recorrido, pero el acceso directo `selectLink()` a menudo es más conveniente:

```
$impulsor->selectLink('Haz clic aquí');
```

Este selecciona los enlaces que contienen el texto dado, o hace clic en imágenes en las que el atributo `alt` contiene el texto dado.

El método `click()` del cliente toma una instancia de `Link` como si la hubiera devuelto el método `link()`:

```
$enlace = $impulsor->link();

$cliente->click($enlace);
```

Truco: El método `links()` devuelve un arreglo de objetos `Link` para todos los nodos.

Formularios

En cuanto a los enlaces, tú seleccionas formularios con el método `selectButton()`:

```
$impulsor->selectButton('submit');
```

Ten en cuenta que seleccionas botones del formulario y no el formulario porque un formulario puede tener varios botones, si utilizas la *API* de recorrido, ten en cuenta que debes buscar un botón.

El método `selectButton()` puede seleccionar etiquetas `button` y enviar etiquetas `input`, este tiene varias heurísticas para encontrarlas:

- El valor `value` del atributo;
- El valor del atributo `id` o `alt` de imágenes;
- El valor del atributo `id` o `name` de las etiquetas `button`.

Cuando tienes un nodo que representa un botón, llama al método `form()` para conseguir una instancia de `Form` que envuelve al nodo botón:

```
$formulario = $impulsor->form();
```

Cuando llamas al método `form()`, también puedes pasar una matriz de valores de campo que sustituyen los valores predeterminados:

```
$formulario = $impulsor->form(array(
    'nombre'       => 'Fabien',
    'like_symfony' => true,
));
```

Y si quieres simular un método *HTTP* específico del formulario, pásalo como segundo argumento:

```
$formulario = $impulsor->form(array(), 'DELETE');
```

El cliente puede enviar instancias de `Form`:

```
$cliente->submit($formulario);
```

Los valores del campo también se pueden pasar como segundo argumento del método `submit()`:

```
$cliente->submit($formulario, array(
    'nombre'       => 'Fabien',
    'like_symfony' => true,
));
```

Para situaciones más complejas, utiliza la instancia de `Form` como una matriz para establecer el valor de cada campo individualmente:

```
// Cambia el valor de un campo
$formulario['nombre'] = 'Fabien';
```

También hay una buena *API* para manipular los valores de los campos de acuerdo a su tipo:

```
// selecciona una opción o un botón de radio
$formulario['country']->select('Francia');

// marca una casilla de verificación (checkbox)
$formulario['like_symfony']->tick();

// carga un archivo
$formulario['photo']->upload('/ruta/a/lucas.jpg');
```

Truco: Puedes conseguir los valores que se enviarán llamando al método `getValues()`. Los archivos subidos están disponibles en un arreglo separado devuelto por `getFiles()`. `getPhpValues()` y `getPhpFiles()` también devuelven los valores presentados, pero en formato PHP (que convierte las claves con notación de corchetes a matrices PHP).

2.20.6 Probando la configuración

Configuración de *PHPUnit*

Cada aplicación tiene su propia configuración de *PHPUnit*, almacenada en el archivo `phpunit.xml.dist`. Puedes editar este archivo para cambiar los valores predeterminados o crear un archivo `phpunit.xml` para modificar la configuración de tu máquina local.

Truco: Guarda el archivo `phpunit.xml.dist` en tu repositorio de código, e ignora el archivo `phpunit.xml`.

De forma predeterminada, sólo las pruebas almacenadas en los paquetes "estándar" las ejecuta la orden `PHPUnit` (las pruebas estándar están bajo el espacio de nombres Vendor*Bundle\Tests). Pero, fácilmente puedes agregar más espacios de nombres. Por ejemplo, la siguiente configuración agrega las pruebas de los paquetes de terceros instalados:

```
<!-- hola/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Proyecto banco de pruebas">
        <directory>../src/*/*Bundle/Tests</directory>
```

```
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

Para incluir otros espacios de nombres en la cobertura de código, también edita la sección `<filter>`:

```
<filter>
    <whitelist>
        <directory>../src</directory>
        <exclude>
            <directory>../src/*/*Bundle/Resources</directory>
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/Acme/Bundle/*Bundle/Resources</directory>
            <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
        </exclude>
    </whitelist>
</filter>
```

Configurando el cliente

El cliente utilizado por las pruebas funcionales crea un núcleo que se ejecuta en un entorno de `prueba` especial, por lo tanto puedes ajustar tanto como desees:

- *YAML*

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

framework:
    error_handler: false
    test: ~

web_profiler:
    toolbar: false
    intercept_redirects: false

monolog:
    handlers:
        main:
            type:  stream
            path:  %kernel.logs_dir%/%kernel.environment%.log
            level: debug
```

- *XML*

```
<!-- app/config/config_test.xml -->
<container>
    <imports>
        <import resource="config_dev.xml" />
    </imports>

    <webprofiler:config
        toolbar="false"
        intercept-redirects="false"
    />

    <framework:config error_handler="false">
```

```
            <framework:test />
        </framework:config>

        <monolog:config>
            <monolog:main
                type="stream"
                path="%kernel.logs_dir%/%kernel.environment%.log"
                level="debug"
             />
        </monolog:config>
    </container>
```

- *PHP*

```
// app/config/config_test.php
$cargador->import('config_dev.php');

$contenedor->loadFromExtension('framework', array(
    'error_handler' => false,
    'test'          => true,
));

$contenedor->loadFromExtension('web_profiler', array(
    'toolbar' => false,
    'intercept-redirects' => false,
));

$contenedor->loadFromExtension('monolog', array(
    'handlers' => array(
        'main' => array('type' => 'stream',
                        'path' => '%kernel.logs_dir%/%kernel.environment%.log'
                        'level' => 'debug')

)));
```

Además, puedes cambiar el entorno predeterminado (`test`) y sustituir el modo de depuración por omisión (`true`) pasándolos como opciones al método `createClient()`:

```
$cliente = static::createClient(array(
    'environment' => 'mi_entorno_de_prueba',
    'debug'       => false,
));
```

Si tu aplicación se comporta de acuerdo a algunas cabeceras *HTTP*, pásalas como segundo argumento de `createClient()`:

```
$cliente = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.ejemplo.com',
    'HTTP_USER_AGENT' => 'MiSuperNavegador/1.0',
));
```

También puedes reemplazar cabeceras *HTTP* en base a la petición:

```
$cliente->request('GET', '/', array(), array(
    'HTTP_HOST'       => 'en.ejemplo.com',
    'HTTP_USER_AGENT' => 'MiSuperNavegador/1.0',
));
```

Truco: Para proveer tu propio cliente, reemplaza el parámetro `test.client.class`, o define un servicio `test.client`.

2.20.7 Aprende más en el recetario

- *Cómo simular autenticación HTTP en una prueba funcional* (Página 365)
- *Cómo probar la interacción de varios clientes* (Página 365)
- *Cómo utilizar el generador de perfiles en una prueba funcional* (Página 366)

2.21 Validando

La validación es una tarea muy común en aplicaciones web. Los datos introducidos en formularios se tienen que validar. Los datos también se deben validar antes de escribirlos en una base de datos o pasarlos a un servicio web.

Symfony2 viene con un componente Validator que facilita esta tarea transparentemente. Este componente está basado en la especificación JSR303 Bean Validation. ¿Qué? ¿Una especificación de Java en *PHP*? Has oído bien, pero no es tan malo como suena. Vamos a ver cómo se puede utilizar en *PHP*.

2.21.1 Fundamentos de la validación

La mejor manera de entender la validación es verla en acción. Para empezar, supongamos que hemos creado un objeto plano en *PHP* el cual en algún lugar tiene que utilizar tu aplicación:

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

class Autor
{
    public $nombre;
}
```

Hasta ahora, esto es sólo una clase ordinaria que sirve a algún propósito dentro de tu aplicación. El objetivo de la validación es decir si o no los datos de un objeto son válidos. Para que esto funcione, debes configurar una lista de reglas (llamada *constraints -restricciones* (Página 153)) que el objeto debe seguir para ser válido. Estas reglas se pueden especificar a través de una serie de formatos diferentes (YAML, XML, anotaciones o PHP).

Por ejemplo, para garantizar que la propiedad `$nombre` no esté vacía, agrega lo siguiente:

- *YAML*

  ```
  # src/Acme/BlogBundle/Resources/config/validation.yml
  Acme\BlogBundle\Entity\Autor:
      properties:
          nombre:
              - NotBlank: ~
  ```

- *Annotations*

  ```
  // src/Acme/BlogBundle/Entity/Autor.php
  use Symfony\Component\Validator\Constraints as Assert;

  class Autor
  {
  ```

```
    /**
     * @Assert\NotBlank()
     */
    public $nombre;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Autor">
        <property name="nombre">
            <constraint name="NotBlank" />
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Autor
{
    public $nombre;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('nombre', new NotBlank());
    }
}
```

Truco: Las propiedades protegidas y privadas también se pueden validar, así como los métodos "get" (consulta la sección *Objetivos de restricción* (Página 155)).

Usando el servicio `validador`

A continuación, para validar realmente un objeto `Autor`, utiliza el método `validate` del servicio `validador` (clase `Symfony\Component\Validator\Validator`). El trabajo del `validador` es fácil: lee las restricciones (es decir, las reglas) de una clase y comprueba si los datos en el objeto satisfacen esas restricciones. Si la validación falla, devuelve un arreglo de errores. Toma este sencillo ejemplo desde el interior de un controlador:

```
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Autor;
// ...

public function indexAction()
{
    $autor = new Autor();
    // ... haz algo con el objeto $autor
```

```
    $validador = $this->get('validator');
    $errores = $validador->validate($autor);

    if (count($errores) > 0) {
        return new Response(print_r($errores, true));
    } else {
        return new Response('¡El autor es válido! ¡Sí!');
    }
}
```

Si la propiedad `$nombre` está vacía, verás el siguiente mensaje de error:

```
Acme\BlogBundle\Autor.nombre:
    Este valor no debe estar en blanco
```

Si insertas un valor en la propiedad `nombre`, aparecerá el satisfactorio mensaje de éxito.

Truco: La mayor parte del tiempo, no interactúas directamente con el servicio `validador` o necesitas preocuparte por imprimir los errores. La mayoría de las veces, vas a utilizar la validación indirectamente al manejar los datos de formularios presentados. Para más información, consulta la sección *Validación y formularios* (Página 152).

También puedes pasar la colección de errores a una plantilla.

```
if (count($errores) > 0) {
    return $this->render('AcmeBlogBundle:Autor:validar.html.twig', array(
        'errores' => $errores,
    ));
} else {
    // ...
}
```

Dentro de la plantilla, puedes sacar la lista de errores exactamente como la necesites:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Autor/validate.html.twig #}

<h3>El autor tiene los siguientes errores</h3>
<ul>
{% for error in errores %}
    <li>{{ error.mensaje }}</li>
{% endfor %}
</ul>
```

- *PHP*

```
<!-- src/Acme/BlogBundle/Resources/views/Autor/validar.html.php -->

<h3>El autor tiene los siguientes errores</h3>
<ul>
<?php foreach ($errores as $error): ?>
    <li><?php echo $error->getMessage() ?></li>
<?php endforeach; ?>
</ul>
```

Nota: Cada error de validación (conocido cómo "violación de restricción"), está representado por un objeto `Symfony\Component\Validator\ConstraintViolation`.

Validación y formularios

Puedes utilizar el servicio `validator` en cualquier momento para validar cualquier objeto. En realidad, sin embargo, por lo general al trabajar con formularios vas a trabajar con el `validador` indirectamente. La biblioteca de formularios de *Symfony* utiliza internamente el servicio `validador` para validar el objeto subyacente después de que los valores se han presentado y vinculado. Las violaciones de restricción en el objeto se convierten en objetos `FieldError` los cuales puedes mostrar fácilmente en tu formulario. El flujo de trabajo típico en la presentación del formulario se parece a lo siguiente visto desde el interior de un controlador:

```
use Acme\BlogBundle\Entity\Autor;
use Acme\BlogBundle\Form\AutorType;
use Symfony\Component\HttpFoundation\Request;
// ...

public function updateAction(Request $peticion)
{
    $autor = new Acme\BlogBundle\Entity\Autor();
    $formulario = $this->createForm(new AutorType(), $author);

    if ($peticion->getMethod() == 'POST') {
        $formulario->bindRequest($peticion);

        if ($formulario->isValid()) {
            // validación pasada, haz algo con el objeto $autor

            $this->redirect($this->generateUrl('...'));
        }
    }

    return $this->render('BlogBundle:Autor:formulario.html.twig', array(
        'formulario' => $formulario->createView(),
    ));
}
```

Nota: Este ejemplo utiliza un formulario de la clase `AutorType`, el cual no mostramos aquí.

Para más información, consulta el capítulo *Formularios* (Página 160).

2.21.2 Configurando

El validador de *Symfony2* está activado por omisión, pero debes habilitar explícitamente las anotaciones si estás utilizando el método de anotación para especificar tus restricciones:

- *YAML*

```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:validation enable_annotations="true" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array('validation' => array(
    'enable_annotations' => true,
)));
```

2.21.3 Restricciones

El `validador` está diseñado para validar objetos contra *restricciones* (es decir, reglas). A fin de validar un objeto, basta con asignar una o más restricciones a tu clase y luego pasarla al servicio `validador`.

Detrás del escenario, una restricción es simplemente un objeto PHP que hace una declaración afirmativa. En la vida real, una restricción podría ser: "El pastel no se debe quemar". En *Symfony2*, las restricciones son similares: son aserciones de que una condición es verdadera. Dado un valor, una restricción te dirá si o no el valor se adhiere a las reglas de tu restricción.

Restricciones compatibles

Symfony2 viene con un gran número de las más comunes restricciones necesarias. La lista completa de restricciones con detalles está disponible en la *sección referencia de restricciones* (Página 514).

Configurando restricciones

Algunas restricciones, como *NotBlank* (Página 514), son simples, mientras que otras, como la restricción *Choice* (Página 535), tienen varias opciones de configuración disponibles. Supongamos que la clase `Autor` tiene otra propiedad, `género` que se puede configurar como "masculino" o "femenino":

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        genero:
            - Choice: { choices: [masculino, femenino], message: Elige un género válido. }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Choice(
     *     choices = { "masculino", "femenino" },
     *     message = "Elige un género válido."
     * )
     */
    public $genero;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

        <class name="Acme\BlogBundle\Entity\Autor">
            <property name="genero">
                <constraint name="Choice">
                    <option name="choices">
                        <value>masculino</value>
                        <value>femenino</value>
                    </option>
                    <option name="message">Elige un género válido.</option>
                </constraint>
            </property>
        </class>
    </constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Autor
{
    public $genero;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('genero', new Choice(array(
            'choices' => array('masculino', 'femenino'),
            'message' => 'Elige un género.',
        )));
    }
}
```

Las opciones de una restricción siempre se pueden pasar como una matriz. Algunas restricciones, sin embargo, también te permiten pasar el valor de una opción "*predeterminada*", en lugar del arreglo. En el caso de la restricción `Choice`, las `opciones` se pueden especificar de esta manera.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        genero:
            - Choice: [masculino, femenino]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Choice({"masculino", "femenino"})
     */
    protected $genero;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Autor">
        <property name="genero">
            <constraint name="Choice">
                <value>masculino</value>
                <value>femenino</value>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Choice;

class Autor
{
    protected $genero;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('genero', new Choice(array('masculino', 'femenino')));
    }
}
```

Esto, simplemente está destinado a hacer que la configuración de las opciones más comunes de una restricción sea más breve y rápida.

Si alguna vez no está seguro de cómo especificar una opción, o bien consulta la documentación de la *API* por la restricción o juega a lo seguro pasando siempre las opciones en un arreglo (el primer método se muestra más arriba).

2.21.4 Objetivos de restricción

Las restricciones se pueden aplicar a una propiedad de clase (por ejemplo, `nombre`) o a un método captador público (por ejemplo `getNombreCompleto`). El primero es el más común y fácil de usar, pero el segundo te permite especificar reglas de validación más complejas.

Propiedades

La validación de propiedades de clase es la técnica de validación más básica. *Symfony2* te permite validar propiedades privadas, protegidas o públicas. El siguiente listado muestra cómo configurar la propiedad `$nombreDePila` de una clase `Autor` para que por lo menos tenga 3 caracteres.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
```

```
        nombreDePila:
            - NotBlank: ~
            - MinLength: 3
```

- *Annotations*

```
// Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $nombreDePila;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="nombreDePila">
        <constraint name="NotBlank" />
        <constraint name="MinLength">3</constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class Autor
{
    private $nombreDePila;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('nombreDePila', new NotBlank());
        $metadatos->addPropertyConstraint('nombreDePila', new MinLength(3));
    }
}
```

Captadores

Las restricciones también se pueden aplicar al valor devuelto por un método. *Symfony2* te permite agregar una restricción a cualquier método público cuyo nombre comience con `get` o `is`. En esta guía, ambos métodos de este tipo son conocidos como "captadores" o `getters`.

La ventaja de esta técnica es que te permite validar el objeto de forma dinámica. Por ejemplo, supongamos que quieres asegurarte de que un campo de contraseña no coincide con el nombre del usuario (por razones de seguridad). Puedes hacerlo creando un método `isPaseLegal`, a continuación, acertar que este método debe devolver `true`:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    getters:
        paseLegal:
            - "True": { message: "La contraseña no puede coincidir con tu nombre" }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\True(message = "La contraseña no debe coincidir con tu nombre")
     */
    public function isPaseLegal()
    {
        // devuelve true o false
    }
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <getter property="paseLegal">
        <constraint name="True">
            <option name="message">La contraseña no debe coincidir con tu nombre</option>
        </constraint>
    </getter>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\True;

class Autor
{
    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addGetterConstraint('paseLegal', new True(array(
            'message' => 'La contraseña no debe coincidir con tu nombre',
        )));
    }
}
```

Ahora, crea el método `isPaseLegal()` e incluye la lógica que necesites:

```
public function isPaseLegal()
{
    return ($this->nombreDePila != $this->password);
}
```

Nota: El ojo perspicaz se habrá dado cuenta de que el prefijo del captador (`get` o `is`) se omite en la asignación. Esto te permite mover la restricción a una propiedad con el mismo nombre más adelante (o viceversa) sin cambiar la lógica de validación.

Clases

Algunas restricciones se aplican a toda la clase que se va a validar. Por ejemplo, la restricción *Retrollamada* (Página 551) es una restricción que se aplica a la clase en sí misma: Cuando se valide esa clase, los métodos especificados por esta restricción se ejecutarán simplemente para que cada uno pueda proporcionar una validación más personalizada.

2.21.5 Validando grupos

Hasta ahora, hemos sido capaces de agregar restricciones a una clase y consultar si o no esa clase pasa todas las restricciones definidas. En algunos casos, sin embargo, tendrás que validar un objeto contra únicamente *algunas* restricciones de esa clase. Para ello, puedes organizar cada restricción en uno o más "grupos de validación", y luego aplicar la validación contra un solo grupo de restricciones.

Por ejemplo, supongamos que tienes una clase `Usuario`, la cual se usa más adelante tanto cuando un usuario se registra como cuando un usuario actualiza su información de contacto:

- *YAML*

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\User:
    properties:
        email:
            - Email: { groups: [registration] }
        password:
            - NotBlank: { groups: [registration] }
            - MinLength: { limit: 7, groups: [registration] }
        ciudad:
            - MinLength: 2
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/User.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface
use Symfony\Component\Validator\Constraints as Assert;

class User implements UserInterface
{
    /**
    * @Assert\Email(groups={"registration"})
    */
    private $email;

    /**
    * @Assert\NotBlank(groups={"registration"})
    * @Assert\MinLength(limit=7, groups={"registration"})
    */
    private $password;

    /**
    * @Assert\MinLength(2)
    */
```

```
    private $ciudad;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\User">
    <property name="email">
        <constraint name="Email">
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
    </property>
    <property name="password">
        <constraint name="NotBlank">
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
        <constraint name="MinLength">
            <option name="limit">7</option>
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
    </property>
    <property name="ciudad">
        <constraint name="MinLength">7</constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/User.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class User
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('email', new Email(array(
            'groups' => array('registration')
        )));

        $metadata->addPropertyConstraint('password', new NotBlank(array(
            'groups' => array('registration')
        )));
        $metadata->addPropertyConstraint('password', new MinLength(array(
            'limit'  => 7,
            'groups' => array('registration')
        )));

        $metadatos->addPropertyConstraint('ciudad', new MinLength(3));
```

```
        }
    }
```

Con esta configuración, hay dos grupos de validación:

- `Default` - contiene las restricciones no asignadas a ningún otro grupo;
- `alta` - contiene restricciones sólo en los campos de `correo electrónico` y `contraseña`.

Para decir al validador que use un grupo específico, pasa uno o más nombres de grupo como segundo argumento al método `validate()`:

```
$errores = $validador->validate($autor, array('registro'));
```

Por supuesto, por lo general vas a trabajar con la validación indirectamente a través de la biblioteca de formularios. Para obtener información sobre cómo utilizar la validación de grupos dentro de los formularios, consulta *Validando grupos* (Página 166).

2.21.6 Consideraciones finales

El `validador` de *Symfony2* es una herramienta poderosa que puedes aprovechar para garantizar que los datos de cualquier objeto son "válidos". El poder detrás de la validación radica en las "restricciones", las cuales son reglas que se pueden aplicar a propiedades o métodos captadores de tu objeto. Y mientras más utilices la plataforma de validación indirectamente cuando uses formularios, recordarás que puedes utilizarla en cualquier lugar para validar cualquier objeto.

2.21.7 Aprende más en el recetario

- *Cómo crear una restricción de validación personalizada* (Página 323)

2.22 Formularios

Utilizar formularios *HTML* es una de las más comunes - y desafiantes - tareas de un desarrollador web. *Symfony2* integra un componente `Form` que se ocupa de facilitarnos la utilización de formularios. En este capítulo, construirás un formulario complejo desde el principio, del cual, de paso, aprenderás las características más importantes de la biblioteca de formularios.

Nota: El componente `Form` de *Symfony* es una biblioteca independiente que se puede utilizar fuera de los proyectos *Symfony2*. Para más información, consulta el Componente Form de Symfony2 en Github.

2.22.1 Creando un formulario simple

Supongamos que estás construyendo una sencilla aplicación de tareas pendientes que necesita mostrar tus "pendientes". Debido a que tus usuarios tendrán que editar y crear tareas, tienes que crear un formulario. Pero antes de empezar, vamos a concentrarnos en la clase genérica `Tarea` que representa y almacena los datos para una sola tarea:

```
// src/Acme/TareaBundle/Entity/Tarea.php
namespace Acme\TareaBundle\Entity;

class Tarea
{
```

```
    protected $tarea;

    protected $fechaVencimiento;

    public function getTarea()
    {
        return $this->tarea;
    }
    public function setTarea($tarea)
    {
        $this->tarea = $tarea;
    }

    public function getFechaVencimiento()
    {
        return $this->fechaVencimiento;
    }
    public function setfechaVencimiento(\DateTime $fechaVencimiento = null)
    {
        $this->fechaVencimiento = $fechaVencimiento;
    }
}
```

Nota: Si estás codificando este ejemplo, primero crea el `AcmeTareaBundle` ejecutando la siguiente orden (aceptando todas las opciones predeterminadas):

```
php app/console generate:bundle --namespace=Acme/TareaBundle
```

Esta clase es una "antiguo objeto *PHP* sencillo", ya que, hasta ahora, no tiene nada que ver con *Symfony* o cualquier otra biblioteca. Es simplemente un objeto PHP normal que directamente resuelve un problema dentro de *tu* aplicación (es decir, la necesidad de representar una tarea pendiente en tu aplicación). Por supuesto, al final de este capítulo, serás capaz de enviar datos a una instancia de `Tarea` (a través de un formulario), validar sus datos, y persistirla en una base de datos.

Construyendo el formulario

Ahora que has creado una clase `Tarea`, el siguiente paso es crear y reproducir el formulario *HTML* real. En *Symfony2*, esto se hace construyendo un objeto `Form` y luego reproduciéndolo en una plantilla. Por ahora, esto se puede hacer en el interior de un controlador:

```
// src/Acme/TareaBundle/Controller/DefaultController.php
namespace Acme\TareaBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TareaBundle\Entity\Tarea;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function nuevaAction(Request $peticion)
    {
        // crea una tarea y proporciona algunos datos ficticios para este ejemplo
        $tarea = new Tarea();
        $tarea->setTarea('Escribir una entrada para el blog');
        $tarea->setFechaVencimiento(new \DateTime('tomorrow'));
```

```
        $form = $this->createFormBuilder($tarea)
            ->add('tarea', 'text')
            ->add('fechaVencimiento', 'date')
            ->getForm();

        return $this->render('AcmeTareaBundle:Default:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

Truco: Este ejemplo muestra cómo crear el formulario directamente en el controlador. Más tarde, en la sección "*Creando clases Form* (Página 172)", aprenderás cómo construir tu formulario en una clase independiente, lo cual es muy recomendable puesto que vuelve reutilizable tu formulario.

La creación de un formulario requiere poco código relativamente, porque los objetos `form` de *Symfony2* se construyen con un "constructor de formularios". El propósito del constructor del formulario es que puedas escribir sencillas "recetas" de formularios, y hacer todo el trabajo pesado, de hecho construye el formulario.

En este ejemplo, hemos añadido dos campos al formulario - `tarea` y `fechaVencimiento` - que corresponden a las propiedades `tarea` y `fechaVencimiento` de la clase `Tarea`. También has asignado a cada uno un "tipo" (por ejemplo, `text`, `date`), que, entre otras cosas, determinan qué etiqueta de formulario *HTML* se reproduce para ese campo.

Symfony2 viene con muchos tipos integrados que explicaremos en breve (consulta *Tipos de campo integrados* (Página 166)).

Reproduciendo el `Form`

Ahora que hemos creado el formulario, el siguiente paso es reproducirlo. Lo puedes hacer fácilmente pasando un objeto especial del formulario, `view`, a tu plantilla (consulta el `$form->createView()` en el controlador de arriba) y utilizando un conjunto de funciones ayudantes de formulario:

- *Twig*

```
{# src/Acme/TareaBundle/Resources/views/Default/nueva.html.twig #}

<form action="{{ path('tarea_nueva') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- src/Acme/TareaBundle/Resources/views/Default/nueva.html.php -->

<form action="<?php echo $view['router']->generate('tarea_nueva') ?>" method="post" <?php echo $
    <?php echo $view['form']->widget($form) ?>

    <input type="submit" />
</form>
```

Nota: Este ejemplo asume que has creado una ruta llamada `tarea_nueva` que apunta a la `AcmeTareaBundle:Default:nueva` controlador creado anteriormente.

¡Eso es todo! Al imprimir `form_widget(form)`, se reproduce cada campo en el formulario, junto con la etiqueta y un mensaje de error (si lo hay). Tan fácil como esto, aunque no es muy flexible (todavía). Por lo general, querrás reproducir individualmente cada campo del formulario para que puedas controlar la apariencia del formulario. Aprenderás cómo hacerlo en la sección "*Reproduciendo un formulario en una plantilla* (Página 169)".

Antes de continuar, observa cómo el campo de entrada `tarea` reproducido tiene el valor de la propiedad `tarea` del objeto `$tarea` (es decir, "Escribir una entrada del blog"). El primer trabajo de un formulario es: tomar datos de un objeto y traducirlos a un formato idóneo para reproducirlos en un formulario *HTML*.

Truco: El sistema de formularios es lo suficientemente inteligente como para acceder al valor de la propiedad protegida `tarea` a través de los métodos `getTarea()` y `setTarea()` de la clase `Tarea`. A menos que una propiedad sea pública, *debe* tener métodos "captadores" y "definidores" para que el componente `Form` pueda obtener y fijar datos en la propiedad. Para una propiedad booleana, puedes utilizar un método "isser" (por "es servicio", por ejemplo, `isPublicado()`) en lugar de un captador (por ejemplo, `getPublicado()`).

Procesando el envío de formularios

El segundo trabajo de un `Form` es traducir los datos enviados por el usuario a las propiedades de un objeto. Para lograrlo, los datos presentados por el usuario deben estar vinculados al formulario. Añade las siguientes funcionalidad a tu controlador:

```
// ...

public function nuevaAction(Request $peticion)
{
    // sólo configura un objeto $tarea fresco (remueve los datos de prueba)
    $tarea = new Tarea();

    $formulario = $this->createFormBuilder($tarea)
        ->add('tarea', 'text')
        ->add('fechaVencimiento', 'date')
        ->getForm();

    if ($peticion->getMethod() == 'POST') {
        $formulario->bindRequest($peticion);

        if ($formulario->isValid()) {
            // realiza alguna acción, tal como guardar la tarea en la base de datos

            return $this->redirect($this->generateUrl('tarea_exito'));
```

```
        }
    }

    // ...
}
```

Ahora, cuando se presente el formulario, el controlador vincula al formulario los datos presentados, los cuales se traducen en los nuevos datos de las propiedades `tarea` y `fechaVencimiento` del objeto `$tarea`. Todo esto ocurre a través del método `bindRequest()`.

Nota: Tan pronto como se llama a `bindRequest()`, los datos presentados se transfieren inmediatamente al objeto subyacente. Esto ocurre independientemente de si los datos subyacentes son válidados realmente.

Este controlador sigue un patrón común para el manejo de formularios, y tiene tres posibles rutas:

1. Inicialmente, cuando, se carga el formulario en un navegador, el método de la petición es `GET`, lo cual significa simplemente que se debe crear y reproducir el formulario;
2. Cuando el usuario envía el formulario (es decir, el método es `POST`), pero los datos presentados no son válidos (la validación se trata en la siguiente sección), el formulario es vinculado y, a continuación reproducido, esta vez mostrando todos los errores de validación;
3. Cuando el usuario envía el formulario con datos válidos, el formulario es vinculado y en ese momento tienes la oportunidad de realizar algunas acciones usando el objeto `$tarea` (por ejemplo, persistirlo a la base de datos) antes de redirigir al usuario a otra página (por ejemplo, una página de "agradecimiento" o "éxito").

Nota: Redirigir a un usuario después de un exitoso envío de formularios evita que el usuario pueda hacer clic en "actualizar" y volver a enviar los datos.

2.22.2 Validando formularios

En la sección anterior, aprendiste cómo se puede presentar un formulario con datos válidos o no válidos. En *Symfony2*, la validación se aplica al objeto subyacente (por ejemplo, `Tarea`). En otras palabras, la cuestión no es si el "formulario" es válido, sino más bien si el objeto `$tarea` es válido después de aplicarle los datos enviados en el formulario. Invocar a `$formulario->isValid()` es un atajo que pregunta al objeto `$tarea` si tiene datos válidos o no.

La validación se realiza añadiendo un conjunto de reglas (llamadas restricciones) a una clase. Para ver esto en acción, añade restricciones de validación para que el campo `tarea` no pueda estar vacío y el campo `fechaVencimiento` no pueda estar vacío y debe ser un objeto `\DateTime` válido.

- *YAML*

```
# Acme/TareaBundle/Resources/config/validation.yml
Acme\TareaBundle\Entity\Tarea:
    properties:
        tarea:
            - NotBlank: ~
        fechaVencimiento:
            - NotBlank: ~
            - Type: \DateTime
```

- *Annotations*

```
// Acme/TareaBundle/Entity/Tarea.php
use Symfony\Component\Validator\Constraints as Assert;
```

```
class Tarea
{
    /**
     * @Assert\NotBlank()
     */
    public $tarea;

    /**
     * @Assert\NotBlank()
     * @Assert\Type("\DateTime")
     */
    protected $fechaVencimiento;
}
```

- *XML*

```
<!-- Acme/TareaBundle/Resources/config/validation.xml -->
<class name="Acme\TareaBundle\Entity\Tarea">
    <property name="tarea">
        <constraint name="NotBlank" />
    </property>
    <property name="fechaVencimiento">
        <constraint name="NotBlank" />
        <constraint name="Type">
            <value>\DateTime</value>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// Acme/TareaBundle/Entity/Tarea.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Type;

class Tarea
{
    // ...

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('tarea', new NotBlank());

        $metadatos->addPropertyConstraint('fechaVencimiento', new NotBlank());
        $metadatos->addPropertyConstraint('fechaVencimiento', new Type('\DateTime'));
    }
}
```

¡Eso es todo! Si vuelves a enviar el formulario con datos no válidos, verás replicados los errores correspondientes en el formulario.

Validación HTML5

A partir de HTML5, muchos navegadores nativamente pueden imponer ciertas restricciones de validación en el lado del cliente. La validación más común se activa al reproducir un atributo `required` en los campos que son obligatorios. Para el navegador compatible con HTML5, esto se traducirá en un mensaje nativo del navegador que muestra si el usuario intenta enviar el formulario con ese campo en blanco.
Los formularios generados sacan el máximo provecho de esta nueva característica añadiendo atributos *HTML* razonables que desencadenan la validación. La validación del lado del cliente, sin embargo, se puede desactivar añadiendo el atributo `novalidate` de la etiqueta `form` o `formnovalidate` a la etiqueta de envío. Esto es especialmente útil cuando deseas probar tus limitaciones en el lado del la validación del servidor, pero su navegador las previene, por ejemplo, la presentación de campos en blanco.

La validación es una característica muy poderosa de *Symfony2* y tiene su propio *capítulo dedicado* (Página 149).

Validando grupos

Truco: Si no estás utilizando la *validación de grupos* (Página 158), entonces puedes saltarte esta sección.

Si tu objeto aprovecha la *validación de grupos* (Página 158), tendrás que especificar la validación de grupos que utiliza tu formulario:

```
$form = $this->createFormBuilder($usuarios, array(
    'validation_groups' => array('registro'),
))->add(...)
;
```

Si vas a crear *clases form* (Página 172) (una buena práctica), entonces tendrás que agregar lo siguiente al método `getDefaultOptions()`:

```
public function getDefaultOptions(array $opciones)
{
    return array(
        'validation_groups' => array('registro')
    );
}
```

En ambos casos, *sólo* se utilizará el grupo de validación `registro` para validar el objeto subyacente.

2.22.3 Tipos de campo integrados

Symfony estándar viene con un gran grupo de tipos de campo que cubre todos los campos de formulario comunes y tipos de datos necesarios:

Campos de texto

- *text* (Página 502)
- *textarea* (Página 504)
- *email* (Página 475)
- *integer* (Página 483)
- *money* (Página 490)

- *number* (Página 492)
- *password* (Página 494)
- *percent* (Página 496)
- *search* (Página 501)
- *url* (Página 510)

Campos de elección

- *choice* (Página 462)
- *entity* (Página 476)
- *country* (Página 466)
- *language* (Página 485)
- *locale* (Página 488)
- *timezone* (Página 507)

Campos de fecha y hora

- *date* (Página 469)
- *datetime* (Página 472)
- *time* (Página 505)
- *birthday* (Página 459)

Otros campos

- *checkbox* (Página 461)
- *file* (Página 480)
- *radio* (Página 498)

Campos de grupos

- *collection* (Página 466)
- *repeated* (Página 499)

Campos ocultos

- *hidden* (Página 483)
- *csrf* (Página 468)

Campos base

- *field* (Página 482)
- *form* (Página 483)

También puedes crear tus propios tipos de campo personalizados. Este tema se trata en el artículo "*Cómo crear un tipo de campo de formulario personalizado* (Página 322)" del recetario.

Opciones del tipo de campo

Cada tipo de campo tiene una serie de opciones que puedes utilizar para configurarlo. Por ejemplo, el campo `fechaVencimiento` se está traduciendo como 3 cajas de selección. Sin embargo, puedes configurar el *campo de fecha* (Página 469) para que sea interpretado como un cuadro de texto (donde el usuario introduce la fecha como una cadena en el cuadro):

```
->add('fechaVencimiento', 'date', array('widget' => 'single_text'))
```

Cada tipo de campo tiene una diferente serie de opciones que le puedes pasar. Muchas de ellas son específicas para el tipo de campo y puedes encontrar los detalles en la documentación de cada tipo.

La opción `required`

La opción más común es la opción `required`, la cual puedes aplicar a cualquier campo. De manera predeterminada, la opción `required` está establecida en `true`, lo cual significa que los navegadores preparados para HTML5 aplicarán la validación en el cliente si el campo se deja en blanco. Si no deseas este comportamiento, establece la opción `required` en tu campo a `false` o *desactiva la validación de HTML5* (Página 165).
También ten en cuenta que al establecer la opción `required` a `true` **no** resultará en aplicar la validación de lado del servidor. En otras palabras, si un usuario envía un valor en blanco para el campo (ya sea con un navegador antiguo o un servicio web, por ejemplo), será aceptado como un valor válido a menos que utilices la validación de restricción `NotBlank` o `NotNull` de *Symfony*.
En otras palabras, la opción `required` es "agradable", pero ciertamente *siempre* se debe utilizar de lado del servidor.

2.22.4 Adivinando el tipo de campo

Ahora que has añadido metadatos de validación a la clase `Tarea`, *Symfony* ya sabe un poco sobre tus campos. Si le permites, *Symfony* puede "adivinar" el tipo de tu campo y configurarlo por ti. En este ejemplo, *Symfony* puede adivinar a partir de las reglas de validación de ambos campos, `tarea` es un campo de texto normal y `fechaVencimiento` es un campo `date`:

```
public function nuevaAction()
{
    $tarea = new Tarea();

    $formulario = $this->createFormBuilder($tarea)
        ->add('tarea')
        ->add('fechaVencimiento', null, array('widget' => 'single_text'))
```

```
        ->getForm();
}
```

El "adivino" se activa cuando omites el segundo argumento del método `add()` (o si le pasas `null`). Si pasas una matriz de opciones como tercer argumento (hecho por `fechaVencimiento` arriba), estas opciones se aplican al campo adivinado.

Prudencia: Si tu formulario utiliza una validación de grupo específica, el adivinador del tipo de campo seguirá considerando *todas* las restricciones de validación cuando adivina el tipo de campo (incluyendo las restricciones que no son parte de la validación de grupo utilizada).

Opciones para adivinar el tipo de campo

Además de adivinar el "tipo" de un campo, *Symfony* también puede tratar de adivinar los valores correctos de una serie de opciones de campo.

Truco: Cuando estas opciones están establecidas, el campo se reproducirá con atributos *HTML* especiales proporcionados para validación *HTML5* en el cliente. Sin embargo, no genera el equivalente de las restricciones de lado del servidor (por ejemplo, `Assert\MaxLength`). Y aunque tendrás que agregar manualmente la validación de lado del servidor, estas opciones del tipo de campo entonces se pueden adivinar a partir de esa información.

- `required`: La opción `required` se puede adivinar basándose en las reglas de validación (es decir, el campo es `NotBlank` o `NotNull`) o los metadatos de *Doctrine* (es decir, el campo puede ser nulo - `nullable`). Esto es muy útil, ya que tu validación de lado del cliente se ajustará automáticamente a tus reglas de validación.
- `min_length`: Si el campo es una especie de campo de texto, entonces la opción `min_length` se puede adivinar a partir de las restricciones de validación (si se utiliza `minLength` o `Min`) o de los metadatos de *Doctrine* (vía la longitud del campo).
- `max_length`: Similar a `min_length`, la longitud máxima también se puede adivinar.

Nota: Estas opciones de campo *sólo* se adivinan si estás utilizando *Symfony* para averiguar el tipo de campo (es decir, omitir o pasar `null` como segundo argumento de `add()`).

Si quieres cambiar uno de los valores adivinados, lo puedes redefinir pasando la opción en la matriz de las opciones del campo:

```
->add('tarea', null, array('min_length' => 4))
```

2.22.5 Reproduciendo un formulario en una plantilla

Hasta ahora, has visto cómo se puede reproducir todo el formulario con una sola línea de código. Por supuesto, generalmente necesitarás mucha más flexibilidad al reproducirlo:

- *Twig*

```
{# src/Acme/TareaBundle/Resources/views/Default/nueva.html.twig #}

<form action="{{ path('tarea_nyeva') }}" method="post" {{ form_enctype(form) }}>
    {{ form_errors(form) }}

    {{ form_row(form.tarea) }}
    {{ form_row(form.fechaVencimiento) }}
```

```
    {{ form_rest(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- // src/Acme/TareaBundle/Resources/views/Default/nuevaAction.html.php -->

<form action="<?php echo $view['router']->generate('tarea_nueva') ?>" method="post" <?php echo $
    <?php echo $view['form']->errors($form) ?>

    <?php echo $view['form']->row($form['tarea']) ?>
    <?php echo $view['form']->row($form['fechaVencimiento']) ?>

    <?php echo $view['form']->rest($formulario) ?>

    <input type="submit" />
</form>
```

Echemos un vistazo a cada parte:

- `form_enctype(form)` - Si por lo menos un campo es un campo de carga de archivos, se reproduce el obligado `enctype="multipart/form-data"`;
- `form_errors(form)` - Reproduce cualquier error global para todo el formulario (los errores específicos de campo se muestran junto a cada campo);
- `form_row(form.dueDate)` - Reproduce la etiqueta, cualquier error, y el elemento gráfico *HTML* del formulario para el campo en cuestión (por ejemplo, `fechaVencimiento`), por omisión, en el interior un elemento `div`;
- `form_rest(form)` - Reproduce todos los campos que aún no se han representado. Por lo general es buena idea realizar una llamada a este ayudante en la parte inferior de cada formulario (en caso de haber olvidado sacar un campo o si no quieres preocuparte de reproducir manualmente los campos ocultos). Este ayudante también es útil para tomar ventaja de la *Protección CSRF* (Página 180) automática.

La mayor parte del trabajo la realiza el ayudante `form_row`, el cual de manera predeterminada reproduce la etiqueta, los errores y el elemento gráfico *HTML* de cada campo del formulario dentro de una etiqueta `div`. En la sección *Tematizando formularios* (Página 176), aprenderás cómo puedes personalizar `form_row` en muchos niveles diferentes.

Reproduciendo cada campo a mano

El ayudante `form_row` es magnífico porque rápidamente puedes reproducir cada campo del formulario (y también puedes personalizar el formato utilizado para la "fila"). Pero, puesto que la vida no siempre es tan simple, también puedes reproducir cada campo totalmente a mano. El producto final del siguiente fragmento es el mismo que cuando usas el ayudante `form_row`:

- *Twig*

```
{{ form_errors(form) }}

<div>
    {{ form_label(form.tarea) }}
    {{ form_errors(form.tarea) }}
    {{ form_widget(form.tarea) }}
</div>
```

```
<div>
    {{ form_label(form.fechaVencimiento) }}
    {{ form_errors(form.fechaVencimiento) }}
    {{ form_widget(form.fechaVencimiento) }}
</div>

{{ form_rest(form) }}
```

- *PHP*

```
<?php echo $view['form']->errors($formulario) ?>

<div>
    <?php echo $view['form']->label($form['tarea']) ?>
    <?php echo $view['form']->errors($form['tarea']) ?>
    <?php echo $view['form']->widget($form['tarea']) ?>
</div>

<div>
    <?php echo $view['form']->label($form['fechaVencimiento']) ?>
    <?php echo $view['form']->errors($form['fechaVencimiento']) ?>
    <?php echo $view['form']->widget($form['fechaVencimiento']) ?>
</div>

<?php echo $view['form']->rest($formulario) ?>
```

Si la etiqueta generada automáticamente para un campo no es del todo correcta, la puedes especificar explícitamente:

- *Twig*

```
{{ form_label(form.tarea, 'Descripción de tarea') }}
```

- *PHP*

```
<?php echo $view['form']->label($form['tarea'], 'Descripción de tarea') ?>
```

Por último, algunos tipos de campo tienen más opciones para su representación que puedes pasar al elemento gráfico. Estas opciones están documentadas con cada tipo, pero una de las opciones común es `attr`, la cual te permite modificar los atributos en el elemento del formulario. El siguiente debería añadir la clase `campo_tarea` al campo de entrada de texto reproducido:

- *Twig*

```
{{ form_widget(form.tarea, { 'attr': {'class': 'campo_tarea'} }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['tarea'], array(
    'attr' => array('class' => 'campo_tarea'),
)) ?>
```

Referencia de funciones de plantilla *Twig*

Si estás utilizando *Twig*, hay disponible una referencia completa de las funciones de reproducción de formularios en el *Manual de referencia* (Página 513). Lee esto para conocer todo acerca de los ayudantes y opciones disponibles que puedes utilizar con cada uno.

2.22.6 Creando clases `Form`

Como hemos visto, puedes crear un formulario y utilizarlo directamente en un controlador. Sin embargo, una mejor práctica es construir el formulario en una clase separada, independiente de las clases *PHP*, la cual puedes reutilizar en cualquier lugar de tu aplicación. Crea una nueva clase que albergará la lógica para la construcción del formulario de la tarea:

```
// src/Acme/TareaBundle/Form/Type/TareaType.php

namespace Acme\TareaBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TareaType extends AbstractType
{
    public function buildForm(FormBuilder $generador, array $opciones)
    {
        $generador->add('tarea');
        $generador->add('fechaVencimiento', null, array('widget' => 'single_text'));
    }

    public function getName()
    {
        return 'tarea';
    }
}
```

Esta nueva clase contiene todas las indicaciones necesarias para crear el formulario de la tarea (observa que el método `getName()` debe devolver un identificador único para este "tipo" de formulario). La puedes utilizar para construir rápidamente un objeto formulario en el controlador:

```
// src/Acme/TareaBundle/Controller/DefaultController.php

// agrega esta nueva declaración use en lo alto de la clase
use Acme\TareaBundle\Form\Type\TareaType;

public function nuevaAction()
{
    $tarea = // ...
    $form = $this->createForm(new TareaType(), $tarea);

    // ...
}
```

Colocar la lógica del formulario en su propia clase significa que fácilmente puedes reutilizar el formulario en otra parte del proyecto. Esta es la mejor manera de crear formularios, pero la decisión en última instancia, depende de ti.

Configurando `data_class`

Cada formulario tiene que conocer el nombre de la clase que contiene los datos subyacentes (por ejemplo, `Acme\TareaBundle\Entity\Tarea`). Por lo general, esto sólo se adivina basándose en el objeto pasado como segundo argumento de `createForm` (es decir, `$tarea`). Más tarde, cuando comiences a incorporar formularios, esto ya no será suficiente. Así que, si bien no siempre es necesario, generalmente es buena idea especificar explícitamente la opción `data_class` añadiendo lo siguiente al tipo de tu clase formulario:

```
public function getDefaultOptions(array $opciones)
{
    return array(
        'data_class' => 'Acme\TareaBundle\Entity\Tarea',
    );
}
```

2.22.7 Formularios y *Doctrine*

El objetivo de un formulario es traducir los datos de un objeto (por ejemplo, `Tarea`) a un formulario *HTML* y luego traducir los datos enviados por el usuario al objeto original. Como tal, el tema de la persistencia del objeto `Tarea` a la base de datos es del todo ajeno al tema de los formularios. Pero, si has configurado la clase `Tarea` para persistirla a través de *Doctrine* (es decir, que le has añadido *metadatos de asignación* (Página 117)), entonces persistirla después de la presentación de un formulario se puede hacer cuando el formulario es válido:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($tarea);
    $em->flush();

    return $this->redirect($this->generateUrl('tarea_exito'));
}
```

Si por alguna razón, no tienes acceso a tu objeto `$tarea` original, lo puedes recuperar desde el formulario:

```
$tarea = $formulario->getData();
```

Para más información, consulta el capítulo *ORM de Doctrine* (Página 115).

La clave es entender que cuando el formulario está vinculado, los datos presentados inmediatamente se transfieren al objeto subyacente. Si deseas conservar los datos, sólo tendrás que conservar el objeto en sí (el cual ya contiene los datos presentados).

2.22.8 Integrando formularios

A menudo, querrás crear un formulario que incluye campos de muchos objetos diferentes. Por ejemplo, un formulario de registro puede contener datos que pertenecen a un objeto `Usuario`, así como a muchos objetos `Domicilio`. Afortunadamente, esto es fácil y natural con el componente `Form`.

Integrando un solo objeto

Supongamos que cada `Tarea` pertenece a un simple objeto `Categoría`. Inicia, por supuesto, creando el objeto `Categoría`:

```
// src/Acme/TareaBundle/Entity/Categoria.php
namespace Acme\TareaBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Categoria
{
    /**
     * @Assert\NotBlank()
     */
    public $nombre;
}
```

A continuación, añade una nueva propiedad `categoría` a la clase `Tarea`:

```
// ...

class Tarea
{
    // ...

    /**
     * @Assert\Type(type="Acme\TareaBundle\Entity\Categoria")
     */
    protected $categoria;

    // ...

    public function getCategoria()
    {
        return $this->categoria;
    }

    public function setCategoria(Categoria $categoria = null)
    {
        $this->categoria = $categoria;
    }
}
```

Ahora que hemos actualizado tu aplicación para reflejar las nuevas necesidades, crea una clase formulario para que el usuario pueda modificar un objeto `Categoría`:

```
// src/Acme/TareaBundle/Form/Type/CategoriaType.php
namespace Acme\TareaBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class CategoriaType extends AbstractType
{
    public function buildForm(FormBuilder $generador, array $opciones)
    {
        $generador->add('nombre');
    }

    public function getDefaultOptions(array $opciones)
    {
        return array(
            'data_class' => 'Acme\TareaBundle\Entity\Categoria',
```

```
        );
    }

    public function getNombre()
    {
        return 'categoria';
    }
}
```

El objetivo final es permitir que la `Categoría` de una `Tarea` sea modificada justo dentro del mismo formulario de la tarea. Para lograr esto, añade un campo `categoría` al objeto `TareaType` cuyo tipo es una instancia de la nueva clase `CategoriaType`:

```
public function buildForm(FormBuilder $generador, array $opciones)
{
    // ...

    $generador->add('categoria', new CategoriaType());
}
```

Los campos de `CategoriaType` ahora se pueden reproducir junto a los de la clase `TareaType`. Reproduce los campos de la `Categoría` de la misma manera que los campos de la `Tarea` original:

- *Twig*

```
{# ... #}

<h3>Categoría</h3>
<div class="categoria">
    {{ form_row(form.categoria.nombre) }}
</div>

{{ form_rest(form) }}
{# ... #}
```

- *PHP*

```
<!-- ... -->

<h3>Categoría</h3>
<div class="categoria">
    <?php echo $view['form']->row($formulario['categoria']['nombre']) ?>
</div>

<?php echo $view['form']->rest($formulario) ?>
<!-- ... -->
```

Cuando el usuario envía el formulario, los datos presentados para los campos de la `Categoría` se utilizan para construir una instancia de la `Categoría`, que se establece en el campo `Categoría` de la instancia `Tarea`.

La instancia de `Categoría` es accesible naturalmente vía `$tarea->getCategoria()` y la puedes persistir en la base de datos o utilizarla como necesites.

Integrando una colección de formularios

También puedes integrar una colección de formularios en un formulario. Esto se consigue usando el tipo de campo `collection`. Para más información, consulta *referencia del tipo de campo* (Página 466).

2.22.9 Tematizando formularios

Puedes personalizar cómo se reproduce cada parte de un formulario. Eres libre de cambiar la forma en que se reproduce cada "fila" del formulario, cambiar el formato que sirve para reproducir errores, e incluso personalizar la forma en que se debe reproducir una etiqueta `textarea`. Nada está fuera de límites, y puedes usar diferentes personalizaciones en diferentes lugares.

Symfony utiliza plantillas para reproducir todas y cada una de las partes de un formulario, como las etiquetas `label`, etiquetas `input`, mensajes de error y todo lo demás.

En *Twig*, cada "fragmento" del formulario está representado por un bloque *Twig*. Para personalizar alguna parte de cómo se reproduce un formulario, sólo hay que reemplazar el bloque adecuado.

En PHP, cada "fragmento" del formulario se reproduce vía un archivo de plantilla individual. Para personalizar cualquier parte de cómo se reproduce un formulario, sólo hay que reemplazar la plantilla existente creando una nueva.

Para entender cómo funciona esto, vamos a personalizar el fragmento `form_row` añadiendo un atributo de clase al elemento `div` que rodea cada fila. Para ello, crea un nuevo archivo de plantilla que almacenará el nuevo marcado:

- *Twig*

```
{# src/Acme/TareaBundle/Resources/views/Form/fields.html.twig #}

{% block field_row %}
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock field_row %}
```

- *PHP*

```
<!-- src/Acme/TareaBundle/Resources/views/Form/field_row.html.php -->

<div class="form_row">
    <?php echo $view['form']->label($formulario, $label) ?>
    <?php echo $view['form']->errors($formulario) ?>
    <?php echo $view['form']->widget($formulario, $parameters) ?>
</div>
```

El fragmento `field_row` del formulario se usa cuando se reproduce la mayoría de los campos a través de la función `form_row`. Para decir al componente `Form` que utilice tu nuevo fragmento `field_row` definido anteriormente, añade lo siguiente en la parte superior de la plantilla que reproduce el formulario:

- *Twig*

```
{# src/Acme/TareaBundle/Resources/views/Default/nueva.html.twig #}

{% form_theme form 'AcmeTareaBundle:Form:fields.html.twig' %}

<form ...>
```

- *PHP*

```
<!-- src/Acme/TareaBundle/Resources/views/Default/nueva.html.php -->

<?php $view['form']->setTheme($form, array('AcmeTareaBundle:Form')) ?>
```

```
<form ...>
```

La etiqueta `form_theme` (en *Twig*) "importa" los fragmentos definidos en la plantilla dada y los utiliza al reproducir el formulario. *En otras palabras, cuando se llame a la función ''form_row'* más adelante en esta plantilla, se utilizará el bloque `field_row` de tu tema personalizado (en lugar del bloque `field_row` predeterminado suministrado con *Symfony*).

Para personalizar cualquier parte de un formulario, sólo tienes que reemplazar el fragmento apropiado. Saber exactamente qué bloque sustituir es el tema de la siguiente sección.

Para una explicación más extensa, consulta *Cómo personalizar la reproducción de un formulario* (Página 309).

Nombrando fragmentos de formulario

En *Symfony*, cada parte que se reproduce de un formulario - elementos *HTML* de formulario, errores, etiquetas, etc. - se define en un tema base, el cual es una colección de bloques en *Twig* y una colección de archivos de plantilla en *PHP*.

En *Twig*, cada bloque necesario se define en un solo archivo de plantilla (form_div_base.html.twig) que vive dentro de Twig Bridge. Dentro de este archivo, puedes ver todos los bloques necesarios para reproducir un formulario y cada tipo de campo predeterminado.

En PHP, los fragmentos son archivos de plantilla individuales. De manera predeterminada se encuentran en el directorio `Resources/views/Form` del paquete de la plataforma (ver en GitHub).

El nombre de cada fragmento sigue el mismo patrón básico y se divide en dos partes, separadas por un solo carácter de subrayado (_). Algunos ejemplos son:

- `field_row` - usado por `form_row` para reproducir la mayoría de los campos;
- `textarea_widget` - usado por `form_widget` para reproducir un campo de tipo `textarea`;
- `field_errors` - usado por `form_errors` para reproducir los errores de un campo;

Cada fragmento sigue el mismo patrón básico: `tipo_parte`. La porción `tipo` corresponde al *tipo* del campo que se está reproduciendo (por ejemplo, `textarea`, `checkbox`, `date`, etc.), mientras que la porción `parte` corresponde a *qué* se está reproduciendo (por ejemplo, `label`, `widget`, `errores`, etc.). De manera predeterminada, hay cuatro posibles *partes* de un formulario que puedes reproducir:

`label`	(p. ej. `field_label`)	reproduce la etiqueta del campo
`widget`	(p. ej. `field_widget`)	reproduce el *HTML* del campo
`errors`	(p. ej. `field_errors`)	reproduce los errores del campo
`row`	(p. ej. `field_row`)	reproduce el renglón completo (etiqueta, elemento gráfico y errores)

Nota: En realidad, hay otras 3 *partes* - `filas`, `resto` y `enctype` - pero que rara vez o nunca te tienes que preocupar de cómo reemplazarlas.

Al conocer el tipo de campo (por ejemplo, `textarea`) y cual parte deseas personalizar (por ejemplo, `widget`), puedes construir el nombre del fragmento que se debe redefinir (por ejemplo, `textarea_widget`).

Heredando fragmentos de plantilla

En algunos casos, parece que falta el fragmento que deseas personalizar. Por ejemplo, no hay fragmento `textarea_errors` en los temas predeterminados provistos con *Symfony*. Entonces, ¿cómo se reproducen los errores de un campo `textarea`?

La respuesta es: a través del fragmento `field_errors`. Cuando *Symfony* pinta los errores del tipo `textarea`, primero busca un fragmento `textarea_errors` antes de caer de nuevo al fragmento `field_errors`. Cada tipo de campo tiene un tipo *padre* (el tipo primario del `textarea` es `field`), y *Symfony* utiliza el fragmento para el tipo del padre si no existe el fragmento base.

Por lo tanto, para sustituir *sólo* los errores de los campos `textarea`, copia el fragmento `field_errors`, cámbiale el nombre a `textarea_errors` y personalizarlo. Para sustituir la reproducción predeterminada para error de *todos* los campos, copia y personaliza el fragmento `field_errors` directamente.

Truco: El tipo "padre" de cada tipo de campo está disponible en la *referencia del tipo form* (Página 459) para cada tipo de campo.

Tematizando formularios globalmente

En el ejemplo anterior, utilizaste el ayudante `form_theme` (en *Twig*) para "importar" fragmentos de formulario personalizados *sólo* para ese formulario. También puedes decirle a *Symfony* que importe formularios personalizados a través de tu proyecto.

Twig

Para incluir automáticamente en *todas* las plantillas los bloques personalizados de la plantilla `fields.html.twig` creada anteriormente, modifica el archivo de configuración de tu aplicación:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeTareaBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
        <twig:form>
            <resource>AcmeTaskBundle:Form:fields.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```
// app/config/config.php

$container->loadFromExtension('twig', array(
    'form' => array('resources' => array(
        'AcmeTareaBundle:Form:fields.html.twig',
     ))
    // ...
));
```

Ahora se utilizan todos los bloques dentro de la plantilla `fields.html.twig` a nivel global para definir el formulario producido.

Personalizando toda la salida del formulario en un único archivo con *Twig*

En *Twig*, también puedes personalizar el bloque correcto de un formulario dentro de la plantilla donde se necesita esa personalización:

```
{% extends '::base.html.twig' %}

{# import "_self" como el tema del formulario #}
{% form_theme form _self %}

{# hace la personalización del fragmento del formulario #}
{% block field_row %}
    {# produce la fila del campo personalizado #}
{% endblock field_row %}

{% block contenido %}
    {# ... #}

    {{ form_row(form.tarea) }}
{% endblock %}
```

La etiqueta `{% form_theme form_self %}` permite personalizar bloques directamente dentro de la plantilla que utilizará las personalizaciones. Utiliza este método para crear rápidamente formularios personalizados que sólo son necesarios en una sola plantilla.

PHP

Para incluir automáticamente en *todas* las plantillas personalizadas del directorio *Acme/TareaBundle/Resources/views/Form* creado anteriormente, modifica el archivo de configuración de tu aplicación:

- *YAML*

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'AcmeTareaBundle:Form'
# ...
```

- *XML*

```
<!-- app/config/config.xml -->

<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>AcmeTareaBundle:Form</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
// app/config/config.php

$contenedor->loadFromExtension('framework', array(
    'templating' => array('form' =>
        array('resources' => array(
            'AcmeTareaBundle:Form',
     )))
    // ...
));
```

Cualquier fragmento dentro del directorio *Acme/TareaBundle/Resources/views/Form* ahora se utiliza globalmente para definir la salida del formulario.

2.22.10 Protección CSRF

CSRF (Cross-site request forgery) - o Falsificación de petición en sitios cruzados - es un método por el cual un usuario malintencionado intenta hacer que tus usuarios legítimos, sin saberlo, presenten datos que no tienen la intención de enviar. Afortunadamente, los ataques CSRF se pueden prevenir usando un elemento CSRF dentro de tus formularios.

La buena nueva es que, por omisión, *Symfony* integra y valida elementos CSRF automáticamente. Esto significa que puedes aprovechar la protección CSRF sin hacer nada. De hecho, ¡cada formulario en este capítulo se ha aprovechado de la protección CSRF!

La protección CSRF funciona añadiendo un campo oculto al formulario - por omisión denominado `_token` - el cual contiene un valor que sólo tú y tu usuario conocen. Esto garantiza que el usuario - y no alguna otra entidad - es el que presenta dichos datos. *Symfony* automáticamente valida la presencia y exactitud de este elemento.

El campo `_token` es un campo oculto y será reproducido automáticamente si se incluye la función `form_rest()` de la plantilla, la cual garantiza que se presenten todos los campos producidos.

El elemento CSRF se puede personalizar formulario por formulario. Por ejemplo:

```
class TareaType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $opciones)
    {
        return array(
            'data_class'      => 'Acme\TareaBundle\Entity\Tarea',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // una clave única para ayudar a generar el elemento secreto
            'intention'       => 'task_item',
        );
    }

    // ...
}
```

Para desactivar la protección CSRF, fija la opción `csrf_protection` a false. Las personalizaciones también se pueden hacer a nivel global en tu proyecto. Para más información, consulta la sección *referencia de configuración de formularios*.

Nota: La opción `intention` es opcional pero mejora considerablemente la seguridad del elemento generado produciendo uno diferente para cada formulario.

2.22.11 Consideraciones finales

Ahora ya conoces todos los bloques de construcción necesarios para construir formularios complejos y funcionales para tu aplicación. Cuando construyas formularios, ten en cuenta que el primer objetivo de un formulario es traducir los datos de un objeto (`Tarea`) a un formulario *HTML* para que el usuario pueda modificar esos datos. El segundo objetivo de un formulario es tomar los datos presentados por el usuario y volverlos a aplicar al objeto.

Todavía hay mucho más que aprender sobre el poderoso mundo de los formularios, tal como la forma de *manejar con *Doctrine* los archivos subidos* (Página 294) o cómo crear un formulario donde puedes agregar una serie dinámica de subformularios (por ejemplo, una lista de tareas donde puedes seguir añadiendo más campos a través de *Javascript* antes de presentarlo). Consulta el recetario para estos temas. Además, asegúrate de apoyarte en la documentación *referencia de tipo de campo* (Página 459), que incluye ejemplos de cómo utilizar cada tipo de campo y sus opciones.

2.22.12 Aprende más en el recetario

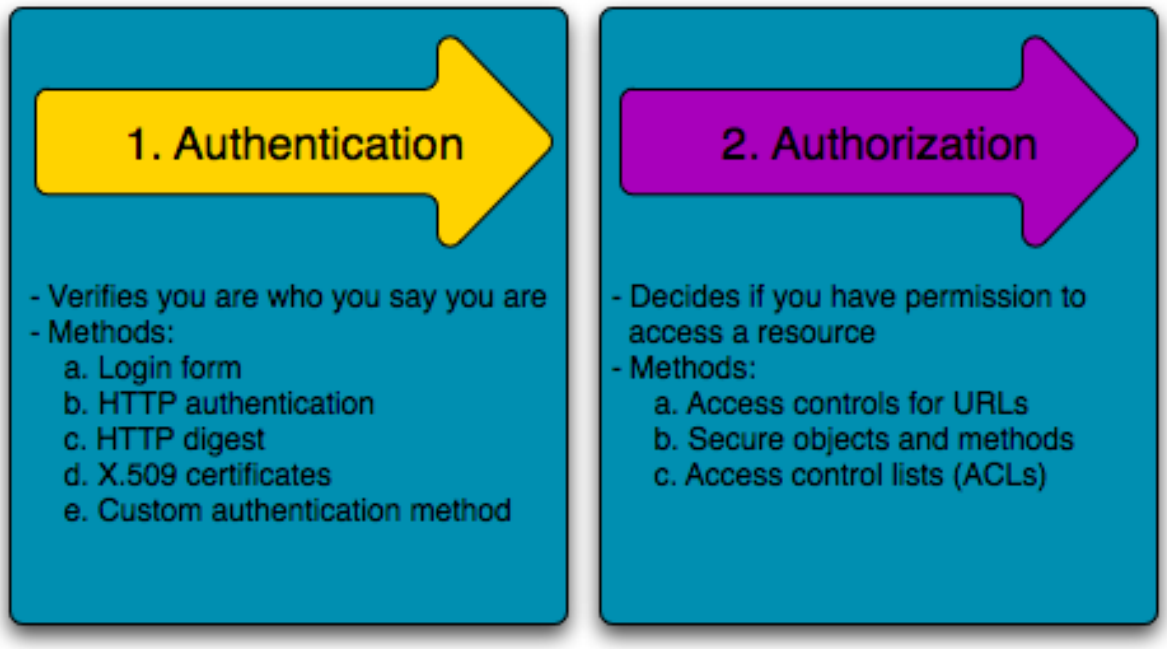
- *Cómo manejar archivos subidos con Doctrine* (Página 294)
- *Referencia del campo file* (Página 480)
- *Creando tipos de campo personalizados* (Página 322)
- *Cómo personalizar la reproducción de un formulario* (Página 309)

2.23 Seguridad

La seguridad es un proceso de dos etapas, cuyo objetivo es evitar que un usuario acceda a un recurso al cual él/ella no debería tener acceso.

En el primer paso del proceso, el sistema de seguridad identifica quién es el usuario obligándolo a presentar algún tipo de identificación. Esto se llama **autenticación**, y significa que el sistema está tratando de averiguar quién eres.

Una vez que el sistema sabe quien eres, el siguiente paso es determinar si deberías tener acceso a un determinado recurso. Esta parte del proceso se llama **autorización**, y significa que el sistema está comprobando para ver si tienes suficientes privilegios para realizar una determinada acción.



Puesto que la mejor manera de aprender es viendo un ejemplo, vamos a zambullirnos en este.

Nota: El componente Security de *Symfony* está disponible como una biblioteca *PHP* independiente para usarla dentro de cualquier proyecto *PHP*.

2.23.1 Ejemplo básico: Autenticación *HTTP*

Puedes configurar el componente de seguridad a través de la configuración de tu aplicación. De hecho, la mayoría de los ajustes de seguridad estándar son sólo cuestión de usar la configuración correcta. La siguiente configuración le dice a *Symfony* que proteja cualquier *URL* coincidente con `/admin/*` y pida al usuario sus credenciales mediante autenticación *HTTP* básica (es decir, el cuadro de dialogo a la vieja escuela de nombre de usuario/contraseña):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

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

- *XML*

```
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <http-basic realm="Demo de área protegida" />
        </firewall>

        <access-control>
            <rule path="^/admin" role="ROLE_ADMIN" />
        </access-control>

        <provider name="in_memory">
            <user name="ryan" password="ryanpass" roles="ROLE_USER" />
            <user name="admin" password="kitten" roles="ROLE_ADMIN" />
        </provider>

        <encoder class="Symfony\Component\Security\Core\User\User" algorithm="plaintext" />
    </config>
</srv:container>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'http_basic' => array(
                'realm' => 'Demo de área protegida',
            ),
        ),
    ),
    'access_control' => array(
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
    'providers' => array(
        'in_memory' => array(
            'users' => array(
                'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
            ),
        ),
    ),
    'encoders' => array(
        'Symfony\Component\Security\Core\User\User' => 'plaintext',
    ),
));
```

Truco: Una distribución estándar de *Symfony* separa la configuración de seguridad en un archivo independiente (por ejemplo, `app/config/security.yml`). Si no tienes un archivo de seguridad por separado, puedes poner la configuración directamente en el archivo de configuración principal (por ejemplo, `app/config/config.yml`).

El resultado final de esta configuración es un sistema de seguridad totalmente funcional que tiene el siguiente aspecto:

- Hay dos usuarios en el sistema (`ryan` y `admin`);
- Los usuarios se autentican a través de la autenticación *HTTP* básica del sistema;
- Cualquier *URL* que coincida con `/admin/*` está protegida, y sólo el usuario `admin` puede acceder a ella;
- Todas las *URL* que *no* coincidan con `/admin/*` son accesibles por todos los usuarios (y nunca se pide al usuario que se registre).

Veamos brevemente cómo funciona la seguridad y cómo entra en juego cada parte de la configuración.

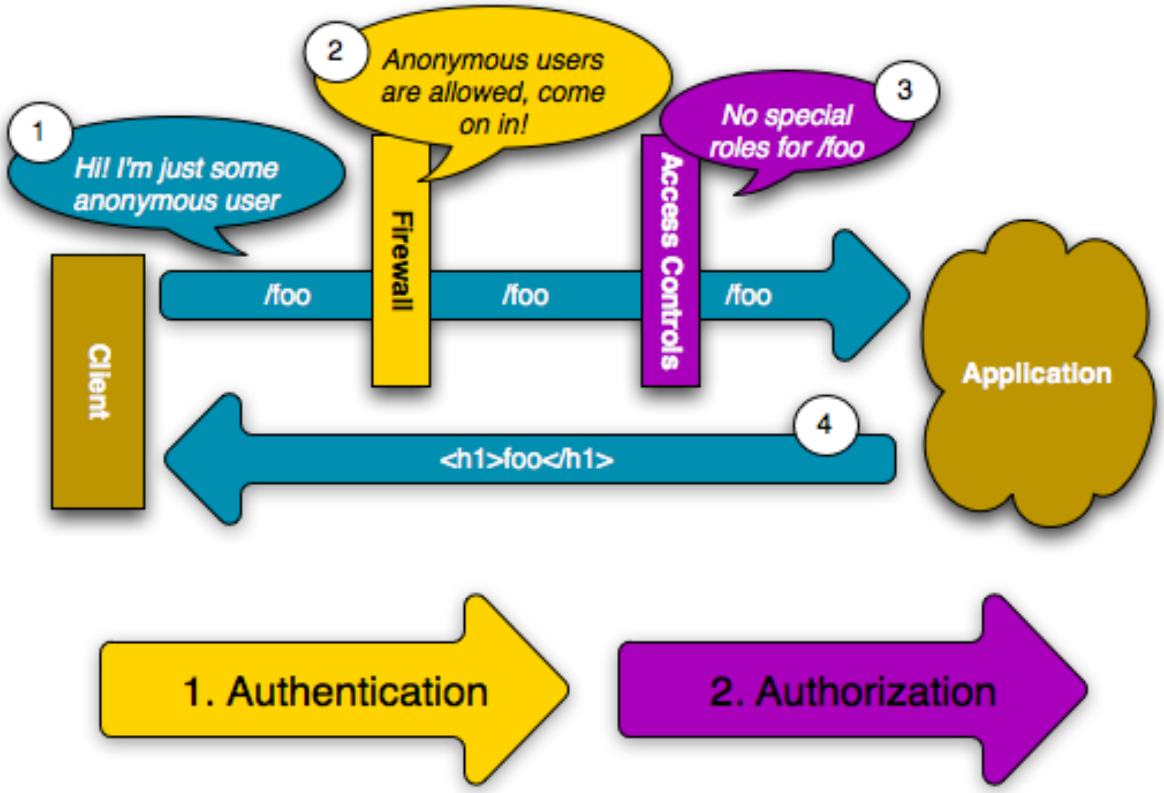
2.23.2 Cómo funciona la seguridad: autenticación y autorización

El sistema de seguridad de *Symfony* trabaja identificando a un usuario (es decir, la autenticación) y comprobando si ese usuario debe tener acceso a un recurso o *URL* específico.

Cortafuegos (autenticación)

Cuando un usuario hace una petición a una *URL* que está protegida por un cortafuegos, se activa el sistema de seguridad. El trabajo del cortafuegos es determinar si el usuario necesita estar autenticado, y si lo hace, enviar una respuesta al usuario para iniciar el proceso de autenticación.

Un cortafuegos se activa cuando la *URL* de una petición entrante concuerda con el `patrón` de la expresión regular configurada en el valor 'config' del cortafuegos. En este ejemplo el `patrón` (`^/`) concordará con *cada* petición entrante. El hecho de que el cortafuegos esté activado *no* significa, sin embargo, que el nombre de usuario de autenticación *HTTP* y el cuadro de diálogo de la contraseña se muestre en cada *URL*. Por ejemplo, cualquier usuario puede acceder a `/foo` sin que se le pida se autentique.

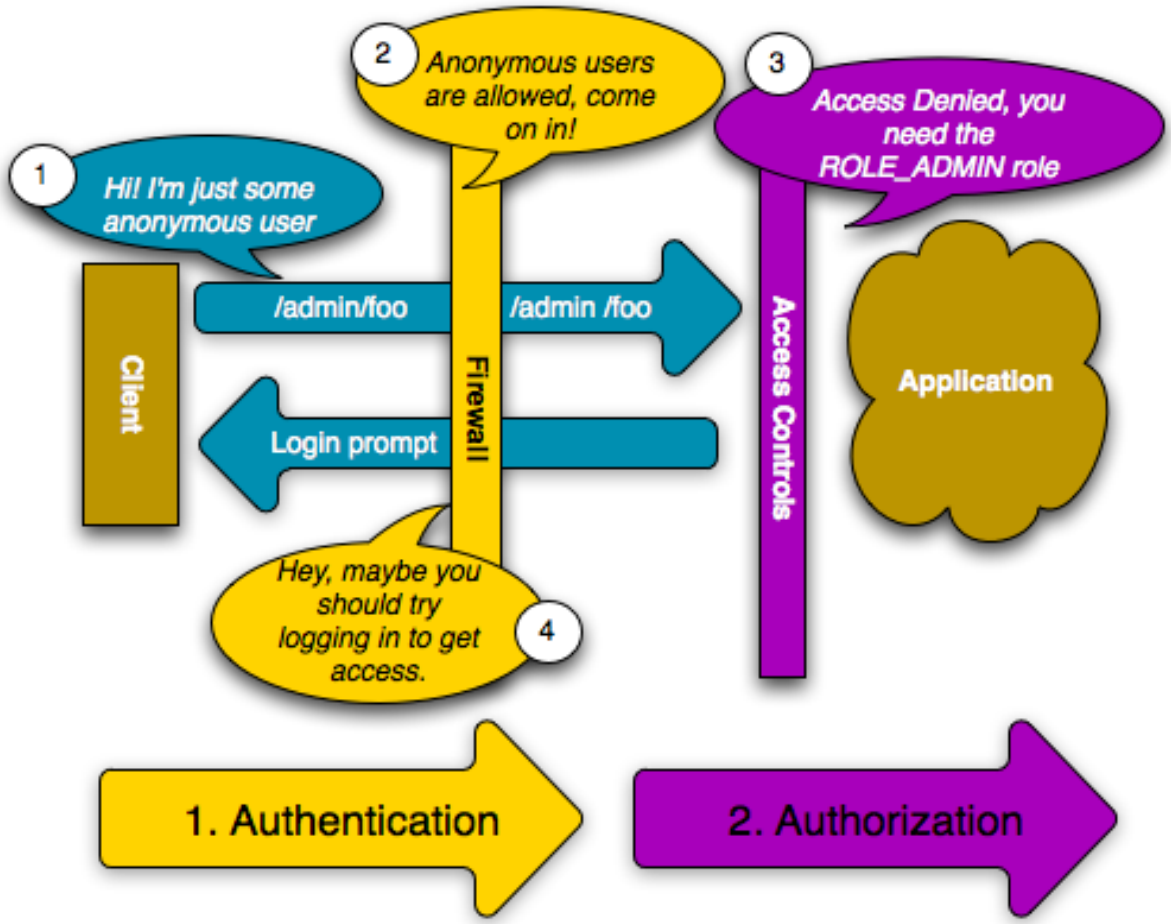


Esto funciona en primer lugar porque el cortafuegos permite *usuarios anónimos* a través del parámetro de configuración `anonymous`. En otras palabras, el cortafuegos no requiere que el usuario se autentique plenamente de inmediato. Y puesto que no hay `rol` especial necesario para acceder a `/foo` (bajo la sección `access_control`), la petición se puede llevar a cabo sin solicitar al usuario se autentique.

Si eliminas la clave `anonymous`, el cortafuegos *siempre* hará que un usuario se autentique inmediatamente.

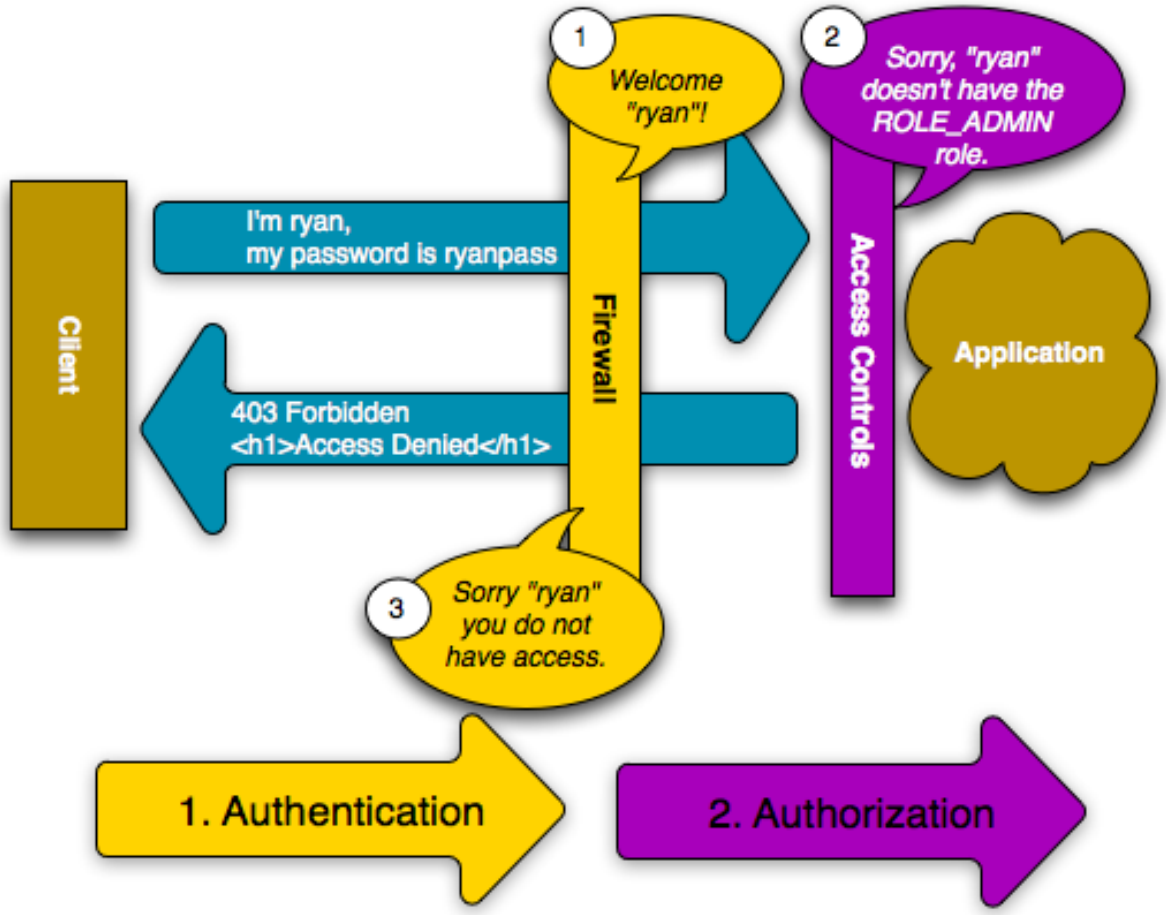
Control de acceso (autorización)

Si un usuario solicita `/admin/foo`, sin embargo, el proceso se comporta de manera diferente. Esto se debe a la sección de configuración `access_control` la cual dice que cualquier *URL* coincidente con el patrón de la expresión regular `^/admin` (es decir, `/admin` o cualquier cosa coincidente con `/admin/*`) requiere el rol `ROLE_ADMIN`. Los roles son la base para la mayor parte de la autorización: el usuario puede acceder a `/admin/foo` sólo si cuenta con el rol `ROLE_ADMIN`.



Como antes, cuando el usuario hace la petición originalmente, el cortafuegos no solicita ningún tipo de identificación. Sin embargo, tan pronto como la capa de control de acceso niega el acceso a los usuarios (ya que el usuario anónimo no tiene el rol `ROLE_ADMIN`), el servidor de seguridad entra en acción e inicia el proceso de autenticación). El proceso de autenticación depende del mecanismo de autenticación que utilice. Por ejemplo, si estás utilizando el método de autenticación con formulario de acceso, el usuario será redirigido a la página de inicio de sesión. Si estás utilizando autenticación *HTTP*, se enviará al usuario una respuesta *HTTP 401* para que el usuario vea el cuadro de diálogo de nombre de usuario y contraseña.

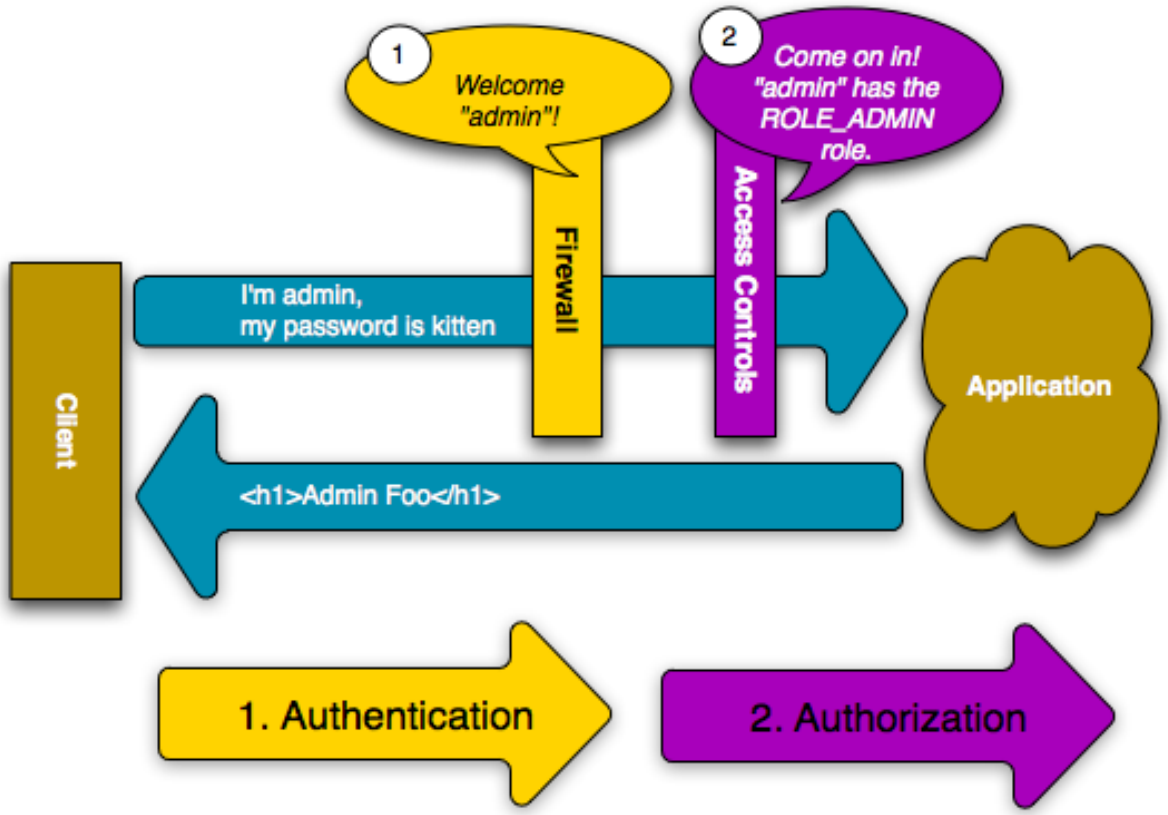
Ahora el usuario de nuevo tiene la posibilidad de presentar sus credenciales a la aplicación. Si las credenciales son válidas, se puede intentar de nuevo la petición original.



En este ejemplo, el usuario `ryan` se autentica correctamente con el cortafuegos. Pero como `ryan` no cuenta con el rol `ROLE_ADMIN`, se le sigue negando el acceso a `/admin/foo`. En última instancia, esto significa que el usuario debe ver algún tipo de mensaje indicándole que se le ha denegado el acceso.

Truco: Cuando *Symfony* niega el acceso al usuario, él verá una pantalla de error y recibe un código de estado *HTTP 403* (`Prohibido`). Puedes personalizar la pantalla de error, acceso denegado, siguiendo las instrucciones de las *Páginas de error* (Página 278) en la entrada del recetario para personalizar la página de error 403.

Por último, si el usuario `admin` solicita `/admin/foo`, se lleva a cabo un proceso similar, excepto que ahora, después de haberse autenticado, la capa de control de acceso le permitirá pasar a través de la petición:



El flujo de la petición cuando un usuario solicita un recurso protegido es sencillo, pero increíblemente flexible. Como verás más adelante, la autenticación se puede realizar de varias maneras, incluyendo a través de un formulario de acceso, certificados X.509 o la autenticación del usuario a través de *Twitter*. Independientemente del método de autenticación, el flujo de la petición siempre es el mismo:

1. Un usuario accede a un recurso protegido;
2. La aplicación redirige al usuario al formulario de acceso;
3. El usuario presenta sus credenciales (por ejemplo nombre de usuario/contraseña);
4. El cortafuegos autentica al usuario;
5. El nuevo usuario autenticado intenta de nuevo la petición original.

Nota: El proceso *exacto* realmente depende un poco en el mecanismo de autenticación utilizado. Por ejemplo, cuando utilizas el formulario de acceso, el usuario presenta sus credenciales a una *URL* que procesa el formulario (por ejemplo `/login_check`) y luego es redirigido a la dirección solicitada originalmente (por ejemplo `/admin/foo`). Pero con la autenticación *HTTP*, el usuario envía sus credenciales directamente a la *URL* original (por ejemplo `/admin/foo`) y luego la página se devuelve al usuario en la misma petición (es decir, sin redirección).

Este tipo de idiosincrasia no debería causar ningún problema, pero es bueno tenerla en cuenta.

Truco: También aprenderás más adelante cómo puedes proteger *cualquier cosa* en *Symfony2*, incluidos controladores específicos, objetos, e incluso métodos *PHP*.

2.23.3 Usando un formulario de acceso tradicional

Hasta ahora, hemos visto cómo cubrir tu aplicación bajo un cortafuegos y proteger el acceso a determinadas zonas con roles. Al usar la autenticación *HTTP*, sin esfuerzo puedes aprovechar el cuadro de diálogo nativo nombre de usuario/contraseña que ofrecen todos los navegadores. Sin embargo, fuera de la caja, *Symfony* permite múltiples mecanismos de autenticación. Para información detallada sobre todos ellos, consulta la *Referencia en configurando Security* (Página 452).

En esta sección, vamos a mejorar este proceso permitiendo la autenticación del usuario a través de un formulario de acceso *HTML* tradicional.

En primer lugar, activa el formulario de acceso en el cortafuegos:

- *YAML*

```
# app/config/config.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

- *XML*

```
<!-- app/config/config.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <form-login login_path="/login" check_path="/login_check" />
        </firewall>
    </config>
</srv:container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'form_login' => array(
                'login_path' => '/login',
                'check_path' => '/login_check',
            ),
        ),
    ),
));
```

Truco: Si no necesitas personalizar tus valores `login_path` o `check_path` (los valores utilizados aquí son los valores predeterminados), puedes acortar tu configuración:

- *YAML*

```
form_login: ~
```

- *XML*

```
<form-login />
```

- *PHP*

```
'form_login' => array(),
```

Ahora, cuando el sistema de seguridad inicia el proceso de autenticación, redirige al usuario al formulario de acceso (`/login` predeterminado). La implementación visual de este formulario de acceso es tu trabajo. En primer lugar, crea dos rutas: una que muestre el formulario de acceso (es decir, `/login`) y una que maneje el envío del formulario de acceso (es decir, `/login_check`):

- *YAML*

```
# app/config/routing.yml
login:
    pattern:   /login
    defaults:  { _controller: AcmeSecurityBundle:Security:login }
login_check:
    pattern:   /login_check
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="login" pattern="/login">
        <default key="_controller">AcmeSecurityBundle:Security:login</default>
    </route>
    <route id="login_check" pattern="/login_check" />

</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('login', new Route('/login', array(
    '_controller' => 'AcmeDemoBundle:Security:login',
)));
$coleccion->add('login_check', new Route('/login_check', array()));

return $coleccion;
```

Nota: *No* necesitas implementar un controlador para la *URL* `/login_check` ya que el cortafuegos automáticamente captura y procesa cualquier formulario enviado a esta *URL*. Es opcional, pero útil, crear una ruta para que puedas usarla al generar la *URL* de envío del formulario en la plantilla de entrada, a continuación.

Observa que el nombre de la ruta `login` no es importante. Lo importante es que la *URL* de la ruta (`/login`) coincida con el valor `login_path` configurado, ya que es donde el sistema de seguridad redirige a los usuarios que necesitan acceder.

A continuación, crea el controlador que mostrará el formulario de acceso:

```
// src/Acme/SecurityBundle/Controller/Main;
namespace Acme\SecurityBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        $peticion = $this->getRequest();
        $sesion = $peticion->getSession();

        // obtiene el error de inicio de sesión si lo hay
        if ($peticion->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $peticion->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $sesion->get(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('AcmeSecurityBundle:Security:login.html.twig', array(
            // el último nombre de usuario ingresado por el usuario
            'ultimo_nombreusuario' => $sesion->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

No dejes que este controlador te confunda. Como veremos en un momento, cuando el usuario envía el formulario, el sistema de seguridad se encarga de automatizar el envío del formulario por ti. Si el usuario ha presentado un nombre de usuario o contraseña no válidos, este controlador lee el error del formulario enviado desde el sistema de seguridad de modo que se pueda mostrar al usuario.

En otras palabras, su trabajo es mostrar el formulario al usuario y los errores de ingreso que puedan haber ocurrido, pero, el propio sistema de seguridad se encarga de verificar el nombre de usuario y contraseña y la autenticación del usuario.

Por último, crea la plantilla correspondiente:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="nombreusuario">Nombre:</label>
    <input type="text" id="nombreusuario" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />
```

```
        {#
            Si deseas controlar la URL a la que rediriges al usuario en caso de éxito (más detalles
            <input type="hidden" name="_target_path" value="/account" />
        #}

        <input type="submit" name="login" />
    </form>
```

- *PHP*

```
    <?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
    <?php if ($error): ?>
        <div><?php echo $error->getMessage() ?></div>
    <?php endif; ?>

    <form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
        <label for="nombreusuario">Nombre:</label>
        <input type="text" id="nombreusuario" name="_username" value="<?php echo $last_username ?>"

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        <!--
            Si deseas controlar la URL a la que rediriges al usuario en caso de éxito (más detalles
            <input type="hidden" name="_target_path" value="/account" />
        -->

        <input type="submit" name="login" />
    </form>
```

Truco: La variable `error` pasada a la plantilla es una instancia de `Symfony\Component\Security\Core\Exception\AuthenticationException`. Esta puede contener más información - o incluso información confidencial - sobre el fallo de autenticación, ¡por lo tanto utilízalo prudentemente!

El formulario tiene muy pocos requisitos. En primer lugar, presentando el formulario a `/login_check` (a través de la ruta `login_check`), el sistema de seguridad debe interceptar el envío del formulario y procesarlo automáticamente. En segundo lugar, el sistema de seguridad espera que los campos presentados se llamen `_username` y `_password` (estos nombres de campo se pueden *configurar* (Página 454)).

¡Y eso es todo! Cuando envías el formulario, el sistema de seguridad automáticamente comprobará las credenciales del usuario y, o bien autenticará al usuario o enviará al usuario al formulario de acceso donde se puede mostrar el error.

Vamos a revisar todo el proceso:

1. El usuario intenta acceder a un recurso que está protegido;
2. El cortafuegos inicia el proceso de autenticación redirigiendo al usuario al formulario de acceso (`/login`);
3. La página `/login` reproduce el formulario de acceso a través de la ruta y el controlador creado en este ejemplo;
4. El usuario envía el formulario de acceso a `/login_check`;
5. El sistema de seguridad intercepta la petición, comprueba las credenciales presentadas por el usuario, autentica al usuario si todo está correcto, y si no, envía al usuario de nuevo al formulario de acceso.

Por omisión, si las credenciales presentadas son correctas, el usuario será redirigido a la página solicitada originalmente (por ejemplo `/admin/foo`). Si originalmente el usuario fue directo a la página de inicio de sesión, será redirigido a

la página principal. Esto puede ser altamente personalizado, lo cual te permite, por ejemplo, redirigir al usuario a una *URL* específica.

Para más detalles sobre esto y cómo personalizar el proceso de entrada en general, consulta *Cómo personalizar el formulario de acceso* (Página 383).

Evitando errores comunes

Cuando prepares tu formulario de acceso, ten cuidado con unas cuantas trampas muy comunes.

1. Crea las rutas correctas

En primer lugar, asegúrate de que has definido las rutas `/login` y `/login_check` correctamente y que correspondan a los valores de configuración `login_path` y `check_path`. Una mala configuración aquí puede significar que serás redirigido a una página de error 404 en lugar de la página de acceso, o que al presentar el formulario de acceso no haga nada (sólo verás el formulario de acceso una y otra vez).

2. Asegúrate de que la página de inicio de sesión no es segura

Además, asegúrate de que la página de entrada *no* requiere ningún rol para verla. Por ejemplo, la siguiente configuración - la cual requiere el rol `ROLE_ADMIN` para todas las *URL* (incluyendo la *URL* `/login`), provocará un bucle de redirección:

- *YAML*

```
access_control:
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Quitar el control de acceso en la *URL* `/login` soluciona el problema:

- *YAML*

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/login" role="IS_AUTHENTICATED_ANONYMOUSLY" />
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY'),
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Además, si el cortafuegos *no* permite usuarios anónimos, necesitas crear un cortafuegos especial que permita usuarios anónimos en la página de acceso:

- *YAML*

```
firewalls:
    login_firewall:
        pattern:    ^/login$
        anonymous:  ~
    secured_area:
        pattern:    ^/
        form_login: ~
```

- *XML*

```
<firewall name="login_firewall" pattern="^/login$">
    <anonymous />
</firewall>
<firewall name="secured_area" pattern="^/">
    <form_login />
</firewall>
```

- *PHP*

2.23.4 Autorizando

El primer paso en la seguridad siempre es la autenticación: el proceso de verificar quién es el usuario. Con *Symfony*, la autenticación se puede hacer de cualquier manera - a través de un formulario de acceso, autenticación básica *HTTP*, e incluso a través de *Facebook*.

Una vez que el usuario se ha autenticado, comienza la autorización. La autorización proporciona una forma estándar y potente para decidir si un usuario puede acceder a algún recurso (una *URL*, un modelo de objetos, una llamada a un método, ...). Esto funciona asignando roles específicos a cada usuario y, a continuación, requiriendo diferentes roles para diferentes recursos.

El proceso de autorización tiene dos lados diferentes:

1. El usuario tiene un conjunto de roles específico;
2. Un recurso requiere un rol específico a fin de tener acceso.

En esta sección, nos centraremos en cómo proteger diferentes recursos (por ejemplo, direcciones *URL*, llamadas a métodos, etc.) con diferentes roles. Más tarde, aprenderás más sobre cómo crear y asignar roles a los usuarios.

Protegiendo patrones de *URL* específicas

La forma más básica para proteger parte de tu aplicación es asegurar un patrón de direcciones *URL* completo. Ya has visto en el primer ejemplo de este capítulo, donde algo que coincide con el patrón de la expresión regular `^/admin` requiere el rol `ROLE_ADMIN`.

Puedes definir tantos patrones *URL* como necesites - cada uno es una expresión regular.

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <access-control>
        <rule path="^/admin/users" role="ROLE_SUPER_ADMIN" />
        <rule path="^/admin" role="ROLE_ADMIN" />
    </access-control>
</config>
```

- *PHP*

```php
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    // ...
    'access_control' => array(
        array('path' => '^/admin/users', 'role' => 'ROLE_SUPER_ADMIN'),
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
));
```

Truco: Al prefijar la ruta con ^ te aseguras que sólo coinciden las *URL* que *comienzan* con ese patrón. Por ejemplo, una ruta de simplemente `/admin` (sin el ^) correctamente coincidirá con `/admin/foo` pero también coincide con *URLs* como `/foo/admin`.

Para cada petición entrante, *Symfony2* trata de encontrar una regla de control de acceso coincidente (la primera gana). Si el usuario no está autenticado, sin embargo, se inicia el proceso de autenticación (es decir, se le da al usuario una oportunidad de ingresar). Sin embargo, si el usuario *está* autenticado, pero no tiene el rol necesario, se lanza una excepción `Symfony\ComponentSecurity\Core\Exception\AccessDeniedException`, que puedes manejar y convertir en una bonita página de error "acceso denegado" para el usuario. Consulta *Cómo personalizar páginas de error* (Página 277) para más información.

Debido a que *Symfony* utiliza la primera regla de control de acceso coincidente, una *URL* como `/admin/usuarios/nuevo` coincidirá con la primera regla y sólo requiere el rol `ROLE_SUPER_ADMIN`. Cualquier *URL* como `/admin/blog` coincidirá con la segunda regla y requiere un `ROLE_ADMIN`.

También puedes forzar *HTTP* o *HTTPS* a través de una entrada `access_control`. Para más información, consulta *Cómo forzar HTTPS o HTTP a diferentes URL* (Página 382).

Protegiendo un controlador

Proteger tu aplicación basándote en los patrones *URL* es fácil, pero, en algunos casos, puede no estar suficientemente bien ajustado. Cuando sea necesario, fácilmente puedes forzar la autorización al interior de un controlador:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException
// ...

public function holaAction($nombre)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

También puedes optar por instalar y utilizar el opcional `JMSSecurityExtraBundle`, el cual puede asegurar tu controlador usando anotaciones:

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function holaAction($nombre)
{
    // ...
}
```

Para más información, consulta la documentación de JMSSecurityExtraBundle. Si estás usando la distribución estándar de *Symfony*, este paquete está disponible de forma predeterminada. Si no es así, lo puedes descargar e instalar.

Protegiendo otros servicios

De hecho, en *Symfony* puedes proteger cualquier cosa utilizando una estrategia similar a la observada en la sección anterior. Por ejemplo, supongamos que tienes un servicio (es decir, una clase *PHP*), cuyo trabajo consiste en enviar

mensajes de correo electrónico de un usuario a otro. Puedes restringir el uso de esta clase - no importa dónde se esté utilizando - a los usuarios que tienen un rol específico.

Para más información sobre cómo utilizar el componente de seguridad para proteger diferentes servicios y métodos en tu aplicación, consulta *Cómo proteger cualquier servicio o método de tu aplicación* (Página 389).

Listas de control de acceso (ACL): Protegiendo objetos individuales de base de datos

Imagina que estás diseñando un sistema de blog donde los usuarios pueden comentar tus mensajes. Ahora, deseas que un usuario pueda editar sus propios comentarios, pero no los de otros usuarios. Además, como usuario `admin`, quieres tener la posibilidad de editar *todos* los comentarios.

El componente de seguridad viene con un sistema opcional de lista de control de acceso (ACL) que puedes utilizar cuando sea necesario para controlar el acceso a instancias individuales de un objeto en el sistema. *Sin* ACL, puedes proteger tu sistema para que sólo determinados usuarios puedan editar los comentarios del blog en general. Pero *con* ACL, puedes restringir o permitir el acceso en base a comentario por comentario.

Para más información, consulta el artículo del recetario: `/book/security/acl`.

2.23.5 Usuarios

En las secciones anteriores, aprendiste cómo puedes proteger diferentes recursos que requieren un conjunto de *roles* para un recurso. En esta sección vamos a explorar el otro lado de la autorización: los usuarios.

¿De dónde provienen los usuarios? (*Proveedores de usuarios*)

Durante la autenticación, el usuario envía un conjunto de credenciales (por lo general un nombre de usuario y contraseña). El trabajo del sistema de autenticación es concordar esas credenciales contra una piscina de usuarios. Entonces, ¿de dónde viene esta lista de usuarios?

En *Symfony2*, los usuarios pueden venir de cualquier parte - un archivo de configuración, una tabla de base de datos, un servicio web, o cualquier otra cosa que se te ocurra. Todo lo que proporcione uno o más usuarios al sistema de autenticación se conoce como "proveedor de usuario". *Symfony2* de serie viene con los dos proveedores de usuario más comunes: uno que carga los usuarios de un archivo de configuración y otro que carga usuarios de una tabla de base de datos.

Especificando usuarios en un archivo de configuración

La forma más fácil para especificar usuarios es directamente en un archivo de configuración. De hecho, ya lo has visto en algunos ejemplos de este capítulo.

- *YAML*

```
# app/config/config.yml
security:
    # ...
    providers:
        default_provider:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="default_provider">
        <user name="ryan" password="ryanpass" roles="ROLE_USER" />
        <user name="admin" password="kitten" roles="ROLE_ADMIN" />
    </provider>
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'default_provider' => array(
            'users' => array(
                'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
            ),
        ),
    ),
));
```

Este proveedor de usuario se denomina proveedor de usuario "en memoria", ya que los usuarios no se almacenan en alguna parte de una base de datos. El objeto usuario en realidad lo proporciona *Symfony* (`Symfony\Component\Security\Core\User\User`).

Truco: Cualquier proveedor de usuario puede cargar usuarios directamente desde la configuración especificando el parámetro de configuración `users` y la lista de usuarios debajo de él.

Prudencia: Si tu nombre de usuario es completamente numérico (por ejemplo, `77`) o contiene un guión (por ejemplo, `nombre-usuario`), debes utilizar la sintaxis alterna al especificar usuarios en YAML:

```
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

Para sitios pequeños, este método es rápido y fácil de configurar. Para sistemas más complejos, querrás cargar usuarios desde la base de datos.

Cargando usuarios de la base de datos

Si deseas cargar tus usuarios a través del *ORM* de *Doctrine*, lo puedes hacer creando una clase `Usuario` y configurando el proveedor `entity`.

Con este enfoque, primero crea tu propia clase `User`, la cual se almacenará en la base de datos.

```
// src/Acme/UserBundle/Entity/Usuario.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;

/**
```

```
 * @ORM\Entity
 */
class Usuario implements UserInterface
{
    /**
     * @ORM\Column(type="string", length="255")
     */
    protected $nombreusuario;

    // ...
}
```

En cuanto al sistema de seguridad se refiere, el único requisito para tu clase Usuario personalizada es que implemente la interfaz `Symfony\Component\Security\Core\User\UserInterface`. Esto significa que el concepto de un "usuario" puede ser cualquier cosa, siempre y cuando implemente esta interfaz.

Nota: El objeto de usuario se debe serializar y guardar en la sesión durante las peticiones, por lo tanto se recomienda que implementes la interfaz Serializable en tu objeto usuario. Esto es especialmente importante si tu clase `Usuario` tiene una clase padre con propiedades privadas.

A continuación, configura una `entidad` proveedora de usuario, y apuntala a tu clase `Usuario`:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: nombreusuario }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <provider name="main">
        <entity class="Acme\UserBundle\Entity\User" property="nombreusuario" />
    </provider>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'providers' => array(
        'main' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'nombreusu
        ),
    ),
));
```

Con la introducción de este nuevo proveedor, el sistema de autenticación intenta cargar un objeto `Usuario` de la base de datos utilizando el campo `nombreusuario` de esa clase.

Para más información sobre cómo crear tu propio proveedor personalizado (por ejemplo, si es necesario cargar los usuarios a través de un servicio Web), consulta *Cómo crear un proveedor de usuario personalizado* (Página 393).

Codificando la contraseña del usuario

Hasta ahora, por simplicidad, todos los ejemplos tienen las contraseñas de los usuarios almacenadas en texto plano (si los usuarios se almacenan en un archivo de configuración o en alguna base de datos). Por supuesto, en una aplicación real, por razones de seguridad, desearás codificar las contraseñas de los usuarios. Esto se logra fácilmente asignando la clase Usuario a una de las varias integradas en `encoders`. Por ejemplo, para almacenar los usuarios en memoria, pero ocultar sus contraseñas a través de `sha1`, haz lo siguiente:

- *YAML*

```
# app/config/config.yml
security:
    # ...
    providers:
        in_memory:
            users:
                ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles: 'ROLE_USER'
                admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles: 'ROLE_ADMIN'

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm:   sha1
            iterations: 1
            encode_as_base64: false
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="in_memory">
        <user name="ryan" password="bb87a29949f3a1ee0559f8a57357487151281386" roles="ROLE_USER"
        <user name="admin" password="74913f5cd5f61ec0bcfdb775414c2fb3d161b620" roles="ROLE_ADMIN
    </provider>

    <encoder class="Symfony\Component\Security\Core\User\User" algorithm="sha1" iterations="1" e
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'in_memory' => array(
            'users' => array(
                'ryan' => array('password' => 'bb87a29949f3a1ee0559f8a57357487151281386', 'roles
                'admin' => array('password' => '74913f5cd5f61ec0bcfdb775414c2fb3d161b620', 'role
            ),
        ),
    ),
    'encoders' => array(
        'Symfony\Component\Security\Core\User\User' => array(
            'algorithm'         => 'sha1',
            'iterations'        => 1,
            'encode_as_base64'  => false,
        ),
    ),
));
```

Al establecer las `iterations` a `1` y `encode_as_base64` en `false`, la contraseña simplemente se corre una vez a través del algoritmo `sha1` y sin ninguna codificación adicional. Ahora puedes calcular el `hash` de la contraseña mediante programación (por ejemplo, `hash('sha1', 'ryanpass')`) o a través de alguna herramienta en línea como functions-online.com

Si vas a crear dinámicamente a tus usuarios (y almacenarlos en una base de datos), puedes utilizar algoritmos `hash` aún más difíciles y, luego confiar en un objeto codificador de clave real para ayudarte a codificar las contraseñas. Por ejemplo, supongamos que tu objeto usuario es `Acme\UserBundle\Entity\User` (como en el ejemplo anterior). Primero, configura el codificador para ese usuario:

- *YAML*

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

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <!-- ... -->

    <encoder class="Acme\UserBundle\Entity\User" algorithm="sha512" />
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    // ...

    'encoders' => array(
        'Acme\UserBundle\Entity\User' => 'sha512',
    ),
));
```

En este caso, estás utilizando el fuerte algoritmo `SHA512`. Además, puesto que hemos especificado simplemente el algoritmo (`sha512`) como una cadena, el sistema de manera predeterminada revuelve tu contraseña 5000 veces en una fila y luego la codifica como base64. En otras palabras, la contraseña ha sido fuertemente ofuscada por lo tanto la contraseña revuelta no se puede decodificar (es decir, no se puede determinar la contraseña desde la contraseña ofuscada).

Si tienes algún formulario de registro para los usuarios, tendrás que poder determinar la contraseña con el algoritmo `hash`, para que puedas ponerla en tu usuario. No importa qué algoritmo configures para el objeto usuario, la contraseña con algoritmo `hash` siempre la puedes determinar de la siguiente manera desde un controlador:

```
$factory = $this->get('security.encoder_factory');
$user = new Acme\UserBundle\Entity\User();

$encoder = $factory->getEncoder($user);
$pase = $encoder->encodePassword('ryanpass', $user->getSalt());
$user->setPassword($pase);
```

Recuperando el objeto usuario

Después de la autenticación, el objeto `Usuario` del usuario actual se puede acceder a través del servicio `security.context`. Desde el interior de un controlador, este se verá así:

```
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

Nota: Los usuarios anónimos técnicamente están autenticados, lo cual significa que el método `isAuthenticated()` de un objeto usuario anónimo devolverá `true`. Para comprobar si el usuario está autenticado realmente, verifica el rol `IS_AUTHENTICATED_FULLY`.

Usando múltiples proveedores de usuario

Cada mecanismo de autenticación (por ejemplo, la autenticación *HTTP*, formulario de acceso, etc.) utiliza exactamente un proveedor de usuario, y de forma predeterminada utilizará el primer proveedor de usuario declarado. Pero, si deseas especificar unos cuantos usuarios a través de la configuración y el resto de los usuarios en la base de datos? Esto es posible creando un nuevo proveedor que encadene los dos:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        chain_provider:
            providers: [in_memory, user_db]
        in_memory:
            users:
                foo: { password: test }
        user_db:
            entity: { class: Acme\UserBundle\Entity\User, property: nombreusuario }
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <provider name="chain_provider">
        <provider>in_memory</provider>
        <provider>user_db</provider>
    </provider>
    <provider name="in_memory">
        <user name="foo" password="test" />
    </provider>
    <provider name="user_db">
        <entity class="Acme\UserBundle\Entity\User" property="nombreusuario" />
    </provider>
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    'providers' => array(
        'chain_provider' => array(
```

```
            'providers' => array('in_memory', 'user_db'),
        ),
        'in_memory' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
        ),
        'user_db' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'nombreusu
        ),
    ),
));
```

Ahora, todos los mecanismos de autenticación utilizan el `chain_provider`, puesto que es el primero especificado. El `chain_provider`, a su vez, intenta cargar el usuario, tanto el proveedor `in_memory` cómo `USER_DB`.

Truco: Si no tienes razones para separar a tus usuarios `in_memory` de tus usuarios `user_db`, lo puedes hacer aún más fácil combinando las dos fuentes en un único proveedor:

- *YAML*

```yaml
# app/config/security.yml
security:
    providers:
        main_provider:
            users:
                foo: { password: test }
            entity: { class: Acme\UserBundle\Entity\User, property: nombreusuario }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <provider name=="main_provider">
        <user name="foo" password="test" />
        <entity class="Acme\UserBundle\Entity\User" property="nombreusuario" />
    </provider>
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    'providers' => array(
        'main_provider' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'nombreusu
        ),
    ),
));
```

También puedes configurar el cortafuegos o mecanismos de autenticación individuales para utilizar un proveedor específico. Una vez más, a menos que especifiques un proveedor explícitamente, siempre se utiliza el primer proveedor:

- *YAML*

```yaml
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Demo de área protegida"
                provider: in_memory
            form_login: ~
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/" provider="user_db">
        <!-- ... -->
        <http-basic realm="Demo de área protegida" provider="in_memory" />
        <form-login />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'provider' => 'user_db',
            'http_basic' => array(
                // ...
                'provider' => 'in_memory',
            ),
            'form_login' => array(),
        ),
    ),
));
```

En este ejemplo, si un usuario intenta acceder a través de autenticación *HTTP*, el sistema de autenticación debe utilizar el proveedor de usuario `in_memory`. Pero si el usuario intenta acceder a través del formulario de acceso, se utilizará el proveedor `USER_DB` (ya que es el valor predeterminado para el servidor de seguridad en su conjunto).

Para más información acerca de los proveedores de usuario y la configuración del cortafuegos, consulta la *Referencia en configurando Security* (Página 452).

2.23.6 Roles

La idea de un "rol" es clave para el proceso de autorización. Cada usuario tiene asignado un conjunto de roles y cada recurso requiere uno o más roles. Si el usuario tiene los roles necesarios, se le concede acceso. En caso contrario se deniega el acceso.

Los roles son bastante simples, y básicamente son cadenas que puedes inventar y utilizar cuando sea necesario (aunque los roles son objetos internos). Por ejemplo, si necesitas comenzar a limitar el acceso a la sección admin del blog de tu sitio web, puedes proteger esa sección con un rol llamado `ROLE_BLOG_ADMIN`. Este rol no necesita estar definido en ningún lugar - puedes comenzar a usarlo.

Nota: Todos los roles **deben** comenzar con el prefijo `ROLE_` el cual será gestionado por *Symfony2*. Si defines tu propios roles con una clase `Rol` dedicada (más avanzada), no utilices el prefijo `ROLE_`.

Roles jerárquicos

En lugar de asociar muchos roles a los usuarios, puedes definir reglas de herencia creando una jerarquía de roles:

- *YAML*

```yaml
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <role-hierarchy>
        <role id="ROLE_ADMIN">ROLE_USER</role>
        <role id="ROLE_SUPER_ADMIN">ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH</role>
    </role-hierarchy>
</config>
```

- *PHP*

```php
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'role_hierarchy' => array(
        'ROLE_ADMIN'       => 'ROLE_USER',
        'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_ALLOWED_TO_SWITCH'),
    ),
));
```

En la configuración anterior, los usuarios con rol `ROLE_ADMIN` también tendrán el rol de `ROLE_USER`. El rol `ROLE_SUPER_ADMIN` tiene `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` y `ROLE_USER` (heredado de `ROLE_ADMIN`).

2.23.7 Cerrando sesión

Por lo general, también quieres que tus usuarios puedan salir. Afortunadamente, el cortafuegos puede manejar esto automáticamente cuando activas el parámetro de configuración `logout`:

- *YAML*

```yaml
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/">
        <!-- ... -->
        <logout path="/logout" target="/" />
    </firewall>
    <!-- ... -->
</config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'logout' => array('path' => 'logout', 'target' => '/'),
        ),
    ),
    // ...
));
```

Una vez que este está configurado en tu cortafuegos, enviar a un usuario a `/logout` (o cualquiera que sea tu `path` configurada), debes desautenticar al usuario actual. El usuario será enviado a la página de inicio (el valor definido por el parámetro `target`). Ambos parámetros `path` y `target` por omisión se configuran a lo que esté especificado aquí. En otras palabras, a menos que necesites personalizarlos, los puedes omitir por completo y acortar tu configuración:

- *YAML*

```
logout: ~
```

- *XML*

```
<logout />
```

- *PHP*

```
'logout' => array(),
```

Ten en cuenta que *no* es necesario implementar un controlador para la *URL* `/logout` porque el cortafuegos se encarga de todo. Puedes, sin embargo, desear crear una ruta para que puedas utilizarla para generar la *URL*:

- *YAML*

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

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="logout" pattern="/logout" />

</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('logout', new Route('/logout', array()));

return $coleccion;
```

Una vez que el usuario ha cerrado la sesión, será redirigido a cualquier ruta definida por el parámetro `target` anterior (por ejemplo, la página `principal`). Para más información sobre cómo configurar el cierre de sesión, consulta *Referencia en configurando Security* (Página 452).

2.23.8 Controlando el acceso en plantillas

Si deseas comprobar si el usuario actual tiene un rol dentro de una plantilla, utiliza la función ayudante incorporada:

- *Twig*

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

- *PHP*

```
<?php if ($view['security']->isGranted('ROLE_ADMIN')): ?>
    <a href="...">Delete</a>
<?php endif; ?>
```

Nota: Si utilizas esta función y *no* estás en una *URL* donde haya un cortafuegos activo, se lanzará una excepción. Una vez más, casi siempre es buena idea tener un cortafuegos principal que cubra todas las *URL* (como hemos mostrado en este capítulo).

2.23.9 Controlando el acceso en controladores

Si deseas comprobar en tu controlador si el usuario actual tiene un rol, utiliza el método `isGranted` del contexto de seguridad:

```
public function indexAction()
{
    // muestra diferente contenido para los usuarios admin
    if($this->get('security.context')->isGranted('ADMIN')) {
        // carga el contenido admin aquí
    }
    // carga el contenido regular aquí
}
```

Nota: Un cortafuegos debe estar activo o cuando se llame al método `isGranted` se producirá una excepción. Ve la nota anterior acerca de las plantillas para más detalles.

2.23.10 Suplantando a un usuario

A veces, es útil poder cambiar de un usuario a otro sin tener que iniciar sesión de nuevo (por ejemplo, cuando depuras o tratas de entender un error que un usuario ve y que no se puede reproducir). Esto se puede hacer fácilmente activando el escucha `switch_user` del cortafuegos:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => true
        ),
    ),
));
```

Para cambiar a otro usuario, sólo tienes que añadir una cadena de consulta con el parámetro `_switch_user` y el nombre de usuario como el valor de la dirección actual:

```
http://ejemplo.com/somewhere?_switch_user=thomas
```

Para volver al usuario original, utiliza el nombre de usuario especial `_exit`:

```
http://ejemplo.com/somewhere?_switch_user=_exit
```

Por supuesto, esta función se debe poner a disposición de un pequeño grupo de usuarios. De forma predeterminada, el acceso está restringido a usuarios que tienen el rol `ROLE_ALLOWED_TO_SWITCH`. El nombre de esta función se puede modificar a través de la configuración `role`. Para mayor seguridad, también puedes cambiar el nombre del parámetro de consulta a través de la configuración `parameter`:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user role="ROLE_ADMIN" parameter="_want_to_be_this_user" />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => array('role' => 'ROLE_ADMIN', 'parameter' => '_want_to_be_this_user
        ),
    ),
));
```

2.23.11 Autenticación apátrida

De forma predeterminada, *Symfony2* confía en una `cookie` (la Sesión) para persistir el contexto de seguridad del usuario. Pero si utilizas certificados o autenticación *HTTP*, por ejemplo, la persistencia no es necesaria ya que están disponibles las credenciales para cada petición. En ese caso, y si no es necesario almacenar cualquier otra cosa entre peticiones, puedes activar la autenticación apátrida (lo cual significa que *Symfony2* jamás creará una `cookie`):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall stateless="true">
        <http-basic />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('http_basic' => array(), 'stateless' => true),
    ),
));
```

Nota: Si utilizas un formulario de acceso, *Symfony2* creará una `cookie`, incluso si estableces `stateless` a `true`.

2.23.12 Palabras finales

La seguridad puede ser un tema profundo y complejo de resolver correctamente en tu aplicación. Afortunadamente, el componente de seguridad de *Symfony* sigue un modelo de seguridad bien probado en torno a la *autenticación* y *autorización*. Autenticación, siempre sucede en primer lugar, está a cargo de un cortafuegos, cuyo trabajo es determinar la identidad del usuario a través de varios métodos diferentes (por ejemplo, la autenticación *HTTP*, formulario de acceso, etc.) En el recetario, encontrarás ejemplos de otros métodos para manejar la autenticación, incluyendo la manera de implementar una funcionalidad "recuérdame" por medio de `cookie`.

Una vez que un usuario se autentica, la capa de autorización puede determinar si el usuario debe tener acceso a un recurso específico. Por lo general, los *roles* se aplican a las direcciones *URL*, clases o métodos y si el usuario actual no tiene ese rol, se le niega el acceso. La capa de autorización, sin embargo, es mucho más profunda, y sigue un sistema de "voto" para que varias partes puedan determinar si el usuario actual debe tener acceso a un determinado recurso. Para saber más sobre este y otros temas busca en el recetario.

2.23.13 Aprende más en el recetario

- *Forzando HTTP/HTTPS* (Página 382)
- *Lista negra de usuarios por dirección IP con votante personalizado* (Página 373)
- *Listas de control de acceso (ACLs)* (Página 376)
- *Cómo agregar la funcionalidad "recuérdame" al inicio de sesión* (Página 370)

2.24 Caché *HTTP*

La naturaleza de las aplicaciones web ricas significa que son dinámicas. No importa qué tan eficiente sea tu aplicación, cada petición siempre contendrá más sobrecarga que servir un archivo estático.

Y para la mayoría de las aplicaciones Web, está bien. *Symfony2* es tan rápido como el rayo, a menos que estés haciendo una muy complicada aplicación, cada petición se responderá rápidamente sin poner demasiada tensión a tu servidor.

Pero cuando tu sitio crezca, la sobrecarga general se puede convertir en un problema. El procesamiento que se realiza normalmente en cada petición se debe hacer sólo una vez. Este exactamente es el objetivo que tiene que consumar la memoria caché.

2.24.1 La memoria caché en hombros de gigantes

La manera más efectiva para mejorar el rendimiento de una aplicación es memorizar en caché la salida completa de una página y luego eludir por completo la aplicación en cada petición posterior. Por supuesto, esto no siempre es posible para los sitios web altamente dinámicos, ¿o no? En este capítulo, te mostraremos cómo funciona el sistema de caché *Symfony2* y por qué pensamos que este es el mejor enfoque posible.

El sistema de cache *Symfony2* es diferente porque se basa en la simplicidad y el poder de la caché *HTTP* tal como está definido en la *especificación HTTP*. En lugar de reinventar una metodología de memoria caché, *Symfony2* adopta la norma que define la comunicación básica en la Web. Una vez que comprendas los principios fundamentales de los modelos de caducidad y validación de la memoria caché *HTTP*, estarás listo para dominar el sistema de caché *Symfony2*.

Para efectos de aprender cómo guardar en caché con *Symfony2*, vamos a cubrir el tema en cuatro pasos:

- **Paso 1**: Una *pasarela de caché* (Página 211), o delegado inverso, es una capa independiente situada frente a tu aplicación. La caché del delegado inverso responde a medida que son devueltas desde tu aplicación y responde

a peticiones con respuestas de la caché antes de que lleguen a tu aplicación. *Symfony2* proporciona su propio delegado inverso, pero puedes utilizar cualquier delegado inverso.

- **Paso 2** *cache HTTP* (Página 214) las cabeceras se utilizan para comunicarse con la pasarela de caché y cualquier otro caché entre la aplicación y el cliente. *Symfony2* proporciona parámetros predeterminados y una potente interfaz para interactuar con las cabeceras de caché.
- **Paso 3**: HTTP *caducidad y validación* (Página 215) son los dos modelos utilizados para determinar si el contenido memorizado en caché es *fresco* (se puede reutilizar de la memoria caché) u *obsoleto* (lo debe regenerar la aplicación).
- **Paso 4** : *Inclusión del borde lateral* (Página 220) (Edge Side Includes -ESI (Edge Side Includes o Inclusión del borde lateral)) permite que la caché *HTTP* utilice fragmentos de la página en caché (incluso fragmentos anidados) independientemente. Con ESI, incluso puedes guardar en caché una página entera durante 60 minutos, pero una barra lateral integrada sólo por 5 minutos.

Dado que la memoria caché *HTTP* no es exclusiva de *Symfony*, ya existen muchos artículos sobre el tema. Si eres nuevo para la memoria caché *HTTP*, te *recomendamos* el artículo de Ryan Tomayko Things Caches Do. Otro recurso en profundidad es la Guía de caché de Mark Nottingham.

2.24.2 Memoria caché con pasarela de caché

Cuándo memorizar caché con HTTP, la *caché* está separada de tu aplicación por completo y se sitúa entre tu aplicación y el cliente haciendo la petición.

El trabajo de la caché es aceptar las peticiones del cliente y pasarlas de nuevo a tu aplicación. La memoria caché también recibirá las respuestas devueltas por tu aplicación y las remitirá al cliente. La caché es el "geniecillo" de la comunicación petición-respuesta entre el cliente y tu aplicación.

En el camino, la memoria caché almacena cada respuesta que se considere "cacheable" (consulta *Introducción a la memoria caché HTTP* (Página 214)). Si de nuevo se solicita el mismo recurso, la memoria caché envía la respuesta memorizada en caché al cliente, eludiendo tu aplicación por completo.

Este tipo de caché se conoce como caché de puerta de enlace HTTP y existen muchos como Varnish, Squid en modo delegado inverso y el delegado inverso de *Symfony2*.

Tipos de Caché

Sin embargo, una puerta de enlace caché no es el único tipo de caché. De hecho, las cabeceras de caché *HTTP* enviadas por tu aplicación son consumidas e interpretadas por un máximo de tres diferentes tipos de caché:

- *Caché de navegadores*: Cada navegador viene con su propia caché local que es realmente útil para cuando pulsas "atrás" o en imágenes y otros activos. La caché del navegador es una caché *privada*, los recursos memorizados en caché no se comparten con nadie más.
- *Cachés sustitutas*: Una sustituta es una caché *compartida* en que muchas personas pueden estar detrás de una sola. Por lo general instalada por las grandes corporaciones y proveedores de Internet para reducir latencia y tráfico de red.
- *Pasarelas de caché*: Al igual que un sustituto, también es una caché *compartida* pero de lado del servidor. Instalada por los administradores de red, esta tiene sitios web más escalables, confiables y prácticos.

Truco: Las pasarelas de caché a veces se conocen como caché sustituta inversa, cachés alquiladas o incluso aceleradores *HTTP*.

Nota: La importancia de la caché *privada* frente a la *compartida* será más evidente a medida que hablemos de las respuestas en la memoria caché con contenido que es específico para un solo usuario (por ejemplo, información de cuenta).

Cada respuesta de tu aplicación probablemente vaya a través de uno o los dos primeros tipos de caché. Estas cachés están fuera de tu control, pero siguen las instrucciones de caché *HTTP* establecidas en la respuesta.

Delegado inverso de *Symfony2*

Symfony2 viene con una caché sustituta inversa (también llamada memoria caché de puerta de enlace) escrita en *PHP*. Que al activarlo, inmediatamente puede memorizar en caché respuestas de tu aplicación. La instalación es muy fácil. Cada nueva aplicación *Symfony2* viene con un núcleo preconfigurado memorizado en caché (`AppCache`) que envuelve el (`AppKernel`) predeterminado. La memoria caché del núcleo *es* el delegado inverso.

Para habilitar la memoria caché, modifica el código de un controlador frontal para utilizar la memoria caché del núcleo:

```
// 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\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// envuelve el AppKernel predeterminado con un AppCache
$kernel = new AppCache($kernel);
$kernel->handle(Request::createFromGlobals())->send();
```

La memoria caché del núcleo actúa de inmediato como un delegado inverso - memorizando en caché las respuestas de tu aplicación y devolviéndolas al cliente.

Truco: La caché del núcleo tiene un método especial *getLog()*, el cual devuelve una cadena que representa lo que sucedió en la capa de la caché. En el entorno de desarrollo, se usa para depurar y validar la estrategia de caché:

```
error_log($kernel->getLog());
```

El objeto `AppCache` tiene una configuración predeterminada sensible, pero la puedes afinar por medio de un conjunto de opciones que puedes configurar sustituyendo el método `getOptions()`:

```
// app/AppCache.php
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,
        );
```

```
    }
}
```

Truco: A menos que la sustituyas en `getOptions()`, la opción `debug` se establecerá automáticamente al valor de depuración del `AppKernel` envuelto.

Aquí está una lista de las principales opciones:

- `default_ttl`: El número de segundos que una entrada de caché se debe considerar nueva cuando no hay información fresca proporcionada explícitamente en una respuesta. Las cabeceras explícitas `Cache-Control` o `Expires` sustituyen este valor (predeterminado: `0`);
- `private_headers`: Conjunto de cabeceras de la petición que desencadenan el comportamiento "privado" `Cache-Control` en las respuestas en que la respuesta explícitamente no es `pública` o `privada` vía una directiva `Cache-Control`. (predeterminado: `Authorization` y `Cookie`);
- `allow_reload`: Especifica si el cliente puede forzar una recarga desde caché incluyendo una directiva `Cache-Control` "no-cache" en la petición. Selecciona `true` para cumplir con la RFC 2616 (por omisión: `false`);
- `allow_revalidate`: Especifica si el cliente puede forzar una revalidación de caché incluyendo una directiva `Cache-Control max-age = 0` en la petición. Selecciona `true` para cumplir con la RFC 2616 (por omisión: `false`);
- `stale_while_revalidate`: Especifica el número predeterminado de segundos (la granularidad es el segundo como la precisión de respuesta TTL es un segundo) en el que la memoria caché puede regresar inmediatamente una respuesta obsoleta mientras que revalida en el fondo (por omisión: `2`), este valor lo reemplaza `stale-while-revalidate` de la extensión HTTP `Cache-Control` (consulta la RFC 5.861);
- `stale_if_error`: Especifica el número de segundos predeterminado (la granularidad es el segundo) durante el cual la caché puede servir una respuesta obsoleta cuando se detecta un error (por omisión: `60`). Este valor lo reemplaza `stale-if-error` de la extensión HTTP `Cache-Control` (consulta la RFC 5861).

Si `debug` es `true`, *Symfony2* automáticamente agrega una cabecera `X-Symfony-Cache` a la respuesta que contiene información útil acerca de aciertos y errores de caché.

Cambiando de un delegado inverso a otro

El delegado inverso de *Symfony2* es una gran herramienta a utilizar en el desarrollo de tu sitio web o al desplegar tu web en un servidor compartido donde no puedes instalar nada más allá que código *PHP*. Pero está escrito en *PHP*, no puede ser tan rápido como un delegado escrito en C. Es por eso que recomendamos usar *Varnish* o *Squid* en tus servidores de producción de ser posible. La buena nueva es que el cambio de un servidor sustituto a otro es fácil y transparente, sin modificar el código necesario en tu aplicación. Comienza fácilmente con el delegado inverso de *Symfony2* y actualiza a *Varnish* cuando aumente el tráfico.
Para más información sobre el uso de *Varnish* con *Symfony2*, consulta el capítulo *Cómo usar Varnish* (Página 402) del recetario.

Nota: El rendimiento del delegado inverso de *Symfony2* es independiente de la complejidad de tu aplicación. Eso es porque el núcleo de tu aplicación sólo se inicia cuando la petición se debe remitir a ella.

2.24.3 Introducción a la memoria caché *HTTP*

Para aprovechar las ventajas de las capas de memoria caché disponibles, tu aplicación se debe poder comunicar con las respuestas que son memorizables y las reglas que rigen cuándo y cómo la caché será obsoleta. Esto se hace ajustando las cabeceras de caché *HTTP* en la respuesta.

Truco: Ten en cuenta que "HTTP" no es más que el lenguaje (un lenguaje de texto simple) que los clientes web (navegadores, por ejemplo) y los servidores web utilizan para comunicarse entre sí. Cuando hablamos de la memoria caché *HTTP*, estamos hablando de la parte de ese lenguaje que permite a los clientes y servidores intercambiar información relacionada con la memoria caché.

HTTP especifica cuatro cabeceras de memoria caché de la respuesta en que estamos interesados:

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

La cabecera más importante y versátil es la cabecera `Cache-Control`, la cual en realidad es una colección de variada información de la caché.

Nota: Cada una de las cabeceras se explica en detalle en la sección *Caducidad y validación HTTP* (Página 215).

La cabecera `Cache-Control`

La cabecera `Cache-Control` es la única que no contiene una, sino varias piezas de información sobre la memoria caché de una respuesta. Cada pieza de información está separada por una coma:

```
Cache-Control: private, max-age=0, must-revalidate

Cache-Control: max-age=3600, must-revalidate
```

Symfony proporciona una abstracción de la cabecera `Cache-Control` para hacer más manejable su creación:

```
$respuesta = new Response();

// marca la respuesta como publica o privada
$respuesta->setPublic();
$respuesta->setPrivate();

// fija la edad máxima de privado o compartido
$respuesta->setMaxAge(600);
$respuesta->setSharedMaxAge(600);

// fija una directiva personalizada Cache-Control
$respuesta->headers->addCacheControlDirective('must-revalidate', true);
```

Respuestas públicas frente a privadas

Ambas, la pasarela de caché y la caché sustituta, son consideradas como cachés "compartidas" debido a que el contenido memorizado en caché es compartido por más de un usuario. Si cada vez equivocadamente una memoria caché compartida almacena una respuesta específica al usuario, posteriormente la puede devolver a cualquier cantidad de

usuarios diferentes. ¡Imagina si la información de tu cuenta se memoriza en caché y luego la regresa a todos los usuarios posteriores que soliciten la página de su cuenta!

Para manejar esta situación, cada respuesta se puede fijar para que sea pública o privada:

- *public*: Indica que la respuesta se puede memorizar en caché por ambas cachés privadas y compartidas;
- *private*: Indica que la totalidad o parte del mensaje de la respuesta es para un solo usuario y no se debe tener en caché en una memoria caché compartida.

Symfony por omisión conservadoramente fija cada respuesta para que sea privada. Para aprovechar las ventajas de cachés compartidas (como la sustituta inversa de *Symfony2*), la respuesta explícitamente se fijará como pública.

Métodos seguros

La memoria caché *HTTP* sólo funciona para métodos *HTTP* "seguros" (como *GET* y *HEAD*). Estar seguro significa que nunca cambia de estado la aplicación en el servidor al servir la petición (por supuesto puedes registrar información, datos de la caché, etc.) Esto tiene dos consecuencias muy razonables:

- *Nunca* debes cambiar el estado de tu aplicación al responder a una petición GET o HEAD. Incluso si no utilizas una pasarela caché, la presencia de la caché sustituta significa que alguna petición `GET` o `HEAD` puede o no llegar a tu servidor.
- No esperas métodos PUT, POST o DELETE en caché. Estos métodos están diseñados para utilizarse al mutar el estado de tu aplicación (por ejemplo, borrar una entrada de blog). La memoria caché debe impedir que determinadas peticiones toquen y muten tu aplicación.

Reglas de caché y valores predeterminados

HTTP 1.1 por omisión, permite a cualquiera memorizar en caché a menos que haya una cabecera `Cache-Control` explícita. En la práctica, la mayoría de cachés no hacen nada cuando solicitan una galleta, una cabecera de autorización, utilizar un método no seguro (es decir, PUT, POST, DELETE), o cuando las respuestas tienen código de redirección de estado.

Symfony2 automáticamente establece una sensible y conservadora cabecera `Cache-Control` cuando esta no está definida por el desarrollador, siguiendo estas reglas:

- Si no has definido cabecera caché (`Cache-Control`, `Expires`, `ETag` o `Last-Modified`), `Cache-Control` es establecida en `no-cache`, lo cual significa que la respuesta no se memoriza en caché;
- Si `Cache-Control` está vacía (pero una de las otras cabeceras de caché está presente), su valor se establece en `private, must-revalidate`;
- Pero si por lo menos una directiva `Cache-Control` está establecida, y no se han añadido directivas `public` o `private` de forma explícita, *Symfony2* agrega la directiva `private` automáticamente (excepto cuando `s-maxage` está establecida).

2.24.4 Caducidad y validación *HTTP*

La especificación *HTTP* define dos modelos de memoria caché:

- Con el modelo de caducidad, sólo tienes que especificar el tiempo en que la respuesta se debe considerar "fresca" incluyendo una cabecera `Cache-Control` y/o una `Expires`. Las cachés que entienden de expiración no harán la misma petición hasta que la versión en caché llegue a su fecha de caducidad y se convierta en "obsoleta".
- Cuando las páginas realmente son muy dinámicas (es decir, su representación cambia frecuentemente), a menudo es necesario el modelo de validación. Con este modelo, la memoria caché memoriza la respuesta, pero, pregunta al servidor en cada petición si la respuesta memorizada sigue siendo válida. La aplicación utiliza un

identificador de respuesta único (la cabecera `Etag`) y/o una marca de tiempo (la cabecera `Last-Modified`) para comprobar si la página ha cambiado desde su caché.

El objetivo de ambos modelos es nunca generar la misma respuesta en dos ocasiones dependiendo de una caché para almacenar y devolver respuestas "fresco".

Leyendo la especificación *HTTP*

La especificación *HTTP* define un lenguaje sencillo pero potente en el cual clientes y servidores se pueden comunicar. Como desarrollador web, el modelo petición-respuesta de la especificación domina nuestro trabajo. Lamentablemente, el documento de la especificación real - RFC 2616 - puede ser difícil de leer.
Hay un esfuerzo en curso (HTTP Bis) para reescribir la RFC 2616. Este no describe una nueva versión de *HTTP*, sino sobre todo aclara la especificación *HTTP* original. También mejora la organización de la especificación dividiéndola en siete partes, todo lo relacionado con la memoria caché *HTTP* se puede encontrar en dos partes dedicadas (P4 - Petición condicional y P6 - En caché: Exploración y caché intermediaria).
Como desarrollador web, te invitamos a leer la especificación. Su claridad y poder - incluso más de diez años después de su creación - tiene un valor incalculable. No te desanimes por la apariencia de la especificación - su contenido es mucho más bello que la cubierta.

Caducidad

El modelo de caducidad es el más eficiente y simple de los dos modelos de memoria caché y se debe utilizar siempre que sea posible. Cuando una respuesta se memoriza en caché con una caducidad, la caché memorizará la respuesta y la enviará directamente sin tocar a la aplicación hasta que esta caduque.

El modelo de caducidad se puede lograr usando una de dos, casi idénticas, cabeceras *HTTP*: `Expires` o `Cache-Control`.

Caducidad con la cabecera `Expires`

De acuerdo con la especificación *HTTP* "el campo de la cabecera `Expires` da la fecha/hora después de la cual se considera la respuesta es vieja". La cabecera `Expires` se puede establecer con el método `setExpires()` de `Respuesta`. Esta necesita una instancia de `DateTime` como argumento:

```
$date = new DateTime();
$date->modify('+600 seconds');

$respuesta->setExpires($date);
```

El resultado de la cabecera HTTP se verá así:

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

Nota: El método `setExpires()` automáticamente convierte la fecha a la zona horaria GMT como lo requiere la especificación.

La cabecera `Expires` adolece de dos limitaciones. En primer lugar, los relojes en el servidor Web y la caché (por ejemplo el navegador) deben estar sincronizados. Luego, la especificación establece que "los servidores *HTTP*/1.1 no deben enviar a `Expires` fechas de más de un año en el futuro".

Caducidad con la cabecera `Cache-Control`

Debido a las limitaciones de la cabecera `Expires`, la mayor parte del tiempo, debes usar la cabecera `Cache-Control` en su lugar. Recordemos que la cabecera `Cache-Control` se utiliza para especificar muchas directivas de caché diferentes. Para caducidad, hay dos directivas, `max-age` y `s-maxage`. La primera la utilizan todas las cachés, mientras que la segunda sólo se tiene en cuenta por las cachés compartidas:

```
// Establece el número de segundos después de que la
// respuesta ya no se debe considerar fresca
$respuesta->setMaxAge(600);

// Lo mismo que la anterior pero sólo para cachés compartidas
$respuesta->setSharedMaxAge(600);
```

La cabecera `Cache-Control` debería tener el siguiente formato (el cual puede tener directivas adicionales):

```
Cache-Control: max-age=600, s-maxage=600
```

Validando

Cuando un recurso se tiene que actualizar tan pronto como se realiza un cambio en los datos subyacentes, el modelo de caducidad se queda corto. Con el modelo de caducidad, no se pedirá a la aplicación que devuelva la respuesta actualizada hasta que la caché finalmente se convierta en obsoleta.

El modelo de validación soluciona este problema. Bajo este modelo, la memoria caché sigue almacenando las respuestas. La diferencia es que, por cada petición, la caché pregunta a la aplicación cuando o no la respuesta memorizada sigue siendo válida. Si la caché todavía *es* válida, tu aplicación debe devolver un código de estado 304 y no el contenido. Esto le dice a la caché que está bien devolver la respuesta memorizada.

Bajo este modelo, sobre todo ahorras ancho de banda ya que la representación no se envía dos veces al mismo cliente (en su lugar se envía una respuesta 304). Pero si diseñas cuidadosamente tu aplicación, es posible que puedas obtener los datos mínimos necesarios para enviar una respuesta 304 y ahorrar CPU también (más abajo puedes ver una implementación de ejemplo).

Truco: El código de estado 304 significa "No Modificado". Es importante porque este código de estado *no* tiene el contenido real solicitado. En cambio, la respuesta simplemente es un ligero conjunto de instrucciones que indican a la caché que se debe utilizar la versión almacenada.

Al igual que con la caducidad, hay dos cabeceras *HTTP* diferentes que se pueden utilizar para implementar el modelo de validación: `Etag` y `Last-Modified`.

Validación con la cabecera `ETag`

La cabecera `ETag` es una cabecera de cadena (llamada "entidad-etiqueta") que identifica unívocamente una representación del recurso destino. Este es generado completamente y establecido por tu aplicación de modo que puedes decir, por ejemplo, si el recurso memorizado `/sobre` está al día con el que tu aplicación iba a devolver. Una `ETag` es como una huella digital y se utiliza para comparar rápidamente si dos versiones diferentes de un recurso son equivalentes. Como las huellas digitales, cada `ETag` debe ser única en todas las representaciones de un mismo recurso.

Vamos a caminar a través de una aplicación sencilla que genera el ETag como el md5 del contenido:

```
public function indexAction()
{
    $respuesta = $this->render('MiBundle:Main:index.html.twig');
    $respuesta->setETag(md5($respuesta->getContent()));
```

```
    $respuesta->isNotModified($this->getRequest());

    return $respuesta;
}
```

El método `Response::isNotModified()` compara la `ETag` enviada en la `Petición` con la enviada en la `Respuesta`. Si ambas coinciden, el método establece automáticamente el código de estado de la `Respuesta` a 304.

Este algoritmo es bastante simple y muy genérico, pero es necesario crear la `Respuesta` completa antes de ser capaz de calcular el ETag, lo cual es subóptimo. En otras palabras, esta ahorra ancho de banda, pero no ciclos de CPU.

En la sección *Optimizando tu código con validación* (Página 219), vamos a mostrar cómo puedes utilizar la validación de manera más inteligente para determinar la validez de una caché sin hacer tanto trabajo.

Truco: *Symfony2* también apoya ETags débiles pasando `true` como segundo argumento del método **:method:'Symfony\\Component\\HttpFoundation\\Response::setETag'**.

Validación con la cabecera `Last-Modified`

La cabecera `Last-Modified` es la segunda forma de validación. De acuerdo con la especificación *HTTP*, "El campo de la cabecera `Last-Modified` indica la fecha y hora en que el servidor origen considera que la representación fue modificada por última vez". En otras palabras, la aplicación decide si o no el contenido memorizado se ha actualizado en función de si es o no ha sido actualizado desde que la respuesta entró en caché.

Por ejemplo, puedes utilizar la última fecha de actualización de todos los objetos necesarios para calcular la representación del recurso como valor para el valor de la cabecera `Last-Modified`:

```
public function showAction($articleSlug)
{
    // ...

    $articuloFecha = new \DateTime($articulo->getUpdatedAt());
    $autorFecha = new \DateTime($autor->getUpdatedAt());

    $date = $autorFecha > $articuloFecha ? $autorFecha : $articuloFecha;

    $respuesta->setLastModified($fecha);
    $respuesta->isNotModified($this->getRequest());

    return $respuesta;
}
```

El método `Response::isNotModified()` compara la cabecera `If-Modified-Since` enviada por la petición con la cabecera `Last-Modified` situada en la respuesta. Si son equivalentes, la `Respuesta` establecerá un código de estado 304.

Nota: La cabecera `If-Modified-Since` de la petición es igual a la cabecera `Last-Modified` de la última respuesta enviada al cliente por ese recurso en particular. Así es como se comunican el cliente y el servidor entre ellos y deciden si el recurso se ha actualizado desde que se memorizó.

Optimizando tu código con validación

El objetivo principal de cualquier estrategia de memoria caché es aligerar la carga de la aplicación. Dicho de otra manera, cuanto menos hagas en tu aplicación para devolver una respuesta 304, mejor. El método `Response::isNotModified()` hace exactamente eso al exponer un patrón simple y eficiente:

```
public function showAction($articleSlug)
{
    // Obtiene la mínima información para calcular la ETag
    // o el valor de Last-Modified (basado en la petición,
    // los datos se recuperan de una base de datos o un par
    // clave-valor guardado, por ejemplo)
    $articulo = // ...

    // crea una respuesta con una ETag y/o una cabecera Last-Modified
    $respuesta = new Response();
    $respuesta->setETag($articulo->computeETag());
    $respuesta->setLastModified($articulo->getPublishedAt());

    // verifica que la respuesta no sea modificada por la petición dada
    if ($respuesta->isNotModified($this->getRequest())) {
        // devuelve inmediatamente la Respuesta 304
        return $respuesta;
    } else {
        // aquí hace más trabajo - como recuperar más datos
        $comentarios = // ...

        // o reproduce una plantilla con la $respuesta que ya hemos iniciado
        return $this->render(
            'MiBundle:MiController:articulo.html.twig',
            array('articulo' => $articulo, 'comentarios' => $comentarios),
            $respuesta
        );
    }
}
```

Cuando la `Respuesta` no es modificada, el `isNotModified()` automáticamente fija el código de estado de la respuesta a `304`, remueve el contenido, y remueve algunas cabeceras que no deben estar presentes en respuestas `304` (consulta **:method:'Symfony\\Component\\HttpFoundation\\Response::setNotModified'**).

Variando la respuesta

Hasta ahora, hemos supuesto que cada *URI* tiene exactamente una representación del recurso destino. De forma predeterminada, la memoria caché *HTTP* se realiza mediante la *URI* del recurso como la clave de caché. Si dos personas solicitan la misma *URI* de un recurso memorizable, la segunda persona recibirá la versión en caché.

A veces esto no es suficiente y diferentes versiones de la misma *URI* necesitan memorizarse en caché basándose en uno o más valores de las cabeceras de la petición. Por ejemplo, si comprimes las páginas cuando el cliente lo permite, cualquier *URI* tiene dos representaciones: una cuando el cliente es compatible con la compresión, y otra cuando no. Esta determinación se hace por el valor de la cabecera `Accept-Encoding` de la petición.

En este caso, necesitamos que la memoria almacene una versión comprimida y otra sin comprimir de la respuesta para la *URI* particular y devolverlas basándose en el valor de la cabecera `Accept-Encoding`. Esto se hace usando la cabecera `Vary` de la respuesta, la cual es una lista separada por comas de diferentes cabeceras cuyos valores lanzan una representación diferente de los recursos solicitados:

```
Vary: Accept-Encoding, User-Agent
```

Truco: Esta cabecera `Vary` particular debería memorizar diferentes versiones de cada recurso en base a la *URI* y el valor de las cabeceras `Accept-Encoding` y `User-Agent` de la petición.

El objeto `Respuesta` ofrece una interfaz limpia para gestionar la cabecera `Vary`:

```
// establece una cabecera vary
$respuesta->setVary('Accept-Encoding');

// establece múltiples cabeceras vary
$respuesta->setVary(array('Accept-Encoding', 'User-Agent'));
```

El método `setVary()` toma un nombre de cabecera o un arreglo de nombres de cabecera de cual respuesta varía.

Caducidad y validación

Por supuesto, puedes utilizar tanto la caducidad como la validación de la misma `Respuesta`. La caducidad gana a la validación, te puedes beneficiar de lo mejor de ambos mundos. En otras palabras, utilizando tanto la caducidad como la validación, puedes instruir a la caché para que sirva el contenido memorizado, mientras que revisas de nuevo algún intervalo (de caducidad) para verificar que el contenido sigue siendo válido.

Más métodos de respuesta

La clase Respuesta proporciona muchos métodos más relacionados con la caché. Estos son los más útiles:

```
// Marca la respuesta como obsoleta
$respuesta->expire();

// Fuerza a la respuesta a devolver una adecuada respuesta 304 sin contenido
$respuesta->setNotModified();
```

Además, puedes confugurar muchas de las cabeceras *HTTP* relacionadas con la caché a través del método `setCache()`:

```
// Establece la configuración de caché en una llamada
$respuesta->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

2.24.5 Usando inclusión del borde lateral

Las pasarelas de caché son una excelente forma de hacer que tu sitio web tenga un mejor desempeño. Pero tienen una limitación: sólo podrán memorizar páginas enteras. Si no puedes memorizar todas las páginas o si partes de una página tienen "más" elementos dinámicos, se te acabó la suerte. Afortunadamente, *Symfony2* ofrece una solución para estos casos, basada en una tecnología llamada ESI, o Inclusión de bordes laterales (Edge Side Includes). Akamaï escribió esta especificación hace casi 10 años, y esta permite que partes específicas de una página tengan una estrategia de memorización diferente a la de la página principal.

La especificación ESI describe las etiquetas que puedes incrustar en tus páginas para comunicarte con la pasarela de caché. *Symfony2* sólo implementa una etiqueta, `include`, ya que es la única útil fuera del contexto de Akamaï:

```
<html>
    <body>
        Algún contenido

        <!-- aquí integra el contenido de otra página -->
        <esi:include src="http://..." />

        Más contenido
    </body>
</html>
```

Nota: Observa que en el ejemplo cada etiqueta ESI tiene una *URL* completamente cualificada. Una etiqueta ESI representa un fragmento de página que se puede recuperar a través de la *URL*.

Cuando se maneja una petición, la pasarela de caché obtiene toda la página de su caché o la pide a partir de la interfaz de administración de tu aplicación. Si la respuesta contiene una o más etiquetas ESI, estas se procesan de la misma manera. En otras palabras, la pasarela caché o bien, recupera el fragmento de página incluida en su caché o de nuevo pide el fragmento de página desde la interfaz de administración de tu aplicación. Cuando se han resuelto todas las etiquetas ESI, la pasarela caché une cada una en la página principal y envía el contenido final al cliente.

Todo esto sucede de forma transparente a nivel de la pasarela caché (es decir, fuera de tu aplicación). Como verás, si decides tomar ventaja de las etiquetas ESI, *Symfony2* hace que el proceso de incluirlas sea casi sin esfuerzo.

Usando ESI en *Symfony2*

Primero, para usar ESI, asegúrate de activarlo en la configuración de tu aplicación:

- *YAML*

  ```
  # app/config/config.yml
  framework:
      # ...
      esi: { enabled: true }
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <framework:config ...>
      <!-- ... -->
      <framework:esi enabled="true" />
  </framework:config>
  ```

- *PHP*

  ```
  // app/config/config.php
  $contenedor->loadFromExtension('framework', array(
      // ...
      'esi'    => array('enabled' => true),
  ));
  ```

Ahora, supongamos que tenemos una página que es relativamente estable, a excepción de un teletipo de noticias en la parte inferior del contenido. Con ESI, podemos memorizar el teletipo de noticias independiente del resto de la página.

```
public function indexAction()
{
    $respuesta = $this->render('MiBundle:MiController:index.html.twig');
    $respuesta->setSharedMaxAge(600);

    return $respuesta;
}
```

En este ejemplo, hemos dado al caché de la página completa un tiempo de vida de diez minutos. En seguida, vamos a incluir el teletipo de noticias en la plantilla incorporando una acción. Esto se hace a través del ayudante `render` (consulta la sección *Incrustando controladores* (Página 106) para más detalles).

Como el contenido integrado viene de otra página (o controlador en este caso), *Symfony2* utiliza el ayudante `render` estándar para configurar las etiquetas ESI:

- *Twig*

  ```
  {% render '...:noticias' with {}, {'standalone': true} %}
  ```

- *PHP*

  ```
  <?php echo $view['actions']->render('...:noticias', array(), array('standalone' => true)) ?>
  ```

Al establecer `standalone` a `true`, le dices a *Symfony2* que la acción se debe reproducir como una etiqueta ESI. Tal vez te preguntes por qué querría usar un ayudante en vez de escribir la etiqueta ESI en si misma. Eso es porque usar un ayudante hace que tu aplicación trabaje, incluso si no hay pasarela caché instalada. Vamos a ver cómo funciona.

Cuando `standalone` es `false` (predeterminado), *Symfony2* combina el contenido de la página incluida en la principal antes de enviar la respuesta al cliente. Pero cuando `standalone` es `true`, *y* si *Symfony2* detecta que está hablando con una pasarela caché compatible con ESI, genera una etiqueta `include` ESI. Pero si no hay una pasarela caché o si no es compatible con ESI, *Symfony2* termina fusionando el contenido de las páginas incluidas en la principal como lo habría hecho si `standalone` se hubiera establecido en `false`.

Nota: *Symfony2* detecta si una pasarela caché admite ESI a través de otra especificación Akamaï que fuera de la caja es compatible con el delegado inverso de *Symfony2*.

La acción integrada ahora puede especificar sus propias reglas de caché, totalmente independientes de la página principal.

```
public function noticiasAction()
{
  // ...

  $respuesta->setSharedMaxAge(60);
}
```

Con ESI, la caché de la página completa será válida durante 600 segundos, pero la caché del componente de noticias sólo dura 60 segundos.

Un requisito de ESI, sin embargo, es que la acción incrustada sea accesible a través de una *URL* para que la pasarela caché se pueda buscar independientemente del resto de la página. Por supuesto, una acción no se puede acceder a través de una *URL* a menos que haya una ruta que apunte a la misma. *Symfony2* se encarga de esto a través de una ruta genérica y un controlador. Para que la etiqueta ESI `include` funcione correctamente, debes definir la ruta `_internal`:

- *YAML*

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

```
    resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
    prefix:   /_internal
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@FrameworkBundle/Resources/config/routing/internal.xml" prefix="/_internal
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion->addCollection($cargador->import('@FrameworkBundle/Resources/config/routing/internal.

return $coleccion;
```

Truco: Puesto que esta ruta permite que todas las acciones se accedan a través de una *URL*, posiblemente desees protegerla usando el cortafuegos de *Symfony2* (permitiendo acceder al rango IP del delegado inverso).

Una gran ventaja de esta estrategia de memoria caché es que puedes hacer tu aplicación tan dinámica como sea necesario y al mismo tiempo, tocar la aplicación lo menos posible.

Nota: Una vez que comiences a usar ESI, recuerda usar siempre la directiva `s-maxage` en lugar de `max-age`. Como el navegador nunca recibe recursos agregados, no es consciente del subcomponente, y por lo tanto obedecerá la directiva `max-age` y memorizará la página completa. Y no quieres eso.

El ayudante `render` es compatible con otras dos útiles opciones:

- `alt`: utilizada como el atributo `alt` en la etiqueta ESI, el cual te permite especificar una *URL* alternativa para utilizarla si no se puede encontrar `src`;
- `ignore_errors`: si se establece en `true`, se agrega un atributo `onerror` a la ESI con un valor de `continue` indicando que, en caso de una falla, la pasarela caché simplemente debe eliminar la etiqueta ESI silenciosamente.

2.24.6 Invalidando la caché

> "Sólo hay dos cosas difíciles en Ciencias de la Computación. Invalidación de caché y nombrar cosas"
> –Phil Karlton

Nunca debería ser necesario invalidar los datos memorizados en caché porque la invalidación ya se tiene en cuenta de forma nativa en los modelos de caché *HTTP*. Si utilizas la validación, por definición, no será necesario invalidar ninguna cosa, y si se utiliza la caducidad y necesitas invalidar un recurso, significa que estableciste la fecha de caducidad muy adelante en el futuro.

Nota: También es porque no existe un mecanismo de invalidación que cualquier delegado inverso pueda utilizar, sin cambiar nada en el código de tu aplicación.

En realidad, todos los sustitutos inversos proporcionan una manera de purgar datos almacenados en caché, pero la debes evitar tanto como sea posible. La forma más habitual es purgar la caché de una *URL* dada solicitándola con el método especial `PURGE` de *HTTP*.

Aquí está cómo puedes configurar la caché sustituta inversa de *Symfony2* para apoyar al método `PURGE` de *HTTP*:

```
// app/AppCache.php
class AppCache extends Cache
{
    protected function invalidate(Request $peticion)
    {
        if ('PURGE' !== $peticion->getMethod()) {
            return parent::invalidate($peticion);
        }

        $respuesta = new Response();
        if (!$this->store->purge($peticion->getUri())) {
            $respuesta->setStatusCode(404, 'Not purged');
        } else {
            $respuesta->setStatusCode(200, 'Purged');
        }

        return $respuesta;
    }
}
```

Prudencia: De alguna manera, debes proteger el método `PURGE` HTTP para evitar que alguien aleatoriamente purgue los datos memorizados.

2.24.7 Resumen

Symfony2 fue diseñado para seguir las reglas probadas de la carretera: HTTP. El almacenamiento en caché no es una excepción. Dominar el sistema caché de *Symfony2* significa familiarizarse con los modelos de caché *HTTP* y usarlos eficientemente. Esto significa que, en lugar de confiar sólo en la documentación de *Symfony2* y ejemplos de código, tienes acceso a un mundo de conocimientos relacionados con la memorización en caché *HTTP* y la pasarela caché, tal como *Varnish*.

2.24.8 Aprende más en el recetario

- *Cómo utilizar Varnish para acelerar mi sitio web* (Página 402)

2.25 Traduciendo

El término "internacionalización" se refiere al proceso de abstraer cadenas y otros elementos específicos de la configuración regional de tu aplicación a una capa donde se puedan traducir y convertir basándose en la configuración regional del usuario (es decir, el idioma y país). Para el texto, esto significa envolver cada uno con una función capaz de traducir el texto (o "mensaje") al idioma del usuario:

```
// el texto *siempre* se imprime en Inglés
echo 'Hello World';

// el texto se puede traducir al idioma del usuario final o predeterminado en Inglés
echo $translator->trans('Hello World');
```

Nota: El término *locale* se refiere más o menos al lenguaje y país del usuario. Esta puede ser cualquier cadena que entonces la aplicación utilice para manejar las traducciones y otras diferencias de formato (por ejemplo, formato de moneda). Recomendamos el código ISO639-1 para el *idioma*, un subrayado (_), luego el código ISO3166 para el *país* (por ejemplo `es_ES` para Español/España).

En este capítulo, aprenderemos cómo preparar una aplicación para apoyar varias configuraciones regionales y, a continuación cómo crear traducciones para múltiples regiones. En general, el proceso tiene varios pasos comunes:

1. Habilitar y configurar el componente `Translation` de *Symfony*;
2. Abstraer cadenas (es decir, "mensajes") envolviéndolas en llamadas al `Traductor`;
3. Crear recursos de traducción para cada configuración regional compatible, la cual traduce cada mensaje en la aplicación;
4. Determinar, establecer y administrar la configuración regional del usuario en la sesión.

2.25.1 Configurando

Las traducciones están a cargo de un *servicio* `Traductor` que utiliza la configuración regional del usuario para buscar y devolver mensajes traducidos. Antes de usarlo, habilita el `Traductor` en tu configuración:

- *YAML*

```
# app/config/config.yml
framework:
    translator: { fallback: en }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:translator fallback="en" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('framework', array(
    'translator' => array('fallback' => 'en'),
));
```

La opción `fallback` define la configuración regional de reserva cuando la traducción no existe en la configuración regional del usuario.

Truco: Cuando la traducción no existe para una configuración regional, el primer traductor intenta encontrar la traducción para el idioma (`es` si el local es `es_ES` por ejemplo). Si esto también falla, busca una traducción utilizando la configuración regional de reserva.

La región utilizada en las traducciones es la almacenada en la sesión del usuario.

2.25.2 Traducción básica

La traducción de texto se hace a través del servicio `traductor` (`Symfony\Component\Translation\Translator`). Para traducir un bloque de texto (llamado un *mensaje*), utiliza el método **:method:'Symfony\\Component\\Translation\\Translator::trans'**. Supongamos, por ejemplo, que estamos traduciendo un simple mensaje desde el interior de un controlador:

```
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

    return new Response($t);
}
```

Cuando se ejecuta este código, *Symfony2* tratará de traducir el mensaje "Symfony2 is great", basándose en la variable `locale` del usuario. Para que esto funcione, tenemos que decirle a *Symfony2* la manera de traducir el mensaje a través de un "recurso de traducción", el cual es una colección de mensajes traducidos para una determinada configuración regional. Este "diccionario" de traducciones se puede crear en varios formatos diferentes, XLIFF es el formato recomendado:

- *XML*

```
<!-- messages.es.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>Symfony2 es magnífico</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// messages.es.php
return array(
    'Symfony2 is great' => 'Symfony2 es magnífico',
);
```

- *YAML*

```
# messages.es.yml
Symfony2 is great: Symfony2 es magnífico
```

Ahora, si el idioma de la configuración regional del usuario es el Español (por ejemplo, `es_ES` o `es_MX`), el mensaje será traducido a `Symfony2 es magnífico`.

El proceso de traducción

Para empezar a traducir el mensaje, *Symfony2* utiliza un proceso sencillo:

- Se determina el `locale` del usuario actual, el cual está almacenado en la sesión;
- Se carga un catálogo de mensajes traducidos desde los recursos de traducción definidos para la configuración de `locale` (por ejemplo, `es_ES`). Los mensajes de la configuración regional de reserva también se cargan y

se agregan al catálogo si no existen ya. El resultado final es un gran "diccionario" de traducciones. Consulta Catálogos de mensajes (Página 228) para más detalles;

- Si se encuentra el mensaje en el catálogo, devuelve la traducción. En caso contrario, el traductor devuelve el mensaje original.

Cuando se usa el método `trans()`, *Symfony2* busca la cadena exacta dentro del catálogo de mensajes apropiados y la devuelve (si existe).

Marcadores de posición en mensajes

A veces, se debe traducir un mensaje que contiene una variable:

```
public function indexAction($nombre)
{
    $t = $this->get('translator')->trans('Hello '.$nombre);

    return new Response($t);
}
```

Sin embargo, la creación de una traducción de esta cadena es imposible, ya que el traductor tratará de buscar el mensaje exacto, incluyendo las porciones variables (por ejemplo, "Hello Ryan" o "Hello Fabian"). En lugar de escribir una traducción de cada iteración posible de la variable `$nombre`, podemos reemplazar la variable con un "marcador de posición":

```
public function indexAction($nombre)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $nombre));

    new Response($t);
}
```

Symfony2 ahora busca una traducción del mensaje en bruto (`Hello %name%`) y *después* reemplaza los marcadores de posición con sus valores. La creación de una traducción se hace igual que antes:

- *XML*

```
<!-- messages.es.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Hola %name%</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// messages.es.php
return array(
    'Hello %name%' => 'Hola %name%',
);
```

- *YAML*

```
# messages.es.yml
'Hello %name%': Hola %name%
```

Nota: Los marcadores de posición pueden tomar cualquier forma, el mensaje completo se reconstruye usando la función strtr de *PHP*. Sin embargo, se requiere la notación `%var%` cuando se traduce en plantillas *Twig*, y en general es un convenio razonable a seguir.

Como hemos visto, la creación de una traducción es un proceso de dos pasos:

1. Abstraer el mensaje que se necesita traducir procesándolo con el `Traductor`.
2. Crear una traducción del mensaje para cada región que elijas apoyar.

El segundo paso se realiza creando catálogos de mensajes que definen las traducciones para cualquier número de lugares diferentes.

2.25.3 Catálogos de mensajes

Cuando se traduce un mensaje, *Symfony2* compila un catálogo de mensajes para la configuración regional del usuario y busca en ese una traducción del mensaje. Un catálogo de mensajes es como un diccionario de traducciones para una configuración regional específica. Por ejemplo, el catálogo de la configuración regional `es_ES` podría contener la siguiente traducción:

```
Symfony2 is Great => Symfony2 es magnífico
```

Es responsabilidad del desarrollador (o traductor) de una aplicación internacionalizada crear estas traducciones. Las traducciones son almacenadas en el sistema de archivos y descubiertas por *Symfony*, gracias a algunos convenios.

Truco: Cada vez que creas un *nuevo* recurso de traducción (o instalas un paquete que incluye un recurso de traducción), para que *Symfony* pueda descubrir el nuevo recurso de traducción, asegúrate de borrar la memoria caché con la siguiente orden:

```
php app/console cache:clear
```

Ubicación de traducción y convenciones de nomenclatura

Symfony2 busca archivos de mensajes (traducciones) en dos lugares:

- Para los mensajes que se encuentran en un paquete, los archivos de mensajes correspondientes deben vivir en el directorio `Resources/translations/` del paquete;
- Para sustituir la traducción de algún paquete, debes colocar los archivos de mensajes en el directorio `app/Resources/translations`.

El nombre del archivo de las traducciones también es importante ya que *Symfony2* utiliza una convención para determinar los detalles sobre las traducciones. Cada archivo de mensajes se debe nombrar de acuerdo con el siguiente patrón: `dominio.región.cargador`:

- **dominio**: Una forma opcional para organizar los mensajes en grupos (por ejemplo, `admin`, `navegación` o el valor predeterminado `messages`) - consulta Usando mensajes del dominio (Página 232);
- **región**: La región para la cual son las traducciones (por ejemplo, `es_ES`, `es`, etc.);
- **cargador**: ¿Cómo debe cargar y analizar el archivo *Symfony2* (por ejemplo, `XLIFF`, `php` o `yml`).

El cargador puede ser el nombre de cualquier gestor registrado. De manera predeterminada, *Symfony* incluye los siguientes cargadores:

- `xliff`: archivo XLIFF;
- `php`: archivo PHP;
- `yml`: archivo YAML.

La elección del cargador a utilizar es totalmente tuya y es una cuestión de gusto.

Nota: También puedes almacenar las traducciones en una base de datos, o cualquier otro almacenamiento, proporcionando una clase personalizada que implemente la interfaz `Symfony\Component\Translation\Loader\LoaderInterface`. Consulta `Cargadores de traducción personalizados` a continuación para aprender cómo registrar cargadores personalizados.

Creando traducciones

Cada archivo se compone de una serie de pares de identificador de traducción para el dominio y la configuración regional determinada. El `id` es el identificador de la traducción individual, y puede ser el mensaje en la región principal (por ejemplo, "Symfony is great") de tu aplicación o un identificador único (por ejemplo, "symfony2.great" - consulta el recuadro más abajo):

- *XML*

```
<!-- src/Acme/DemoBundle/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="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>Symfony2 es magnífico</target>
            </trans-unit>
            <trans-unit id="2">
                <source>symfony2.great</source>
                <target>Symfony2 es magnífico</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// src/Acme/DemoBundle/Resources/translations/messages.es.php
return array(
    'Symfony2 is great' => 'Symfony2 es magnífico',
    'symfony2.great'    => 'Symfony2 es magnífico',
);
```

- *YAML*

```
# src/Acme/DemoBundle/Resources/translations/messages.es.yml
Symfony2 is great: Symfony2 es magnífico
symfony2.great:    Symfony2 es magnífico
```

Symfony2 descubrirá estos archivos y los utilizará cuando traduce o bien "Symfony2 is graeat" o "symfony2.great" en un Idioma regional de Español (por ejemplo, `es_ES` o `es_MX`).

Usando mensajes reales o palabras clave

Este ejemplo ilustra las dos diferentes filosofías, cuando creas mensajes a traducir:

```
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

En el primer método, los mensajes están escritos en el idioma de la región predeterminada (Inglés en este caso). Ese mensaje se utiliza entonces como el "id" al crear traducciones.
En el segundo método, los mensajes en realidad son "palabras clave" que transmiten la idea del mensaje. Entonces, la palabra clave del mensaje se utiliza como el "id" para las traducciones. En este caso, la traducción se debe hacer para la región predeterminada (es decir, para traducir `symfony2.great` a `Symfony2 es magnífico`).
El segundo método es útil porque la clave del mensaje no se tendrá que cambiar en cada archivo de la traducción si decidimos que el mensaje en realidad debería decir "Symfony2 es realmente magnífico" en la configuración regional predeterminada.
La elección del método a utilizar es totalmente tuya, pero a menudo se recomienda el formato de "palabra clave". Además, es compatible con archivos anidados en formato `php` y `yaml` para evitar repetir siempre lo mismo si utilizas palabras clave en lugar de texto real para tus identificadores:

- *YAML*

```
symfony2:
    is:
        great: Symfony2 es magnífico
        amazing: Symfony2 es asombroso
    has:
        bundles: Symfony2 tiene paquetes
user:
    login: Iniciar sesión
```

- *PHP*

```
return array(
    'symfony2' => array(
        'is' => array(
            'great' => 'Symfony2 es magnífico',
            'amazing' => 'Symfony2 es asombroso',
        ),
        'has' => array(
            'bundles' => 'Symfony2 tiene paquetes',
        ),
    ),
    'user' => array(
        'login' => 'Iniciar sesión',
    ),
);
```

Los niveles múltiples se acoplan en pares de id/traducción añadiendo un punto (.) entre cada nivel, por lo tanto los ejemplos anteriores son equivalentes a los siguientes:

- *YAML*

```
symfony2.is.great: Symfony2 es magnífico
symfony2.is.amazing: Symfony2 es asombroso
symfony2.has.bundles: Symfony2 tiene paquetes
user.login: Iniciar sesión
```

- *PHP*

```
return array(
    'symfony2.is.great' => 'Symfony2 es magnífico',
    'symfony2.is.amazing' => 'Symfony2 es asombroso',
    'symfony2.has.bundles' => 'Symfony2 tiene paquetes',
    'user.login' => 'Iniciar sesión',
);
```

2.25.4 Usando mensajes del dominio

Como hemos visto, los archivos de mensajes se organizan en las diferentes regiones a traducir. Los archivos de mensajes también se pueden organizar en "dominios". Al crear archivos de mensajes, el dominio es la primera porción del nombre de archivo. El dominio predeterminado es `menssages`. Por ejemplo, supongamos que, por organización, las traducciones se dividieron en tres diferentes ámbitos: `messages`, `admin` y `navegacion`. La traducción española debe tener los siguientes archivos de mensaje:

- `messages.es.xliff`
- `admin.es.xliff`
- `navegacion.es.xliff`

Al traducir las cadenas que no están en el dominio predeterminado (`messages`), debes especificar el dominio como tercer argumento de `trans()`:

```
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Symfony2 ahora buscará el mensaje en el dominio `admin` de la configuración regional del usuario.

2.25.5 Manejando la configuración regional del usuario

La configuración regional del usuario actual se almacena en la sesión y se puede acceder a través del servicio `sesión`:

```
$locale = $this->get('session')->getLocale();

$this->get('session')->setLocale('en_US');
```

Configuración regional predeterminada y reserva

Si la configuración regional no se ha establecido explícitamente en la sesión, el parámetro de configuración `fallback_locale` será utilizado por el `Traductor`. El predeterminado del parámetro es `en` (consulta la sección Configurando (Página 225)).

Alternativamente, puedes garantizar que un `locale` está establecido en la sesión del usuario definiendo un `default_locale` para el servicio sesión:

- *YAML*

```
# app/config/config.yml
framework:
    session: { default_locale: en }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:session default-locale="en" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('framework', array(
    'session' => array('default_locale' => 'en'),
));
```

`Locale` y la *URL*

Dado que la configuración regional del usuario se almacena en la sesión, puede ser tentador utilizar la misma *URL* para mostrar un recurso en muchos idiomas diferentes en función de la región del usuario. Por ejemplo, `http://www.ejemplo.com/contacto` podría mostrar el contenido en Inglés para un usuario y en Francés para otro. Por desgracia, esto viola una norma fundamental de la Web: que una *URL* particular devuelve el mismo recurso, independientemente del usuario. A fin de enturbiar el problema, ¿cual sería la versión del contenido indexado por los motores de búsqueda?

Una mejor política es incluir la configuración regional en la *URL*. Esto es totalmente compatible con el sistema de enrutado mediante el parámetro especial `_locale`:

- *YAML*

```
contacto:
    pattern:   /{_locale}/contacto
    defaults:  { _controller: AcmeDemoBundle:Contacto:index, _locale: en }
    requirements:
        _locale: en|es|de
```

- *XML*

```
<route id="contact" pattern="/{_locale}/contacto">
    <default key="_controller">AcmeDemoBundle:Contact:index</default>
    <default key="_locale">en</default>
    <requirement key="_locale">en|es|de</requirement>
</route>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('contact', new Route('/{_locale}/contacto', array(
    '_controller' => 'AcmeDemoBundle:Contacto:index',
    '_locale'     => 'en',
), array(
    '_locale'     => 'en|es|de'
)));

return $coleccion;
```

Cuando utilizas el parámetro especial `_locale` en una ruta, la configuración regional emparejada *automáticamente se establece en la sesión del usuario*. En otras palabras, si un usuario visita la *URI* `/es/contacto`, la región `es` se ajustará automáticamente según la configuración regional de la sesión del usuario.

Ahora puedes utilizar la configuración regional del usuario para crear rutas hacia otras páginas traducidas en tu aplicación.

2.25.6 Pluralización

La pluralización de mensajes es un tema difícil puesto que las reglas pueden ser bastante complejas. Por ejemplo, aquí tienes la representación matemática de las reglas de pluralización de Rusia:

```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10 <= 4) && (
```

Como puedes ver, en Ruso, puedes tener tres formas diferentes del plural, cada una da un índice de 0, 1 o 2. Para todas las formas, el plural es diferente, por lo que la traducción también es diferente.

Cuando una traducción tiene diferentes formas debido a la pluralización, puedes proporcionar todas las formas como una cadena separada por una tubería (`|`):

```
'Hay una manzana|Hay %count% manzanas'
```

Para traducir los mensajes pluralizados, utiliza el método **:method:'Symfony\\Component\\Translation\\Translator::transChoice'**:

```
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

El segundo argumento (`10` en este ejemplo), es el *número* de objetos descrito y se utiliza para determinar cual traducción usar y también para rellenar el marcador de posición `%count%`.

En base al número dado, el traductor elige la forma plural adecuada. En Inglés, la mayoría de las palabras tienen una forma singular cuando hay exactamente un objeto y una forma plural para todos los otros números (0, 2, 3...). Así pues, si `count` es `1`, el traductor utilizará la primera cadena (`Hay una manzana`) como la traducción. De lo contrario, utilizará `Hay %count% manzanas`.

Aquí está la traducción al Francés:

```
'Il y a %count% pomme|Il y a %count% pommes'
```

Incluso si la cadena tiene una apariencia similar (se compone de dos subcadenas separadas por un tubo), las reglas francesas son diferentes: la primera forma (no plural) se utiliza cuando `count` es `0` o `1`. Por lo tanto, el traductor utilizará automáticamente la primera cadena (`Il y a %count% pomme`) cuando `count` es `0` o `1`.

Cada región tiene su propio conjunto de reglas, con algunas que tienen hasta seis formas diferentes de plural con reglas complejas detrás de las cuales los números asignan a tal forma plural. Las reglas son bastante simples para Inglés y Francés, pero para el Ruso, puedes querer una pista para saber qué regla coincide con qué cadena. Para ayudar a los traductores, puedes "etiquetar" cada cadena:

```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

Las etiquetas realmente son pistas sólo para los traductores y no afectan a la lógica utilizada para determinar qué forma plural usar. Las etiquetas pueden ser cualquier cadena descriptiva que termine con dos puntos (`:`). Las etiquetas además no necesitan ser las mismas en el mensaje original cómo en la traducción.

Intervalo explícito de pluralización

La forma más fácil de pluralizar un mensaje es dejar que *Symfony2* utilice su lógica interna para elegir qué cadena se utiliza en base a un número dado. A veces, tendrás más control o quieres una traducción diferente para casos específicos (por `0`, o cuando el número es negativo, por ejemplo). Para estos casos, puedes utilizar intervalos matemáticos explícitos:

```
'{0} There is no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf] There are ma
```

Los intervalos siguen la notación ISO 31-11. La cadena anterior especifica cuatro intervalos diferentes: exactamente `0`, exactamente `1`, `2-19` y `20` y superior.

También puedes mezclar reglas matemáticas explícitas y estándar. En este caso, si la cuenta no corresponde con un intervalo específico, las reglas estándar entran en vigor después de remover las reglas explícitas:

```
'{0} There is no apples|[20,Inf] There are many apples|There is one apple|a_few: There are %count% ap
```

Por ejemplo, para `1` apple, la regla estándar `There is one apple` será utilizada. Para `2-19` apples, la segunda regla estándar `There are %count% apples` será seleccionada.

Un `Symfony\Component\Translation\Interval` puede representar un conjunto finito de números:

```
{1,2,3,4}
```

O números entre otros dos números:

```
[1, +Inf[
]-1,2[
```

El delimitador izquierdo puede ser `[` (inclusive) o `]` (exclusivo). El delimitador derecho puede ser `[` (exclusivo) o `]` (inclusive). Más allá de los números, puedes usar `-Inf` y `+Inf` para el infinito.

2.25.7 Traducciones en plantillas

La mayoría de las veces, la traducción ocurre en las plantillas. *Symfony2* proporciona apoyo nativo para ambas plantillas *Twig* y *PHP*.

Plantillas *Twig*

Symfony2 proporciona etiquetas *Twig* especializadas (`trans` y `transchoice`) para ayudar con la traducción de los mensajes de *bloques estáticos de texto*:

```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

La etiqueta `transchoice` obtiene automáticamente la variable `%count%` a partir del contexto actual y la pasa al traductor. Este mecanismo sólo funciona cuando se utiliza un marcador de posición después del patrón `%var%`.

Truco: Si necesitas utilizar el carácter de porcentaje (`%`) en una cadena, lo tienes que escapar duplicando el siguiente: `{% trans %}Porcentaje: %percent%%%{% endtrans %}`

También puedes especificar el dominio del mensaje y pasar algunas variables adicionales:

```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% transchoice count with {'%nombre%': 'Fabien'} from "app" %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Los filtros `trans` y `transchoice` se pueden utilizar para traducir *texto variable* y expresiones complejas:

```
{{ message | trans }}

{{ message | transchoice(5) }}

{{ message | trans({'%name%': 'Fabien'}, "app") }}

{{ message | transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Truco: Usar etiquetas de traducción o filtros tiene el mismo efecto, pero con una sutil diferencia: la salida escapada automáticamente sólo se aplica a las variables traducidas usando un filtro. En otras palabras, si necesitas estar seguro de que tu variable traducida *no* se escapó en la salida, debes aplicar el filtro crudo después de la traducción del filtro:

```
{# text translated between tags is never escaped #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# a variable translated via a filter is escaped by default #}
{{ message | trans | raw }}

{# but static strings are never escaped #}
{{ '<h3>foo</h3>' | trans }}
```

Plantillas *PHP*

El servicio de traductor es accesible en plantillas *PHP* a través del ayudante `traductor`:

```
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

2.25.8 Forzando la configuración regional del traductor

Al traducir un mensaje, *Symfony2* utiliza la configuración regional de la sesión del usuario o la configuración regional de `reserva` si es necesario. También puedes especificar manualmente la configuración regional utilizada para la traducción:

```
$this->get('translator')->trans(
    'Symfony2 is great',
    array(),
    'messages',
    'es_ES',
);

$this->get('translator')->trans(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10),
    'messages',
    'es_ES',
);
```

2.25.9 Traduciendo contenido de base de datos

La traducción del contenido de la base de datos la debe manejar *Doctrine* a través de la Extensión Translatable. Para más información, consulta la documentación de la biblioteca.

2.25.10 Resumen

Con el componente `Translation` de *Symfony2*, la creación de una aplicación internacionalizada ya no tiene que ser un proceso doloroso y se reduce a sólo algunos pasos básicos:

- Resumir los mensajes en tu aplicación envolviendo cada uno en el método **:method:‘Symfony\\Component\\Translation\\Translator::trans‘** o **:method:‘Symfony\\Component\\Translation\\Translator::transChoice‘**;
- Traducir cada mensaje en varias configuraciones regionales creando archivos de traducción de los mensajes. *Symfony2* descubre y procesa cada archivo porque su nombre sigue una convención específica;
- Administrar la configuración regional del usuario, la cual se almacena en la sesión.

2.26 Contenedor de servicios

Una moderna aplicación *PHP* está llena de objetos. Un objeto te puede facilitar la entrega de mensajes de correo electrónico, mientras que otro te puede permitir mantener información en una base de datos. En tu aplicación, puedes crear un objeto que gestiona tu inventario de productos, u otro objeto que procesa los datos de una *API* de terceros. El punto es que una aplicación moderna hace muchas cosas y está organizada en muchos objetos que se encargan de cada tarea.

En este capítulo, vamos a hablar de un objeto PHP especial en *Symfony2* que te ayuda a crear una instancia, organizar y recuperar muchos objetos de tu aplicación. Este objeto, llamado contenedor de servicios, te permitirá estandarizar y centralizar la forma en que se construyen los objetos en tu aplicación. El contenedor te facilita la vida, es superveloz, y enfatiza una arquitectura que promueve el código reutilizable y disociado. Y como todas las clases *Symfony2* básicas usan el contenedor, aprenderás cómo ampliar, configurar y utilizar cualquier objeto en *Symfony2*. En gran parte, el contenedor de servicios es el mayor contribuyente a la velocidad y extensibilidad de *Symfony2*.

Por último, configurar y usar el contenedor de servicios es fácil. Al final de este capítulo, te sentirás cómodo creando tus propios objetos y personalizando objetos de cualquier paquete de terceros a través del contenedor. Empezarás a escribir código más reutilizable, comprobable y disociado, simplemente porque el contenedor de servicios facilita la escritura de buen código.

2.26.1 ¿Qué es un servicio?

En pocas palabras, un *Servicio* es cualquier objeto PHP que realiza algún tipo de tarea “global”. Es un nombre genérico que se utiliza a propósito en informática para describir un objeto creado para un propósito específico (por ejemplo, la entrega de mensajes de correo electrónico). Cada servicio se utiliza en toda tu aplicación cada vez que necesites la funcionalidad específica que proporciona. No tienes que hacer nada especial para hacer un servicio: simplemente escribe una clase *PHP* con algo de código que realiza una tarea específica. ¡Felicidades, acabas de crear un servicio!

Nota: Por regla general, un objeto PHP es un servicio si se utiliza a nivel global en tu aplicación. Un solo servicio `Mailer` se utiliza a nivel global para enviar mensajes de correo electrónico mientras que muchos objetos `Mensaje` que entrega no son *servicios*. Del mismo modo, un objeto `producto` no es un servicio, sino un objeto `producto` que persiste objetos a una base de datos *es* un servicio.

Entonces, ¿cuál es la ventaja? La ventaja de pensar en "servicios" es que comienzas a pensar en la separación de cada parte de la funcionalidad de tu aplicación como una serie de servicios. Puesto que cada servicio se limita a un trabajo, puedes acceder fácilmente a cada servicio y usar su funcionalidad siempre que la necesites. Cada servicio también se puede probar y configurar más fácilmente, ya que está separado de la otra funcionalidad de tu aplicación. Esta idea se llama arquitectura orientada a servicios y no es única de *Symfony2* e incluso de *PHP*. Estructurando tu aplicación en torno a un conjunto de clases *Servicio* independientes es una bien conocida y confiable práctica mejor orientada a objetos. Estas habilidades son clave para ser un buen desarrollador en casi cualquier lenguaje.

2.26.2 ¿Qué es un contenedor de servicios?

Un *Contenedor de servicios* (o *contenedor de inyección de dependencias*) simplemente es un objeto PHP que gestiona la creación de instancias de servicios (es decir, objetos). Por ejemplo, supongamos que tenemos una clase *PHP* simple que envía mensajes de correo electrónico. Sin un contenedor de servicios, debemos crear manualmente el objeto cada vez que lo necesitemos:

```
use Acme\HolaBundle\Mailer;

$cartero = new Mailer('sendmail');
$cartero->send('ryan@foobar.net', ... );
```

Esto es bastante fácil. La clase imaginaria `Mailer` nos permite configurar el método utilizado para entregar los mensajes de correo electrónico (por ejemplo, `sendmail`, `smtp`, etc.) ¿Pero qué si queremos utilizar el servicio cliente de correo en algún otro lugar? Desde luego, no queremos repetir la configuración del gestor de correo *cada* vez que tenemos que utilizar el objeto `Mailer`. ¿Qué pasa si necesitamos cambiar el `transporte` de `sendmail` a `smtp` en todas partes en la aplicación? Necesitaríamos cazar todos los lugares que crean un servicio `Mailer` y modificarlo.

2.26.3 Creando/configurando servicios en el contenedor

Una mejor respuesta es dejar que el contenedor de servicios cree el objeto `Mailer` para ti. Para que esto funcione, debemos *enseñar* al contenedor cómo crear el servicio `Mailer`. Esto se hace a través de configuración, la cual se puede especificar en *YAML*, *XML* o *PHP*:

- *YAML*

  ```
  # app/config/config.yml
  services:
      mi_cartero:
          class:        Acme\HolaBundle\Mailer
          arguments:    [sendmail]
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <services>
      <service id="mi_cartero" class="Acme\HolaBundle\Mailer">
          <argument>sendmail</argument>
      </service>
  </services>
  ```

- *PHP*

  ```
  // app/config/config.php
  use Symfony\Component\DependencyInjection\Definition;

  $contenedor->setDefinition('mi_cartero', new Definition(
  ```

```
        'Acme\HolaBundle\Mailer',
        array('sendmail')
    ));
```

Nota: Cuando se inicia, por omisión *Symfony2* construye el contenedor de servicios usando la configuración de (`app/config/config.yml`). El archivo exacto que se carga es dictado por el método `AppKernel::registerContainerConfiguration()`, el cual carga un archivo de configuración específico al entorno (por ejemplo, `config_dev.yml` para el entorno `dev` o `config_prod.yml` para `prod`).

Una instancia del objeto `Acme\HolaBundle\Mailer` ahora está disponible a través del contenedor de servicios. El contenedor está disponible en cualquier controlador tradicional de *Symfony2*, desde donde puedes acceder al servicio del contenedor a través del método `get()`:

```
class HolaController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $cartero = $this->get('mi_cartero');
        $cartero->send('ryan@foobar.net', ... );
    }
}
```

Cuando pedimos el servicio `mi_cartero` desde el contenedor, el contenedor construye el objeto y lo devuelve. Esta es otra de las principales ventajas de utilizar el contenedor de servicios. Es decir, un servicio *nunca* es construido hasta que es necesario. Si defines un servicio y no lo utilizas en una petición, el servicio no se crea. Esto ahorra memoria y aumenta la velocidad de tu aplicación. Esto también significa que la sanción en rendimiento por definir muchos servicios es muy poca o ninguna. Los servicios que nunca se usan nunca se construyen.

Como bono adicional, el servicio `Mailer` se crea sólo una vez y se vuelve a utilizar la misma instancia cada vez que solicites el servicio. Este casi siempre es el comportamiento que tendrá (el cual es más flexible y potente), pero vamos a aprender más adelante cómo puedes configurar un servicio que tiene varias instancias.

2.26.4 Parámetros del servicio

La creación de nuevos servicios (es decir, objetos) a través del contenedor es bastante sencilla. Los parámetros provocan que al definir los servicios estén más organizados y sean más flexibles:

- *YAML*

```
# app/config/config.yml
parameters:
    mi_cartero.class:      Acme\HolaBundle\Mailer
    mi_cartero.transport:  sendmail

services:
    mi_cartero:
        class:        %mi_cartero.class%
        arguments:    [%mi_cartero.transport%]
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
```

```
    <parameter key="mi_cartero.class">Acme\HolaBundle\Mailer</parameter>
    <parameter key="mi_cartero.transport">sendmail</parameter>
</parameters>

<services>
    <service id="mi_cartero" class="%mi_cartero.class%">
        <argument>%mi_cartero.transport%</argument>
    </service>
</services>
```

- *PHP*

```
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$contenedor->setParameter('mi_cartero.class', 'Acme\HolaBundle\Mailer');
$contenedor->setParameter('mi_cartero.transport', 'sendmail');

$contenedor->setDefinition('mi_cartero', new Definition(
    '%mi_cartero.class%',
    array('%mi_cartero.transport%')
));
```

El resultado final es exactamente igual que antes - la diferencia es sólo en *cómo* definimos el servicio. Al rodear las cadenas `mi_cartero.class` y `mi_cartero.transport` entre signos de porcentaje (`%`), el contenedor sabe que tiene que buscar los parámetros con esos nombres. Cuando se construye el contenedor, este busca el valor de cada parámetro y lo utiliza en la definición del servicio.

El propósito de los parámetros es alimentar información a los servicios. Por supuesto no había nada malo en la definición del servicio sin utilizar ningún parámetro. Los parámetros, sin embargo, tienen varias ventajas:

- Separan y organizan todo el servicio en "opciones" bajo una sola clave `parameters`;
- Los valores del parámetro se pueden utilizar en la definición de múltiples servicios;
- Cuando creas un servicio en un paquete (vamos a mostrar esto en breve), utilizar parámetros permite que el servicio sea fácil de personalizar en tu aplicación.

La opción de usar o no parámetros depende de ti. Los paquetes de alta calidad de terceros *siempre* usan parámetros, ya que producen servicios almacenados en el contenedor más configurables. Para los servicios de tu aplicación, sin embargo, posiblemente no necesites la flexibilidad de los parámetros.

2.26.5 Importando la configuración de recursos desde otros contenedores

---

Truco: En esta sección, nos referiremos a los archivos de configuración de servicios como *recursos*. Se trata de resaltar el hecho de que, si bien la mayoría de la configuración de recursos debe estar en archivos (por ejemplo, *YAML*, *XML*, *PHP*), *Symfony2* es tan flexible que la configuración se puede cargar desde cualquier lugar (por ejemplo, una base de datos e incluso a través de un servicio web externo).

El contenedor de servicios se construye usando un recurso de configuración simple (`app/config/config.yml` por omisión). Toda la configuración de otros servicios (incluido el núcleo de *Symfony2* y la configuración de paquetes de terceros) se debe importar desde el interior de este archivo en una u otra forma. Esto proporciona absoluta flexibilidad sobre los servicios en tu aplicación.

La configuración externa de servicios se puede importar de dos maneras diferentes. En primer lugar, vamos a hablar sobre el método que utilizarás con más frecuencia en tu aplicación: la Directiva `imports`. En la siguiente sección,

vamos a introducir el segundo método, que es el método flexible y preferido para importar la configuración del servicio desde paquetes de terceros.

Importando configuración con `imports`

Hasta ahora, hemos puesto nuestra definición del contenedor del servicio `mi_cartero` directamente en el archivo de configuración de la aplicación (por ejemplo, `app/config/config.yml`). Por supuesto, debido a que la clase `Mailer` vive dentro de `AcmeHolaBundle`, tiene más sentido poner la definición del contenedor de `mi_cartero` en el paquete también.

En primer lugar, mueve la definición del contenedor de `mi_cartero` a un nuevo archivo contenedor de recursos dentro `AcmeHolaBundle`. Si los directorios `Resourses` y `Resourses/config` no existen, créalos.

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    mi_cartero.class:      Acme\HolaBundle\Mailer
    mi_cartero.transport:  sendmail

services:
    mi_cartero:
        class:        %mi_cartero.class%
        arguments:    [%mi_cartero.transport%]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="mi_cartero.class">Acme\HolaBundle\Mailer</parameter>
    <parameter key="mi_cartero.transport">sendmail</parameter>
</parameters>

<services>
    <service id="mi_cartero" class="%mi_cartero.class%">
        <argument>%mi_cartero.transport%</argument>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$contenedor->setParameter('mi_cartero.class', 'Acme\HolaBundle\Mailer');
$contenedor->setParameter('mi_cartero.transport', 'sendmail');

$contenedor->setDefinition('mi_cartero', new Definition(
    '%mi_cartero.class%',
    array('%mi_cartero.transport%')
));
```

La propia definición no ha cambiado, sólo su ubicación. Por supuesto, el contenedor de servicios no sabe sobre el nuevo archivo de recursos. Afortunadamente, es fácil importar el archivo de recursos utilizando la clave `imports` en la configuración de la aplicación.

- *YAML*

```
# app/config/config.yml
imports:
    hola_bundle:
        resource: @AcmeHolaBundle/Resources/config/services.yml
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="@AcmeHolaBundle/Resources/config/services.xml"/>
</imports>
```

- *PHP*

```
// app/config/config.php
$this->import('@AcmeHolaBundle/Resources/config/services.php');
```

La directiva `imports` permite a tu aplicación incluir recursos de configuración del contenedor de servicios de cualquier otro lugar (comúnmente desde paquetes). La ubicación de `resourses`, para archivos, es la ruta absoluta al archivo de recursos. La sintaxis especial `@AcmeHola` resuelve la ruta al directorio del paquete `AcmeHolaBundle`. Esto te ayuda a especificar la ruta a los recursos sin tener que preocuparte más adelante de si se mueve el `AcmeHolaBundle` a un directorio diferente.

Importando configuración vía extensiones del contenedor

Cuando desarrollas en *Symfony2*, comúnmente debes usar la directiva `imports` para importar la configuración del contenedor desde los paquetes que has creado específicamente para tu aplicación. La configuración del paquete contenedor de terceros, incluyendo los servicios básicos de *Symfony2*, normalmente se cargan con cualquier otro método que sea más flexible y fácil de configurar en tu aplicación.

Así es como funciona. Internamente, cada paquete define sus servicios muy parecido a lo que hemos visto hasta ahora. Es decir, un paquete utiliza uno o más archivos de configuración de recursos (por lo general XML) para especificar los parámetros y servicios para ese paquete. Sin embargo, en lugar de importar cada uno de estos recursos directamente desde la configuración de tu aplicación utilizando la directiva `imports`, sólo tienes que invocar una *extensión contenedora de servicios* dentro del paquete, la cual hace el trabajo por ti. Una extensión del contenedor de servicios es una clase *PHP* creada por el autor del paquete para lograr dos cosas:

- Importar todos los recursos del contenedor de servicios necesarios para configurar los servicios del paquete;
- permite una configuración semántica y directa para poder configurar el paquete sin interactuar con los parámetros de configuración planos del paquete contenedor del servicio.

En otras palabras, una extensión del contenedor de servicios configura los servicios para un paquete en tu nombre. Y como veremos en un momento, la extensión proporciona una interfaz sensible y de alto nivel para configurar el paquete.

Tomemos el `FrameworkBundle` - el núcleo de la plataforma *Symfony2* - como ejemplo. La presencia del siguiente código en la configuración de tu aplicación invoca a la extensión en el interior del contenedor de servicios `FrameworkBundle`:

- *YAML*

```
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    charset:         UTF-8
    form:            true
    csrf_protection: true
```

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

- *XML*

```
<!-- app/config/config.xml -->
<framework:config charset="UTF-8" secret="xxxxxxxxxx">
    <framework:form />
    <framework:csrf-protection />
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <!-- ... -->
</framework>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('framework', array(
    'secret'          => 'xxxxxxxxxx',
    'charset'         => 'UTF-8',
    'form'            => array(),
    'csrf-protection' => array(),
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
    // ...
));
```

Cuando se analiza la configuración, el contenedor busca una extensión que pueda manejar la directiva de configuración `framework`. La extensión en cuestión, que vive en el `FrameworkBundle`, es invocada y cargada la configuración del servicio para el `FrameworkBundle`. Si quitas la clave `framework` del archivo de configuración de tu aplicación por completo, no se cargarán los servicios básicos de *Symfony2*. El punto es que tú tienes el control: la plataforma *Symfony2* no contiene ningún tipo de magia o realiza cualquier acción en que tú no tengas el control.

Por supuesto que puedes hacer mucho más que simplemente "activar" la extensión del contenedor de servicios del `FrameworkBundle`. Cada extensión te permite personalizar fácilmente el paquete, sin tener que preocuparte acerca de cómo se definen los servicios internos.

En este caso, la extensión te permite personalizar el juego de caracteres - `charset`, gestor de errores - `error_handler`, protección CSRF - `csrf_protection`, configuración del ruteador - `router` - y mucho más. Internamente, el `FrameworkBundle` utiliza las opciones especificadas aquí para definir y configurar los servicios específicos del mismo. El paquete se encarga de crear todos los `parámetros` y `servicios` necesarios para el contenedor de servicios, mientras permite que la mayor parte de la configuración se pueda personalizar fácilmente. Como bono adicional, la mayoría de las extensiones del contenedor de servicios también son lo suficientemente inteligentes como para realizar la validación - notificándote opciones omitidas o datos de tipo incorrecto.

Al instalar o configurar un paquete, consulta la documentación del paquete de cómo se deben instalar y configurar los servicios para el paquete. Las opciones disponibles para los paquetes básicos se pueden encontrar dentro de la *Guía de Referencia* (Página 443).

Nota: De forma nativa, el contenedor de servicios sólo reconoce las directivas `parameters`, `services` e `imports`. Cualquier otra directiva es manejada por una extensión del contenedor de servicios.

2.26.6 Refiriendo (inyectando) servicios

Hasta el momento, nuestro servicio original `mi_cartero` es simple: sólo toma un argumento en su constructor, el cual es fácilmente configurable. Como verás, el poder real del contenedor se realiza cuando es necesario crear un servicio que depende de uno o varios otros servicios en el contenedor.

Comencemos con un ejemplo. Supongamos que tenemos un nuevo servicio, `BoletinGestor`, que ayuda a gestionar la preparación y entrega de un mensaje de correo electrónico a una colección de direcciones. Por supuesto el servicio `mi_cartero` ya es realmente bueno en la entrega de mensajes de correo electrónico, así que lo usaremos dentro de `BoletinGestor` para manejar la entrega real de los mensajes. Se pretende que esta clase pudiera ser algo como esto:

```
namespace Acme\HolaBundle\Boletin;

use Acme\HolaBundle\Mailer;

class BoletinGestor
{
    protected $cartero;

    public function __construct(Mailer $cartero)
    {
        $this->cartero = $cartero;
    }

    // ...
}
```

Sin utilizar el contenedor de servicios, podemos crear un nuevo `BoletinGestor` muy fácilmente desde el interior de un controlador:

```
public function sendNewsletterAction()
{
    $cartero = $this->get('mi_cartero');
    $boletin = new Acme\HolaBundle\Boletin\BoletinGestor($cartero);
    // ...
}
```

Este enfoque está bien, pero, ¿si más adelante decidimos que la clase `BoletinGestor` necesita un segundo o tercer argumento constructor? ¿Y si nos decidimos a reconstruir nuestro código y cambiar el nombre de la clase? En ambos casos, habría que encontrar todos los lugares donde se crea una instancia de `BoletinGestor` y modificarla. Por supuesto, el contenedor de servicios nos da una opción mucho más atractiva:

- *YAML*

  ```
  # src/Acme/HolaBundle/Resources/config/services.yml
  parameters:
      # ...
      boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor

  services:
      mi_cartero:
          # ...
      boletin_gestor:
          class:    %boletin_gestor.class%
          arguments: [@mi_cartero]
  ```

- *XML*

  ```
  <!-- src/Acme/HolaBundle/Resources/config/services.xml -->
  <parameters>
      <!-- ... -->
      <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
  </parameters>

  <services>
  ```

```
        <service id="mi_cartero" ... >
          <!-- ... -->
        </service>
        <service id="boletin_gestor" class="%boletin_gestor.class%">
            <argument type="service" id="mi_cartero"/>
        </service>
    </services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%',
    array(new Reference('mi_cartero'))
));
```

En YAML, la sintaxis especial `@mi_cartero` le dice al contenedor que busque un servicio llamado `mi_cartero` y pase ese objeto al constructor de `BoletinGestor`. En este caso, sin embargo, el servicio especificado `mi_cartero` debe existir. Si no es así, lanzará una excepción. Puedes marcar tus dependencias como opcionales - explicaremos esto en la siguiente sección.

La utilización de referencias es una herramienta muy poderosa que te permite crear clases de servicios independientes con dependencias bien definidas. En este ejemplo, el servicio `boletin_gestor` necesita del servicio `mi_cartero` para poder funcionar. Al definir esta dependencia en el contenedor de servicios, el contenedor se encarga de todo el trabajo de crear instancias de objetos.

Dependencias opcionales: Inyección de definidores

Inyectar dependencias en el constructor de esta manera es una excelente manera de asegurarte que la dependencia está disponible para usarla. Si tienes dependencias opcionales para una clase, entonces, la "inyección de definidor" puede ser una mejor opción. Esto significa inyectar la dependencia usando una llamada a método en lugar de a través del constructor. La clase se vería así:

```
namespace Acme\HolaBundle\Boletin;

use Acme\HolaBundle\Mailer;

class BoletinGestor
{
    protected $cartero;

    public function setCartero(Mailer $cartero)
    {
        $this->cartero = $cartero;
    }

    // ...
}
```

La inyección de la dependencia por medio del método definidor sólo necesita un cambio de sintaxis:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor

services:
    mi_cartero:
        # ...
    boletin_gestor:
        class:    %boletin_gestor.class%
        calls:
            - [ setCartero, [ @mi_cartero ] ]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
</parameters>

<services>
    <service id="mi_cartero" ... >
      <!-- ... -->
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%">
        <call method="setCartero">
             <argument type="service" id="mi_cartero" />
        </call>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%'
))->addMethodCall('setCartero', array(
    new Reference('mi_cartero')
));
```

Nota: Los enfoques presentados en esta sección se llaman "inyección de constructor" e "inyección de definidor". El contenedor de servicios de *Symfony2* también es compatible con la "inyección de propiedad".

2.26.7 Haciendo referencias opcionales

A veces, uno de tus servicios puede tener una dependencia opcional, lo cual significa que la dependencia no es necesaria para que el servicio funcione correctamente. En el ejemplo anterior, el servicio `mi_cartero` *debe* existir, si no,

será lanzada una excepción. Al modificar la definición del servicio `boletin_gestor`, puedes hacer opcional esta referencia. Entonces, el contenedor será inyectado si es que existe y no hace nada si no:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...

services:
    boletin_gestor:
        class:    %boletin_gestor.class%
        arguments: [@?mi_cartero]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->

<services>
    <service id="mi_cartero" ... >
      <!-- ... -->
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%">
        <argument type="service" id="mi_cartero" on-invalid="ignore" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\ContainerInterface;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%',
    array(new Reference('mi_cartero', ContainerInterface::IGNORE_ON_INVALID_REFERENCE))
));
```

En YAML, la sintaxis especial `@?` le dice al contenedor de servicios que la dependencia es opcional. Por supuesto, `BoletinGestor` también se debe escribir para permitir una dependencia opcional:

```
public function __construct(Mailer $cartero = null)
{
    // ...
}
```

2.26.8 El núcleo de *Symfony* y servicios en un paquete de terceros

Puesto que *Symfony2* y todos los paquetes de terceros configuran y recuperan sus servicios a través del contenedor, puedes acceder fácilmente a ellos e incluso utilizarlos en tus propios servicios. Para mantener las cosas simples, de manera predeterminada *Symfony2* no requiere que los controladores se definan como servicios. Además *Symfony2* inyecta el contenedor de servicios completo en el controlador. Por ejemplo, para manejar el almacenamiento de información sobre la sesión de un usuario, *Symfony2* proporciona un servicio `sesión`, al cual se puede acceder dentro de

un controlador estándar de la siguiente manera:

```
public function indexAction($bar)
{
    $sesion = $this->get('sesion');
    $sesion->set('foo', $bar);

    // ...
}
```

En *Symfony2*, constantemente vas a utilizar los servicios prestados por el núcleo de *Symfony* o paquetes de terceros para realizar tareas como la reproducción de plantillas (`templating`), el envío de mensajes de correo electrónico (`mailer`), o para acceder a información sobre la petición.

Podemos dar un paso más allá usando estos servicios dentro de los servicios que has creado para tu aplicación. Vamos a modificar el `BoletinGestor` para usar el gestor de correo real de *Symfony2*, el servicio `mailer` (en vez del pretendido `mi_cartero`). También vamos a pasar el servicio del motor de plantillas al `BoletinGestor` para que puedas generar el contenido del correo electrónico a través de una plantilla:

```
namespace Acme\HolaBundle\Boletin;

use Symfony\Component\Templating\EngineInterface;

class BoletinGestor
{
    protected $cartero;

    protected $plantilla;

    public function __construct(\Swift_Mailer $cartero, EngineInterface $plantilla)
    {
        $this->cartero = $cartero;
        $this->plantilla = $plantilla;
    }

    // ...
}
```

Configurar el contenedor de servicios es fácil:

- *YAML*

  ```
  services:
      boletin_gestor:
          class:    %boletin_gestor.class%
          arguments: [@mailer, @templating]
  ```

- *XML*

  ```
  <service id="boletin_gestor" class="%boletin_gestor.class%">
      <argument type="service" id="mailer"/>
      <argument type="service" id="templating"/>
  </service>
  ```

- *PHP*

  ```
  $contenedor->setDefinition('boletin_gestor', new Definition(
      '%boletin_gestor.class%',
      array(
          new Reference('mailer'),
  ```

```
            new Reference('templating')
        )
    ));
```

El servicio `boletin_gestor` ahora tiene acceso a los servicios del núcleo `mailer` y `templating`. Esta es una forma común de crear servicios específicos para tu aplicación que aprovechan el poder de los distintos servicios en la plataforma.

Truco: Asegúrate de que la entrada `SwiftMailer` aparece en la configuración de la aplicación. Como mencionamos en *Importando configuración vía extensiones del contenedor* (Página 242), la clave `SwiftMailer` invoca a la extensión de servicio desde `SwiftmailerBundle`, la cual registra el servicio `mailer`.

2.26.9 Configuración avanzada del contenedor

Como hemos visto, definir servicios dentro del contenedor es fácil, generalmente implica una clave de configuración `service` y algunos parámetros. Sin embargo, el contenedor tiene varias otras herramientas disponibles que ayudan a *etiquetar* servicios por funcionalidad especial, crear servicios más complejos y realizar operaciones después de que el contenedor está construido.

Marcando servicios como público / privado

Cuando definas servicios, generalmente, querrás poder acceder a estas definiciones dentro del código de tu aplicación. Estos servicios se llaman `public`. Por ejemplo, el servicio `doctrine` registrado en el contenedor cuando se utiliza `DoctrineBundle` es un servicio público al que puedes acceder a través de:

```
$doctrine = $contenedor->get('doctrine');
```

Sin embargo, hay casos de uso cuando no quieres que un servicio sea público. Esto es común cuando sólo se define un servicio, ya que se podría utilizar como argumento para otro servicio.

Nota: Si utilizas un servicio privado como argumento a más de otro servicio, esto se traducirá en dos diferentes instancias utilizadas como la creación del servicio privado realizada en línea (por ejemplo, `new PrivateFooBar()`).

Dice simplemente: El servicio será privado cuando no deseas acceder a él directamente desde tu código.

Aquí está un ejemplo:

- *YAML*

  ```
  services:
     foo:
       class: Acme\HolaBundle\Foo
       public: false
  ```

- *XML*

  ```
  <service id="foo" class="Acme\HolaBundle\Foo" public="false" />
  ```

- *PHP*

  ```
  $definicion = new Definition('Acme\HolaBundle\Foo');
  $definicion->setPublic(false);
  $contenedor->setDefinition('foo', $definicion);
  ```

Ahora que el servicio es privado, *no* puedes llamar a:

```
$contenedor->get('foo');
```

Sin embargo, si has marcado un servicio como privado, todavía puedes asignarle un alias (ve más abajo) para acceder a este servicio (a través del alias).

Nota: Los servicios por omisión son públicos.

Rebautizando

Cuando utilizas el núcleo o paquetes de terceros dentro de tu aplicación, posiblemente desees utilizar métodos abreviados para acceder a algunos servicios. Puedes hacerlo rebautizándolos y, además, puedes incluso rebautizar servicios no públicos.

- *YAML*

```
services:
   foo:
     class: Acme\HolaBundle\Foo
   bar:
     alias: foo
```

- *XML*

```
<service id="foo" class="Acme\HolaBundle\Foo"/>

<service id="bar" alias="foo" />
```

- *PHP*

```
$definicion = new Definition('Acme\HolaBundle\Foo');
$contenedor->setDefinition('foo', $definicion);

$containerBuilder->setAlias('bar', 'foo');
```

Esto significa que cuando utilizas el contenedor directamente, puedes acceder al servicio `foo` al pedir el servicio `bar` así:

```
$contenedor->get('bar'); // podrías devolver el servicio foo
```

Incluyendo archivos

Puede haber casos de uso cuando necesites incluir otro archivo justo antes de cargar el servicio en sí. Para ello, puedes utilizar la directiva `file`.

- *YAML*

```
services:
   foo:
     class: Acme\HolaBundle\Foo\Bar
     file: %kernel.root_dir%/src/ruta/al/archivo/foo.php
```

- *XML*

```
<service id="foo" class="Acme\HolaBundle\Foo\Bar">
    <file>%kernel.root_dir%/src/rita/al/archivo/foo.php</file>
</service>
```

- *PHP*

```
$definicion = new Definition('Acme\HolaBundle\Foo\Bar');
$definicion->setFile('%kernel.root_dir%/src/ruta/al/archivo/foo.php');
$contenedor->setDefinition('foo', $definicion);
```

Ten en cuenta que internamente *Symfony* llama a la función *PHP* `require_once`, lo cual significa que el archivo se incluirá una sola vez por petición.

Etiquetas (`tags`)

De la misma manera que en la Web una entrada de blog se puede etiquetar con cosas tales como *"Symfony"* o *"PHP"*, los servicios configurados en el contenedor también se pueden etiquetar. En el contenedor de servicios, una etiqueta implica que el servicio está destinado a usarse para un propósito específico. Tomemos el siguiente ejemplo:

- *YAML*

```
services:
    foo.twig.extension:
        class: Acme\HolaBundle\Extension\FooExtension
        tags:
            -  { name: twig.extension }
```

- *XML*

```
<service id="foo.twig.extension" class="Acme\HolaBundle\Extension\FooExtension">
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
$definicion = new Definition('Acme\HolaBundle\Extension\FooExtension');
$definicion->addTag('twig.extension');
$contenedor->setDefinition('foo.twig.extension', $definicion);
```

La etiqueta `twig.extension` es una etiqueta especial que `TwigBundle` usa durante la configuración. Al dar al servicio esta etiqueta `twig.extension`, el paquete sabe que el servicio `foo.twig.extension` se debe registrar como una extensión *Twig* con *Twig*. En otras palabras, *Twig* encuentra todos los servicios con la etiqueta `twig.extension` y automáticamente los registra como extensiones.

Las etiquetas, entonces, son una manera de decirle a *Symfony2* u otros paquetes de terceros que tu servicio se debe registrar o utilizar de alguna forma especial por el paquete.

La siguiente es una lista de etiquetas disponibles con los paquetes del núcleo de *Symfony2*. Cada una de ellas tiene un efecto diferente en tu servicio y muchas etiquetas requieren argumentos adicionales (más allá de sólo el parámetro `nombre`).

- assetic.filter
- assetic.templating.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

2.26.10 Aprende más en el recetario

- *Cómo utilizar el patrón fábrica para crear servicios* (Página 331)
- *Cómo gestionar dependencias comunes con servicios padre* (Página 335)
- *Cómo definir controladores como servicios* (Página 278)

2.27 Rendimiento

Symfony2 es rápido, desde que lo sacas de la caja. Por supuesto, si realmente necesitas velocidad, hay muchas maneras en las cuales puedes hacer que *Symfony* sea aún más rápido. En este capítulo, podrás explorar muchas de las formas más comunes y potentes para hacer que tu aplicación *Symfony* sea aún más rápida.

2.27.1 Utilizando una caché de código de bytes (p. ej. APC)

Una de las mejores (y más fáciles) cosas que debes hacer para mejorar tu rendimiento es utilizar una "caché de código de bytes". La idea de una caché de código de bytes es eliminar la necesidad de constantemente tener que volver a compilar el código fuente PHP. Hay disponible una serie de cachés de código de bytes, algunas de las cuales son de código abierto. Probablemente, la caché de código de bytes más utilizada sea APC

Usar una caché de código de bytes realmente no tiene ningún inconveniente, y *Symfony2* se ha diseñado para llevar a cabo muy bien en este tipo de entorno.

Optimización adicional

La caché de código de bytes, por lo general, comprueba los cambios de los archivos fuente. Esto asegura que si cambia un archivo fuente, el código de bytes se vuelve a compilar automáticamente. Esto es muy conveniente, pero, obviamente, implica una sobrecarga.

Por esta razón, algunas cachés de código de bytes ofrecen una opción para desactivar esa comprobación. Obviamente, cuando desactivas esta comprobación, será responsabilidad del administrador del servidor asegurarse de que la caché se borra cada vez que cambia un archivo fuente. De lo contrario, no se verán los cambios realizados.

Por ejemplo, para desactivar estos controles en APC, sólo tienes que añadir la opción `apc.stat=0` en tu archivo de configuración `php.ini`.

2.27.2 Usa un autocargador con caché (p. ej. `ApcUniversalClassLoader`)

De manera predeterminada, la edición estándar de *Symfony2* utiliza el `UniversalClassLoader` del archivo autoloader.php. Este autocargador es fácil de usar, ya que automáticamente encontrará cualquier nueva clase que hayas colocado en los directorios registrados.

Desafortunadamente, esto tiene un costo, puesto que el cargador itera en todos los espacios de nombres configurados para encontrar un archivo, haciendo llamadas a `file_exists` hasta que finalmente encuentra el archivo que está buscando.

La solución más sencilla es que la caché memorice la ubicación de cada clase después de encontrarla por primera vez. *Symfony* incluye una clase cargadora - `ApcUniversalClassLoader` - que extiende al `UniversalClassLoader` y almacena la ubicación de las clases en APC.

Para utilizar este cargador de clases, sólo tienes que adaptar tu `autoloader.php` como sigue:

```
// app/autoload.php
require __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$cargador = new ApcUniversalClassLoader('algún prefijo de caché único');
// ...
```

Nota: Al utilizar el cargador *APC* automático, si agregas nuevas clases, las encontrará automáticamente y todo funcionará igual que antes (es decir, no hay razón para "limpiar" la caché). Sin embargo, si cambias la ubicación de un determinado espacio de nombres o prefijo, tendrás que limpiar tu caché *APC*. De lo contrario, el cargador aún buscará en la ubicación anterior todas las clases dentro de ese espacio de nombres.

2.27.3 Utilizando archivos de arranque

Para garantizar una óptima flexibilidad y reutilización de código, las aplicaciones de *Symfony2* aprovechan una variedad de clases y componentes de terceros. Pero cargar todas estas clases desde archivos separados en cada petición puede dar lugar a alguna sobrecarga. Para reducir esta sobrecarga, la edición estándar de *Symfony2* proporciona un guión para generar lo que se conoce como archivo de arranque, el cual contiene la definición de múltiples clases en un solo archivo. Al incluir este archivo (el cual contiene una copia de muchas de las clases del núcleo), *Symfony* ya no tiene que incluir algunos de los archivos de código fuente que contienen las clases. Esto reducirá bastante la E/S del disco.

Si estás utilizando la edición estándar de *Symfony2*, entonces probablemente ya estás utilizando el archivo de arranque. Para estar seguro, abre el controlador frontal (por lo general `app.php`) y asegúrate de que existe la siguiente línea:

```
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Ten en cuenta que hay dos desventajas cuando utilizas un archivo de arranque:

- El archivo se tiene que regenerar cada vez que cambia alguna de las fuentes original (es decir, cuando actualizas el código fuente de *Symfony2* o de las bibliotecas de proveedores);
- En la depuración, será necesario colocar puntos de interrupción dentro del archivo de arranque.

Si estás utilizando la Edición estándar de *Symfony2*, los archivos de arranque se reconstruyen automáticamente después de actualizar las bibliotecas de proveedores a través de la orden `php bin/vendors install`.

Archivos de arranque y caché de código de bytes

Incluso cuando utilizas código de bytes en caché, el rendimiento mejorará cuando utilices un archivo de arranque ya que habrá menos archivos en los cuales supervisar los cambios. Por supuesto, si esta función está desactivada en la caché del código de bytes (por ejemplo, `apc.stat = 0` en APC), ya no hay una razón para utilizar un archivo de arranque.

2.28 Funcionamiento interno

Parece que quieres entender cómo funciona y cómo extender *Symfony2*. ¡Eso me hace muy feliz! Esta sección es una explicación en profundidad de *Symfony2* desde dentro.

Nota: Necesitas leer esta sección sólo si quieres entender cómo funciona *Symfony2* detrás de la escena, o si deseas ampliar *Symfony2*.

2.28.1 Resumen

El código *Symfony2* está hecho de varias capas independientes. Cada capa está construida en lo alto de la anterior.

Truco: La carga automática no es administrada directamente por la plataforma; sino que se hace independientemente con la ayuda de la clase `Symfony\Component\ClassLoader\UniversalClassLoader` y el archivo `src/autoload.php`. Lee el *capítulo dedicado* (Página 407) para más información.

Componente `HttpFoundation`

Al nivel más profundo está el componente **:namespace:'Symfony\\Component\\HttpFoundation'**. `HttpFoundation` proporciona los principales objetos necesarios para hacer frente a *HTTP*. Es una abstracción orientada a objetos de algunas funciones y variables nativas de *PHP*:

- La clase `Symfony\Component\HttpFoundation\Request` resume las principales variables globales de *PHP* como `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES` y `$_SERVER`;
- La clase `Symfony\Component\HttpFoundation\Response` resume algunas funciones *PHP* como `header()`, `setcookie()` y `echo`;
- La clase `Symfony\Component\HttpFoundation\Session` y la interfaz `Symfony\Component\HttpFoundation\SessionStorage\SessionStorageInterface`, resumen la gestión de sesión y las funciones `session_*()`.

Componente `HttpKernel`

En lo alto de `HttpFoundation` está el componente **:namespace:'Symfony\\Component\\HttpKernel'**. `HttpKernel` se encarga de la parte dinámica de *HTTP*, es una fina capa en la parte superior de las clases `Petición` y `Respuesta` para estandarizar la forma en que se manejan las peticiones. Esta también proporciona puntos de extensión y herramientas que lo convierten en el punto de partida ideal para crear una plataforma Web sin demasiado trabajo.

Además, opcionalmente añade configurabilidad y extensibilidad, gracias al componente de inyección de dependencias y un potente sistema de complementos (paquetes).

Ver También:

Lee más sobre el componente `HttpKernel`. Lee más sobre la *Inyección de dependencias* (Página 237) y los *Paquetes* (Página 346).

Paquete `FrameworkBundle`

El paquete **:namespace:'Symfony\\Bundle\\FrameworkBundle'** es el paquete que une los principales componentes y bibliotecas para hacer una plataforma MVC ligera y rápida. Este viene con una configuración predeterminada sensible y convenios para facilitar la curva de aprendizaje.

2.28.2 Kernel

La clase `Symfony\Component\HttpKernel\HttpKernel` es la clase central de *Symfony2* y es responsable de tramitar las peticiones del cliente. Su objetivo principal es "convertir" un objeto `Symfony\Component\HttpFoundation\Request` a un objeto `Symfony\Component\HttpFoundation\Response`.

Cada núcleo de *Symfony2* implementa `Symfony\Component\HttpKernel\HttpKernelInterface`:

```
function handle(Request $peticion, $type = self::MASTER_REQUEST, $catch = true)
```

Controladores

Para convertir una `Petición` a una `Respuesta`, el `Kernel` cuenta con un "Controlador". Un controlador puede ser cualquier PHP ejecutable válido.

El `Kernel` delega la selección de cual controlador se debe ejecutar a una implementación de `Symfony\Component\HttpKernel\Controller\ControllerResolverInterface`:

```
public function getController(Request $peticion);

public function getArguments(Request $peticion, $controller);
```

El método **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterface::getController'** devuelve el controlador (un PHP ejecutable) asociado a la petición dada. La implementación predeterminada de (`Symfony\Component\HttpKernel\Controller\ControllerResolver`) busca un atributo `_controller` en la petición que representa el nombre del controlador (una cadena "clase::método", cómo `Bundle\BlogBundle\PostController:indexAction`).

Truco: La implementación predeterminada utiliza la clase `Symfony\Bundle\FrameworkBundle\EventListener\RouterI` para definir el atributo `_controller` de la petición (consulta *kernel-core_request*).

El método **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterface::getArguments'** devuelve una matriz de argumentos para pasarla al Controlador ejecutable. La implementación predeterminada automáticamente resuelve los argumentos del método, basándose en los atributos de la Petición.

Coincidiendo los argumentos del método Controlador desde los atributos de la `Petición`

Por cada argumento del método, *Symfony2* trata de obtener el valor de un atributo de la petición con el mismo nombre. Si no está definido, el valor predeterminado es el argumento utilizado de estar definido:

```
// Symfony2 debe buscar un atributo
// 'id' (obligatorio) y un
// 'admin' (opcional)
public function showAction($id, $admin = true)
{
    // ...
}
```

Manejando peticiones

El método `handle()` toma una `Petición` y *siempre* devuelve una `Respuesta`. Para convertir la `Petición`, `handle()` confía en el mecanismo de resolución y una cadena ordenada de notificaciones de evento (consulta la siguiente sección para más información acerca de cada evento):

1. Antes de hacer cualquier otra cosa, difunde el evento `kernel.request` - si alguno de los escuchas devuelve una `Respuesta`, salta directamente al paso 8;
2. El mecanismo de resolución es llamado para determinar el controlador a ejecutar;
3. Los escuchas del evento `kernel.controller` ahora pueden manipular el controlador ejecutable como quieras (cambiarlo, envolverlo, ...);
4. El núcleo verifica que el controlador en realidad es un PHP ejecutable válido;
5. Se llama al mecanismo de resolución para determinar los argumentos a pasar al controlador;
6. El `Kernel` llama al controlador;
7. Si el controlador no devuelve una `Respuesta`, los escuchas del evento `kernel.view` pueden convertir en `Respuesta` el valor devuelto por el Controlador;
8. Los escuchas del evento `kernel.response` pueden manipular la `Respuesta` (contenido y cabeceras);
9. Devuelve la respuesta.

Si se produce una Excepción durante el procesamiento, difunde la `kernel.exception` y se da la oportunidad a los escuchas de convertir la excepción en una `Respuesta`. Si esto funciona, se difunde el evento `kernel.response`, si no, se vuelve a lanzar la Excepción.

Si no deseas que se capturen las Excepciones (para peticiones incrustadas, por ejemplo), desactiva el evento `kernel.exception` pasando `false` como tercer argumento del método `handle()`.

Funcionamiento interno de las peticiones

En cualquier momento durante el manejo de una petición (la 'maestra' uno), puede manejar una subpetición. Puedes pasar el tipo de petición al método `handle()` (su segundo argumento):

- `HttpKernelInterface::MASTER_REQUEST`;
- `HttpKernelInterface::SUB_REQUEST`.

El tipo se pasa a todos los eventos y los escuchas pueden actuar en consecuencia (algún procesamiento sólo debe ocurrir en la petición maestra).

Eventos

Cada evento lanzado por el `Kernel` es una subclase de `Symfony\Component\HttpKernel\Event\KernelEvent`. Esto significa que cada evento tiene acceso a la misma información básica:

- `getRequestType()` - devuelve el *tipo* de la petición (`HttpKernelInterface::MASTER_REQUEST` o `HttpKernelInterface::SUB_REQUEST`);
- `getKernel()` - devuelve el `Kernel` que está manejando la petición;
- `getRequest()` - devuelve la `Petición` que se está manejando actualmente.

`getRequestType()`

El método `getRequestType()` permite a los escuchas conocer el tipo de la petición. Por ejemplo, si un escucha sólo debe estar atento a las peticiones maestras, agrega el siguiente código al principio de tu método escucha:

```
use Symfony\Component\HttpKernel\HttpKernelInterface;

if (HttpKernelInterface::MASTER_REQUEST !== $evento->getRequestType()) {
    // regresa inmediatamente
    return;
}
```

Truco: Si todavía no estás familiarizado con el Despachador de Eventos de *Symfony2*, primero lee la sección *Eventos* (Página 259).

Evento `kernel.request`

Clase del evento: `Symfony\Component\HttpKernel\Event\GetResponseEvent`

El objetivo de este evento es devolver inmediatamente un objeto `Respuesta` o variables de configuración para poder invocar un controlador después del evento. Cualquier escucha puede devolver un objeto `Respuesta` a través del método `setResponse()` en el evento. En este caso, todos los otros escuchas no serán llamados.

Este evento lo utiliza el `FrameworkBundle` para llenar el atributo `_controller` de la `Petición`, a través de `Symfony\Bundle\FrameworkBundle\EventListener\RouterListener`. `RequestListener` usa un objeto `Symfony\Component\Routing\RouterInterface` para coincidir la `Petición` y determinar el nombre del controlador (guardado en el atributo `_controller` de la `Petición`).

Evento `kernel.controller`

Clase del evento: `Symfony\Component\HttpKernel\Event\FilterControllerEvent`

Este evento no lo utiliza `FrameworkBundle`, pero puede ser un punto de entrada para modificar el controlador que se debe ejecutar:

```
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $evento)
{
    $controller = $evento->getController();
    // ...
```

```
    // el controlador se puede cambiar a cualquier PHP ejecutable
    $evento->setController($controller);
}
```

Evento `kernel.view`

Clase del evento: `Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent`

Este evento no lo utiliza el `FrameworkBundle`, pero se puede usar para implementar un subsistema de vistas. Este evento se llama *sólo* si el controlador *no* devuelve un objeto `Respuesta`. El propósito del evento es permitir que algún otro valor de retorno se convierta en una `Respuesta`.

El valor devuelto por el controlador es accesible a través del método `getControllerResult`:

```
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $evento)
{
    $val = $evento->getReturnValue();
    $respuesta = new Response();
    // modifica de alguna manera el valor de retorno de la respuesta

    $evento->setResponse($respuesta);
}
```

Evento `kernel.response`

Clase del evento: `Symfony\Component\HttpKernel\Event\FilterResponseEvent`

El propósito de este evento es permitir que otros sistemas modifiquen o sustituyan el objeto `Respuesta` después de su creación:

```
public function onKernelResponse(FilterResponseEvent $evento)
{
    $respuesta = $evento->getResponse();
    // .. modifica el objeto Respuesta
}
```

El `FrameworkBundle` registra varios escuchas:

- `Symfony\Component\HttpKernel\EventListener\ProfilerListener`: recoge los datos de la petición actual;
- `Symfony\Bundle\WebProfilerBundle\EventListener\WebDebugToolbarListener`: inyecta la barra de herramientas de depuración web;
- `Symfony\Component\HttpKernel\EventListener\ResponseListener`: fija el `Content-Type` de la respuesta basándose en el formato de la petición;
- `Symfony\Component\HttpKernel\EventListener\EsiListener`: agrega una cabecera HTTP `Surrogate-Control` cuando la respuesta necesita ser analizada por etiquetas ESI.

Evento `kernel.exception`

Clase del evento: `Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent`

`FrameworkBundle` registra un `Symfony\Component\HttpKernel\EventListener\ExceptionListener` el cual remite la `Petición` a un determinado controlador (el valor del parámetro `exception_listener.controller` – debe estar en notación `clase::método`).

Un escucha en este evento puede crear y establecer un objeto `Respuesta`, crear y establecer un nuevo objeto `Excepción`, o simplemente no hacer nada:

```
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $evento)
{
    $exception = $evento->getException();
    $respuesta = new Response();
    // configura el objeto Respuesta basándose en la excepción capturada
    $evento->setResponse($respuesta);

    // alternativamente puedes establecer una nueva excepción
    // $exception = new \Exception('Alguna excepción especial');
    // $evento->setException($exception);
}
```

2.28.3 El despachador de eventos

El paradigma orientado a objetos ha recorrido un largo camino para garantizar la extensibilidad del código. Al crear clases que tienen responsabilidades bien definidas, el código se vuelve más flexible y un desarrollador lo puede extender con subclases para modificar su comportamiento. Pero si quieres compartir tus cambios con otros desarrolladores que también han hecho sus propias subclases, la herencia de código es discutible.

Consideremos un ejemplo del mundo real en el que deseas proporcionar un sistema de complementos a tu proyecto. Un complemento debe ser capaz de agregar métodos, o hacer algo antes o después de ejecutar un método, sin interferir con otros complementos. Esto no es un problema fácil de resolver con la herencia simple y herencia múltiple (si fuera posible con *PHP*) tiene sus propios inconvenientes.

El despachador de eventos de *Symfony2* implementa el patrón observador en una manera sencilla y efectiva para hacer todo esto posible y para hacer realmente extensibles tus proyectos.

Tomemos un ejemplo simple desde el componente HttpKernel de Symfony2. Una vez creado un objeto `Respuesta`, puede ser útil permitir que otros elementos en el sistema lo modifiquen (por ejemplo, añadan algunas cabeceras caché) antes de utilizarlo realmente. Para hacer esto posible, el núcleo de *Symfony2* lanza un evento - `kernel.response`. He aquí cómo funciona:

- Un *escucha* (objeto PHP) le dice a un objeto *despachador* central que quiere escuchar el evento `kernel.response`;
- En algún momento, el núcleo de *Symfony2* dice al objeto *despachador* que difunda el evento `kernel.response`, pasando con este un objeto `Evento` que tiene acceso al objeto `Respuesta`;
- El despachador notifica a (es decir, llama a un método en) todos los escuchas del evento `kernel.response`, permitiendo que cada uno de ellos haga alguna modificación al objeto `Respuesta`.

Eventos

Cuando se envía un evento, es identificado por un nombre único (por ejemplo, `kernel.response`), al que cualquier cantidad de escuchas podría estar atento. También se crea una instancia de `Symfony\Component\EventDispatcher\Event` y se pasa a todos los escuchas. Como veremos más adelante, el objeto `Evento` mismo, a menudo contiene datos sobre cuando se despachó el evento.

Convenciones de nomenclatura

El nombre único del evento puede ser cualquier cadena, pero opcionalmente sigue una serie de convenciones de nomenclatura simples:

- Usa sólo letras minúsculas, números, puntos (`.`) y subrayados (_);
- Prefija los nombres con un espacio de nombres seguido de un punto (por ejemplo, `kernel.`);
- Termina los nombres con un verbo que indica qué acción se está tomando (por ejemplo, `petición`).

Estos son algunos ejemplos de nombres de evento aceptables:

- `kernel.response`
- `form.pre_set_data`

Nombres de evento y objetos evento

Cuando el despachador notifica a los escuchas, este pasa un objeto `Evento` real a los escuchas. La clase base `Evento` es muy simple: contiene un método para detener la *propagación de eventos* (Página 265), pero no mucho más.

Muchas veces, los datos acerca de un evento específico se tienen que pasar junto con el objeto `Evento` para que los escuchas tengan la información necesaria. En el caso del evento `kernel.response`, el objeto `Evento` creado y pasado a cada escucha realmente es de tipo `Symfony\Component\HttpKernel\Event\FilterResponseEvent`, una subclase del objeto `Evento` base. Esta clase contiene métodos como `getResponse` y `setResponse`, que permiten a los escuchas recibir e incluso sustituir el objeto `Respuesta`.

La moraleja de la historia es esta: cuando creas un escucha para un evento, el objeto `Evento` que se pasa al escucha puede ser una subclase especial que tiene métodos adicionales para recuperar información desde y para responder al evento.

El despachador

El despachador es el objeto central del sistema despachador de eventos. En general, se crea un único despachador, el cual mantiene un registro de escuchas. Cuando se difunde un evento a través del despachador, este notifica a todos los escuchas registrados con ese evento.

```
use Symfony\Component\EventDispatcher\EventDispatcher;

$despachador = new EventDispatcher();
```

Conectando escuchas

Para aprovechar las ventajas de un evento existente, es necesario conectar un escucha con el despachador para que pueda ser notificado cuando se despache el evento. Una llamada al método despachador `addListener()` asocia cualquier objeto PHP ejecutable a un evento:

```
$escucha = new AcmeListener();
$despachador->addListener('foo.action', array($escucha, 'onFooAction'));
```

El método `addListener()` toma hasta tres argumentos:

- El nombre del evento (cadena) que este escucha quiere atender;
- Un objeto PHP ejecutable que será notificado cuando se produzca un evento al que está atento;

- Un entero de prioridad opcional (mayor es igual a más importante) que determina cuando un escucha se activa frente a otros escuchas (por omisión es `0`). Si dos escuchas tienen la misma prioridad, se ejecutan en el orden en que se agregaron al despachador.

Nota: Un PHP ejecutable es una variable *PHP* que la función `call_user_func()` puede utilizar y devuelve `true` cuando pasa a la función `is_callable()`. Esta puede ser una instancia de `\Closure`, una cadena que representa una función, o una matriz que representa a un objeto método o un método de clase.

Hasta ahora, hemos visto cómo los objetos PHP se pueden registrar como escuchas. También puedes registrar Cierres PHP como escuchas de eventos:

```
use Symfony\Component\EventDispatcher\Event;

$despachador->addListener('foo.action', function (Event $evento) {
    // se debe ejecutar cuando se despache el evento foo.action
});
```

Una vez que se registra el escucha en el despachador, este espera hasta que el evento sea notificado. En el ejemplo anterior, cuando se despacha el evento `foo.action`, el despachador llama al método `AcmeListener::onFooAction` y pasa el objeto `Evento` como único argumento:

```
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

    public function onFooAction(Event $evento)
    {
        // hace alguna cosa
    }
}
```

Truco: Si utilizas la plataforma MVC de *Symfony2*, los escuchas se pueden registrar a través de tu *configuración* (Página 561). Como bono adicional, los objetos escucha sólo se crean cuando son necesarios.

En muchos casos, una subclase especial `Evento` específica para el evento dado es pasada al escucha. Esto le da al escucha acceso a información especial sobre el evento. Consulta la documentación o la implementación de cada evento para determinar la instancia exacta de `Symfony\Component\EventDispatcher\Event` que se ha pasado. Por ejemplo, el evento `kernel.event` pasa una instancia de `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

```
use Symfony\Component\HttpKernel\Event\FilterResponseEvent

public function onKernelResponse(FilterResponseEvent $evento)
{
    $respuesta = $evento->getResponse();
    $peticion = $evento->getRequest();

    // ...
}
```

Creando y despachando un evento

Además de registrar escuchas con eventos existentes, puedes crear y lanzar tus propios eventos. Esto es útil cuando creas bibliotecas de terceros y también cuando deseas mantener flexibles y desconectados diferentes componentes de tu propio sistema.

La clase estática `Events`

Supongamos que deseas crear un nuevo evento - `guarda.orden` - el cual se despacha cada vez que es creada una orden dentro de tu aplicación. Para mantener las cosas organizadas, empieza por crear una clase `GuardaEventos` dentro de tu aplicación que sirva para definir y documentar tu evento:

```
namespace Acme\GuardaBundle;

final class GuardaEventos
{
    /**
     * El evento guarda.orden es lanzado cada vez que se crea una orden
     * en el sistema.
     *
     * El escucha del evento recibe una instancia de
     * Acme\GuardaBundle\Event\FiltraOrdenEvent.
     *
     * @var string
     */
    const onGuardaOrden = 'guarda.orden';
}
```

Ten en cuenta que esta clase en realidad no *hace* nada. El propósito de la clase `GuardaEventos` sólo es ser un lugar donde se pueda centralizar la información sobre los eventos comunes. Observa también que se pasará una clase especial `FiltraOrdenEvent` a cada escucha de este evento.

Creando un objeto Evento

Más tarde, cuando despaches este nuevo evento, debes crear una instancia del `Evento` y pasarla al despachador. Entonces el despachador pasa esta misma instancia a cada uno de los escuchas del evento. Si no necesitas pasar alguna información a tus escuchas, puedes utilizar la clase predeterminada `Symfony\Component\EventDispatcher\Event`. La mayoría de las veces, sin embargo, *necesitarás* pasar información sobre el evento a cada escucha. Para lograr esto, vamos a crear una nueva clase que extiende a `Symfony\Component\EventDispatcher\Event`.

En este ejemplo, cada escucha tendrá acceso a algún objeto `Orden`. Crea una clase `Evento` que lo hace posible:

```
namespace Acme\GuardaBundle\Event;

use Symfony\Component\EventDispatcher\Event;
use Acme\GuardaBundle\Orden;

class FiltraOrdenEvent extends Event
{
    protected $orden;

    public function __construct(Orden $orden)
    {
        $this->orden = $orden;
    }
```

```
    public function getOrden()
    {
        return $this->orden;
    }
}
```

Ahora cada escucha tiene acceso al objeto `Orden` a través del método `getOrden`.

Despachando el evento

El método **:method:'Symfony\\Component\\EventDispatcher\\EventDispatcher::dispatch'** notifica a todos los escuchas del evento dado. Este toma dos argumentos: el nombre del evento a despachar, y la instancia del `Evento` a pasar a cada escucha de ese evento:

```
use Acme\GuardaBundle\GuardaEventos;
use Acme\GuardaBundle\Orden;
use Acme\GuardaBundle\Event\FiltraOrdenEvent;

// la orden de alguna manera es creada o recuperada
$orden = new Orden();
// ...

// crea el FiltraOrdenEvent y lo despacha
$evento = new FiltraOrdenEvent($orden);
$despachador->dispatch(GuardaEventos::onGuardaOrden, $evento);
```

Ten en cuenta que el objeto especial `FiltraOrdenEvent` se crea y pasa al método `dispatch`. Ahora, cualquier escucha del evento `guarda.orden` recibirá el `FiltraOrdenEvent` y tendrá acceso al objeto `Orden` a través del método `getOrden`:

```
// alguna clase escucha que se ha registrado para onGuardaOrden
use Acme\GuardaBundle\Event\FiltraOrdenEvent;

public function onGuardaOrden(FiltraOrdenEvent $evento)
{
    $orden = $evento->getOrden();
    // hace algo a/con la orden
}
```

Pasando el objeto despachador de evento

Si echas un vistazo a la clase `EventDispatcher`, te darás cuenta de que la clase no actúa como un Singleton (no hay un método estático `getInstance()`). Esto es intencional, ya que posiblemente desees tener varios despachadores de eventos simultáneos en una sola petición *PHP*. Pero también significa que necesitas una manera de pasar el despachador a los objetos que necesitan conectar o notificar eventos.

La mejor práctica consiste en inyectar el objeto despachador de eventos en tus objetos, también conocido como inyección de dependencias.

Puedes usar el constructor de inyección:

```
class Foo
{
    protected $despachador = null;

    public function __construct(EventDispatcher $despachador)
```

```
    {
        $this->despachador = $despachador;
    }
}
```

O definir la inyección:

```
class Foo
{
    protected $despachador = null;

    public function setEventDispatcher(EventDispatcher $despachador)
    {
        $this->despachador = $despachador;
    }
}
```

La elección entre los dos realmente es cuestión de gusto. Muchos tienden a preferir el constructor de inyección porque los objetos son totalmente iniciados en tiempo de construcción. Pero cuando tienes una larga lista de dependencias, la inyección de definidores puede ser el camino a seguir, especialmente para dependencias opcionales.

Truco: Si utilizas la inyección de dependencias como lo hicimos en los dos ejemplos anteriores, puedes utilizar el componente Inyección de dependencias de Symfony2 para manejar elegantemente estos objetos.

Usando suscriptores de evento

La forma más común para escuchar a un evento es registrar un *escucha de evento* con el despachador. Este escucha puede estar atento a uno o más eventos y ser notificado cada vez que se envían los eventos.

Otra forma de escuchar eventos es a través de un *suscriptor de eventos*. Un suscriptor de eventos es una clase *PHP* que es capaz de decir al despachador exactamente a cuales eventos debe estar suscrito. Este implementa la interfaz `Symfony\Component\EventDispatcher\EventSubscriberInterface`, que requiere un solo método estático llamado `getSubscribedEvents`. Considera el siguiente ejemplo de un suscriptor que está inscrito a los eventos `kernel.response` y `guarda.orden`:

```
namespace Acme\GuardaBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class GuardaSubscriber implements EventSubscriberInterface
{
    static public function getSubscribedEvents()
    {
        return array(
            'kernel.response' => 'onKernelResponse',
            'guarda.orden'     => 'onGuardaOrden',
        );
    }

    public function onKernelResponse(FilterResponseEvent $evento)
    {
        // ...
    }

    public function onGuardaOrden(FiltraOrdenEvent $evento)
```

```
    {
        // ...
    }
}
```

Esto es muy similar a una clase escucha, salvo que la propia clase puede decir al despachador cuales eventos debe escuchar. Para registrar un suscriptor al despachador, utiliza el método **:method:'Symfony\\Component\\EventDispatcher\\EventDispatcher::addSubscriber'**:

```
use Acme\GuardaBundle\Event\GuardaSubscriber;

$subscriber = new GuardaSubscriber();
$despachador->addSubscriber($subscriber);
```

El despachador registrará automáticamente al suscriptor para cada evento devuelto por el método `getSubscribedEvents`. Al igual que con los escuchas, el método `addSubscriber` tiene un segundo argumento opcional, que es la prioridad que se debe dar a cada evento.

Deteniendo el flujo/propagación del evento

En algunos casos, puede tener sentido que un escucha evite que se llame a otros escuchas. En otras palabras, el escucha tiene que poder decirle al despachador detenga la propagación del evento a todos los escuchas en el futuro (es decir, no notificar a más escuchas). Esto se puede lograr desde el interior de un escucha a través del método **:method:'Symfony\\Component\\EventDispatcher\\Event::stopPropagation'**:

```
use Acme\GuardaBundle\Event\FiltraOrdenEvent;

public function onGuardaOrden(FiltraOrdenEvent $evento)
{
    // ...

    $evento->stopPropagation();
}
```

Ahora, cualquier escucha de `guarda.orden` que aún no ha sido llamado *no* será invocado.

2.28.4 Generador de perfiles

Cuando se activa, el generador de perfiles de *Symfony2* recoge información útil sobre cada petición presentada a tu aplicación y la almacena para su posterior análisis. Utiliza el generador de perfiles en el entorno de desarrollo para ayudar a depurar el código y mejorar el rendimiento, úsalo en el entorno de producción para explorar problemas después del hecho.

Rara vez tienes que lidiar con el generador de perfiles directamente puesto que *Symfony2* proporciona herramientas de visualización como la barra de herramientas de depuración web y el generador de perfiles web. Si utilizas la Edición estándar de *Symfony2*, el generador de perfiles, la barra de herramientas de depuración web, y el generador de perfiles web, ya están configurados con ajustes razonables.

Nota: El generador de perfiles recopila información para todas las peticiones (peticiones simples, redirecciones, excepciones, peticiones *Ajax*, peticiones ESI, y para todos los métodos *HTTP* y todos los formatos). Esto significa que para una única *URL*, puedes tener varios perfiles de datos asociados (un par petición/respuesta externa).

Visualizando perfiles de datos

Usando la barra de depuración web

En el entorno de desarrollo, la barra de depuración web está disponible en la parte inferior de todas las páginas. Esta muestra un buen resumen de los datos de perfiles que te da acceso instantáneo a una gran cantidad de información útil cuando algo no funciona como se esperaba.

Si el resumen presentado por las herramientas de la barra de depuración web no es suficiente, haz clic en el enlace simbólico (una cadena compuesta de 13 caracteres aleatorios) para acceder al generador de perfiles web.

Nota: Si no se puede hacer clic en el enlace, significa que las rutas del generador de perfiles no están registradas (más abajo hay información de configuración).

Analizando datos del perfil con el generador de perfiles web

El generador de perfiles web es una herramienta de visualización para el perfilado de datos que puedes utilizar en desarrollo para depurar tu código y mejorar el rendimiento, pero también lo puedes utilizar para explorar problemas que se producen en producción. Este expone toda la información recogida por el generador de perfiles en una interfaz web.

Accediendo a información del generador de perfiles

No es necesario utilizar el visualizador predeterminado para acceder a la información de perfiles. Pero ¿cómo se puede recuperar información de perfiles de una petición específica después del hecho? Cuando el generador de perfiles almacena datos sobre una petición, también asocia un símbolo con ella, esta muestra está disponible en la cabecera HTTP `X-Debug-Token` de la Respuesta:

```
$perfil = $contenedor->get('profiler')->loadProfileFromResponse($respuesta);

$perfil = $contenedor->get('profiler')->loadProfile($muestra);
```

Truco: Cuando el generador de perfiles está habilitado pero no la barra de herramientas de depuración web, o cuando desees obtener el símbolo de una petición Ajax, utiliza una herramienta como Firebug para obtener el valor de la cabecera HTTP `X-Debug-Token`.

Usa el método `find()` para acceder a elementos basándose en algún criterio:

```
// consigue los 10 últimas fragmentos
$muestras = $contenedor->get('profiler')->find('', '', 10);

// consigue los 10 últimos fragmentos de todas las URL que contienen /admin/
$muestras = $contenedor->get('profiler')->find('', '/admin/', 10);

// consigue los 10 últimos fragmentos de peticiones locales
$muestras = $contenedor->get('profiler')->find('127.0.0.1', '', 10);
```

Si deseas manipular los datos del perfil en una máquina diferente a la que generó la información, utiliza los métodos `export()` e `import()`:

```
// en la máquina en producción
$perfil = $contenedor->get('profiler')->loadProfile($muestra);
```

```
$datos = $profiler->export($perfil);

// en la máquina de desarrollo
$profiler->import($datos);
```

Configurando

La configuración predeterminada de *Symfony2* viene con ajustes razonables para el generador de perfiles, la barra de herramientas de depuración web, y el generador de perfiles web. Aquí está por ejemplo la configuración para el entorno de desarrollo:

- *YAML*

```
# carga el generador de perfiles
framework:
    profiler: { only_exceptions: false }

# activa el generador de perfiles web
web_profiler:
    toolbar: true
    intercept_redirects: true
    verbose: true
```

- *XML*

```
<!-- xmlns:webprofiler="http://symfony.com/schema/dic/webprofiler" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/webprofiler http://symfony.com/schema/dic

<!-- carga el generador de perfiles -->
<framework:config>
    <framework:profiler only-exceptions="false" />
</framework:config>

<!-- activa el generador de perfiles web -->
<webprofiler:config
    toolbar="true"
    intercept-redirects="true"
    verbose="true"
/>
```

- *PHP*

```
// carga el generador de perfiles
$contenedor->loadFromExtension('framework', array(
    'profiler' => array('only-exceptions' => false),
));

// activa el generador de perfiles web
$contenedor->loadFromExtension('web_profiler', array(
    'toolbar' => true,
    'intercept-redirects' => true,
    'verbose' => true,
));
```

Cuando `only-exceptions` se establece a `true`, el generador de perfiles sólo recoge datos cuando tu aplicación lanza una excepción.

Cuando `intercept-redirects` está establecido en `true`, el generador de perfiles web intercepta las redirecciones y te da la oportunidad de analizar los datos recogidos antes de seguir la redirección.

Cuando `verbose` está establecido en `true`, la barra de herramientas de depuración web muestra una gran cantidad de información. Configurar `verbose` a `false` oculta algo de información secundaria para hacer más corta la barra de herramientas.

Si activas el generador de perfiles web, también es necesario montar las rutas de los perfiles:

- *YAML*

```
_profiler:
    resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
    prefix:   /_profiler
```

- *XML*

```
<import resource="@WebProfilerBundle/Resources/config/routing/profiler.xml" prefix="/_profiler"
```

- *PHP*

```
$coleccion->addCollection($cargador->import("@WebProfilerBundle/Resources/config/routing/profile
```

Dado que el generador de perfiles añade algo de sobrecarga, posiblemente desees activarlo sólo bajo ciertas circunstancias en el entorno de producción. La configuración `only-exceptions` limita al generador de perfiles a 500 páginas, ¿pero si quieres obtener información cuando el cliente IP proviene de una dirección específica, o para una parte limitada de la página web? Puedes utilizar una emparejadora de petición:

- *YAML*

```
# activa el generador de perfiles sólo para peticiones entrantes de la red 192.168.0.0
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24 }

# activa el generado de perfiles sólo para las URL /admin
framework:
    profiler:
        matcher: { path: "^/admin/" }

# combina reglas
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24, path: "^/admin/" }

# usa una instancia emparejadora personalizada definida en el servicio "custom_matcher"
framework:
    profiler:
        matcher: { service: custom_matcher }
```

- *XML*

```
<!-- activa el generador de perfiles sólo para peticiones entrantes de la red 192.168.0.0 -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" />
    </framework:profiler>
</framework:config>

<!-- activa el generador de perfiles sólo para las URL /admin -->
<framework:config>
```

```
    <framework:profiler>
        <framework:matcher path="^/admin/" />
    </framework:profiler>
</framework:config>

<!-- combina reglas -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" path="^/admin/" />
    </framework:profiler>
</framework:config>

<!-- usa una instancia emparejadora personalizada definida en el servicio "custom_matcher" -->
<framework:config>
    <framework:profiler>
        <framework:matcher service="custom_matcher" />
    </framework:profiler>
</framework:config>
```

- *PHP*

```
// activa el generador de perfiles sólo para peticiones entrantes de la red 192.168.0.0
$contenedor->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('ip' => '192.168.0.0/24'),
    ),
));

// activa el generador de perfiles sólo para las URL /admin
$contenedor->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('path' => '^/admin/'),
    ),
));

// combina reglas
$contenedor->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('ip' => '192.168.0.0/24', 'path' => '^/admin/'),
    ),
));

# usa una instancia emparejadora personalizada definida en el servicio "custom_matcher"
$contenedor->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('service' => 'custom_matcher'),
    ),
));
```

2.28.5 Aprende más en el recetario

- *Cómo utilizar el generador de perfiles en una prueba funcional* (Página 366)
- *Cómo crear un colector de datos personalizado* (Página 425)
- *Cómo extender una clase sin necesidad de utilizar herencia* (Página 421)
- *Cómo personalizar el comportamiento de un método sin utilizar herencia* (Página 423)

2.29 *API* estable de *Symfony2*

La *API* estable de *Symfony2* es un subconjunto de todos los métodos públicos de *Symfony2* (componentes y paquetes básicos) que comparten las siguientes propiedades:

- El nombre del espacio de nombres y clase no va a cambiar;
- El nombre del método no va a cambiar;
- La firma del método (argumentos y tipo del valor de retorno) no va a cambiar;
- La semántica del método no va a cambiar.

Sin embargo, la impremetación en sí misma puede cambiar. El único caso válido para un cambio en la *API* estable es con el fin de corregir un problema de seguridad.

La *API* estable se basa en una lista blanca, marcada con `@api`. Por lo tanto, todo lo no etiquetado explícitamente no es parte de la *API* estable.

Truco: Cualquier paquete de terceros también deberá publicar su propia *API* estable.

A partir de *Symfony 2.0*, los siguientes componentes tienen una *API* etiquetada pública:

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating
- Translation
- Validator
- Yaml
- *Symfony2 y fundamentos HTTP* (Página 33)
- *Symfony2 frente a PHP simple* (Página 41)
- *Instalando y configurando Symfony* (Página 53)
- *Creando páginas en Symfony2* (Página 57)
- *Controlador* (Página 71)

- *Enrutando* (Página 81)
- *Creando y usando plantillas* (Página 98)
- *Bases de datos y Doctrine ("El modelo")* (Página 115)
- *Probando* (Página 137)
- *Validando* (Página 149)
- *Formularios* (Página 160)
- *Seguridad* (Página 181)
- *Caché HTTP* (Página 210)
- *Traduciendo* (Página 224)
- *Contenedor de servicios* (Página 237)
- *Rendimiento* (Página 252)
- *Funcionamiento interno* (Página 254)
- *API estable de Symfony2* (Página 270)
- *Symfony2 y fundamentos HTTP* (Página 33)
- *Symfony2 frente a PHP simple* (Página 41)
- *Instalando y configurando Symfony* (Página 53)
- *Creando páginas en Symfony2* (Página 57)
- *Controlador* (Página 71)
- *Enrutando* (Página 81)
- *Creando y usando plantillas* (Página 98)
- *Bases de datos y Doctrine ("El modelo")* (Página 115)
- *Probando* (Página 137)
- *Validando* (Página 149)
- *Formularios* (Página 160)
- *Seguridad* (Página 181)
- *Caché HTTP* (Página 210)
- *Traduciendo* (Página 224)
- *Contenedor de servicios* (Página 237)
- *Rendimiento* (Página 252)
- *Funcionamiento interno* (Página 254)
- *API estable de Symfony2* (Página 270)

Parte III

Recetario

CAPÍTULO 3

Recetario

3.1 Cómo crear y guardar un proyecto *Symfony2* en *git*

Truco: A pesar de que este artículo específicamente es acerca de *git*, los mismos principios genéricos se aplican si estás guardando el proyecto en *Subversión*.

Una vez hayas leído *Creando páginas en Symfony2* (Página 57) y te sientas cómodo usando *Symfony*, sin duda estarás listo para comenzar tu propio proyecto. En este artículo, aprenderás la mejor manera de empezar un nuevo proyecto *Symfony2* y almacenarlo usando el sistema de control de versión distribuido git.

3.1.1 Configuración inicial del proyecto

Para empezar, tendrás que descargar *Symfony* e iniciar tu repositorio *git* local:

1. Descarga la Edición estándar de Symfony2 `sin vendors`.
2. Descomprime la distribución. Esto creará un directorio llamado *Symfony* con tu nueva estructura del proyecto, archivos de configuración, etc. Cambia el nombre *Symfony* a lo que quieras.
3. Crea un nuevo archivo llamado `.gitignore` en la raíz de tu nuevo proyecto (por ejemplo, al lado del archivo `deps`) y pega lo siguiente en él. Los archivos que coincidan con estos patrones serán ignorados por *git*:

```
/web/bundles/
/app/bootstrap*
/app/cache/*
/app/logs/*
/vendor/
/app/config/parameters.ini
```

4. Copia `app/config/parameters.ini` a `app/config/parameters.ini.dist`. El archivo `parameters.ini` es ignorado por *git* (ve más arriba) para que la configuración específica de la máquina - como las contraseñas de bases de datos - no esté comprometida. Al crear el archivo `parameters.ini.dist`, los nuevos desarrolladores rápidamente pueden clonar el proyecto, copiar este archivo a `parameters.ini`, personalizarlo, y empezar a desarrollar.
5. Inicia el repositorio *git*:

```
$ git init
```

6. Agrega todos los archivos iniciales a *git*:

```
$ git add .
```

7. Crear una consignación inicial en tu proyecto recién iniciado:

```
$ git commit -m "Consigna inicial"
```

8. Finalmente, descarga todas las bibliotecas de terceros:

```
$ php bin/vendors install
```

En este punto, tienes un proyecto *Symfony2* totalmente funcional consignado correctamente a *git*. Puedes comenzar a desarrollarlo inmediatamente, consignando los nuevos cambios al repositorio *git*.

Puedes continuar, siguiendo el capítulo *Creando páginas en Symfony2* (Página 57) para aprender más acerca de cómo configurar y desarrollar tu aplicación.

Truco: La *Edición estándar de Symfony2* viene con alguna funcionalidad de ejemplo. Para eliminar el código de ejemplo, sigue las instrucciones del archivo Readme de la edición estándar.

3.1.2 Gestionando bibliotecas de proveedores con `bin/vendors` y `deps`

Cada proyecto *Symfony* utiliza un gran número de bibliotecas de terceros - "vendor".

Por omisión, estas bibliotecas se descargan ejecutando el guión `php bin/vendors install`. Este guión lee el archivo `deps`, y descarga las bibliotecas especificadas en el directorio `vendor/`. También lee el archivo `deps.lock`, fijando cada biblioteca listada al `hash` de consignación exacto.

En esta configuración, las bibliotecas de los proveedores no son parte de tu repositorio *git*, ni siquiera como submódulos. En su lugar, nos basamos en los archivos `deps` y `deps.lock`, y en el guión `bin/vendors` para manejar todo. Estos archivos son parte de tu repositorio, por lo que las versiones necesarias de cada biblioteca de terceros están bajo el control de versiones en *git*, y puedes usar el guión `vendors` para actualizar tu proyecto hasta la fecha.

Cada vez que un desarrollador clone un proyecto, él/ella debe ejecutar el guión `php bin/vendors install` para asegurarse de que se descarguen todas las bibliotecas de proveedores necesarias.

Actualizando *Symfony*

Debido a que *Symfony* es un grupo de bibliotecas de terceros y las bibliotecas de terceros son controladas completamente a través de `deps` y `deps.lock`, actualizar *Symfony* significa simplemente actualizar cada uno de estos archivos para que coincida su estado con la última *Edición estándar de Symfony*.
Por supuesto, si has agregado nuevas entradas a `deps` o `deps.lock`, asegúrate de sustituir sólo las partes originales (es decir, asegúrate de no eliminar también cualquiera de tus entradas personalizadas).

Prudencia: También hay una orden `php bin/vendors upgrade`, pero esta no tiene nada que ver con la actualización de tu proyecto y normalmente no hay que usarla. Esta orden se utiliza para congelar las versiones de todas las bibliotecas de tu proveedor leyendo su estado actual y registrándolo en el archivo `deps.lock`.

`Vendors` y submódulos

En lugar de utilizar el sistema `deps`, `bin/vendors` para gestionar tus bibliotecas de proveedores, puedes optar en su lugar por utilizar los submódulos git nativos. No hay nada malo con este enfoque, aunque el sistema `deps` es la forma oficial de solucionar este problema y los submódulos de *git* pueden ser difíciles de trabajar con el tiempo.

3.1.3 Almacenando tu proyecto en un servidor remoto

Ahora tienes un proyecto *Symfony2* totalmente funcional almacenado en *git*. Sin embargo, en la mayoría de los casos, también desearás guardar tu proyecto en un servidor remoto, tanto con fines de seguridad, como para que otros desarrolladores puedan colaborar en el proyecto.

La manera más fácil de almacenar tu proyecto en un servidor remoto es a través de GitHub. Los repositorios públicos son gratuitos, sin embargo tendrás que pagar una cuota mensual para tener repositorios privados.

Alternativamente, puedes almacenar tu repositorio *git* en cualquier servidor creando un repositorio minimalista y luego `empujar` al mismo. Una biblioteca que te ayuda a gestionar esto es Gitosis.

3.2 Cómo personalizar páginas de error

Cuando se lanza alguna excepción en *Symfony2*, la excepción es capturada dentro de la clase `Kernel` y, finalmente, remitida a un controlador especial, `TwigBundle:Exception:show` para procesarla. Este controlador, el cual vive dentro del núcleo de `TwigBundle`, determina cual plantilla de error mostrar y el código de estado que se debe establecer para la excepción dada.

Puedes personalizar las páginas de error de dos formas diferentes, dependiendo de la cantidad de control que necesites:

1. Personalizando las plantillas de error de las diferentes páginas de error (se explica más adelante);
2. Reemplazando el controlador de excepciones predeterminado `TwigBundle::Exception:show` con tu propio controlador y procésalo como quieras (consulta *exception_controller en la referencia de Twig* (Página 456));

Truco: Personalizar el tratamiento de las excepciones en realidad es mucho más poderoso que lo escrito aquí. Produce un evento interno, `core.exception`, el cual te permite completo control sobre el manejo de la excepción. Para más información, consulta el *Evento kernel.exception* (Página 258).

Todas las plantillas de error viven dentro de `TwigBundle`. Para sustituir las plantillas, simplemente confiamos en el método estándar para sustituir las plantillas que viven dentro de un paquete. Para más información, consulta *Sustituyendo plantillas del paquete* (Página 111).

Por ejemplo, para sustituir la plantilla de error predeterminada mostrada al usuario final, crea una nueva plantilla ubicada en `app/Resources/TwigBundle/views/Exception/error.html.twig`:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Ha ocurrido un error: {{ texto_estado }}</title>
</head>
<body>
    <h1>¡Oops! Ha ocurrido un error</h1>
    <h2>El servidor devolvió un "{{ codigo_estado }} {{ texto_estado }}".</h2>
</body>
</html>
```

Truco: Si no estás familiarizado con *Twig*, no te preocupes. *Twig* es un sencillo, potente y opcional motor de plantillas que se integra con *Symfony2*. Para más información sobre *Twig* consulta *Creando y usando plantillas* (Página 98).

Además de la página de error *HTML* estándar, *Symfony* proporciona una página de error predeterminada para muchos de los más comunes formatos de respuesta, como *JSON* (`error.json.twig`), *XML* (`error.xml.twig`) e incluso *Javascript* (`error.js.twig`), por nombrar algunos. Para sustituir cualquiera de estas plantillas, basta con crear un nuevo archivo con el mismo nombre en el directorio `app/Resources/TwigBundle/views/Exception`. Esta es la manera estándar de sustituir cualquier plantilla que viva dentro de un paquete.

3.2.1 Personalizando la página 404 y otras páginas de error

También puedes personalizar plantillas de error específicas de acuerdo con el código de estado *HTTP*. Por ejemplo, crea una plantilla `app/Resources/TwigBundle/views/Exception/error404.html.twig` para mostrar una página especial para los errores 404 (página no encontrada).

Symfony utiliza el siguiente algoritmo para determinar qué plantilla utilizar:

- En primer lugar, busca una plantilla para el formato dado y el código de estado (como `error404.json.twig`);
- Si no existe, busca una plantilla para el formato propuesto (como `error.json.twig`);
- Si no existe, este cae de nuevo a la plantilla *HTML* (como `error.html.twig`).

Truco: Para ver la lista completa de plantillas de error predeterminadas, revisa el directorio `Resources/views/Exception` de `TwigBundle`. En una instalación estándar de *Symfony2*, el `TwigBundle` se puede encontrar en `vendor/symfony/src/Symfony/Bundle/TwigBundle`. A menudo, la forma más fácil de personalizar una página de error es copiarla de `TwigBundle` a `app/Resources/TwigBundle/views/Exception` y luego modificarla.

Nota: El amigable depurador de páginas de excepción muestra al desarrollador cómo, incluso, puede personalizar de la misma manera creando plantillas como `exception.html.twig` para la página de excepción *HTML* estándar o `exception.json.twig` para la página de excepción *JSON*.

3.3 Cómo definir controladores como servicios

En el libro, has aprendido lo fácilmente que se puede utilizar un controlador cuando se extiende la clase base `Symfony\Bundle\FrameworkBundle\Controller\Controller`. Si bien esto funciona estupendamente, los controladores también se pueden especificar como servicios.

Para referir un controlador que se defina como servicio, utiliza la notación de dos puntos individuales (:). Por ejemplo, supongamos que hemos definido un servicio llamado `mi_controlador` y queremos que redirija a un método llamado `indexAction()` dentro del servicio:

```
$this->forward('mi_controlador:indexAction', array('foo' => $bar));
```

Necesitas usar la misma notación para definir el valor `_controller` de la ruta:

```
mi_controlador:
    pattern:   /
    defaults:  { _controller: mi_controlador:indexAction }
```

Para utilizar un controlador de esta manera, este se debe definir en la configuración del contenedor de servicios. Para más información, consulta el capítulo *Contenedor de servicios* (Página 237).

Cuando se utiliza un controlador definido como servicio, lo más probable es no ampliar la clase base `Controller`. En lugar de confiar en sus métodos de acceso directo, debes interactuar directamente con los servicios que necesitas. Afortunadamente, esto suele ser bastante fácil y la clase base `Controller` en sí es una gran fuente sobre la manera de realizar muchas tareas comunes.

Nota: Especificar un controlador como servicio requiere un poco más de trabajo. La principal ventaja es que el controlador completo o cualquier otro servicio pasado al controlador se puede modificar a través de la configuración del contenedor de servicios. Esto es útil especialmente cuando desarrollas un paquete de código abierto o cualquier paquete que se pueda utilizar en muchos proyectos diferentes. Así que, aunque no especifiques los controladores como servicios, es probable que veas hacer esto en algunos paquetes de código abierto de *Symfony2*.

3.4 Cómo forzar las rutas para utilizar siempre *HTTPS*

A veces, deseas proteger algunas rutas y estar seguro de que siempre se accede a ellas a través del protocolo *HTTPS*. El componente `Routing` te permite forzar el esquema HTTP a través del requisito `_scheme`:

- *YAML*

```
protegida:
    pattern:  /protegida
    defaults: { _controller: AcmeDemoBundle:Principal:protegida }
    requirements:
        _scheme:  https
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="protegida" pattern="/protegida">
        <default key="_controller">AcmeDemoBundle:Principal:protegida</default>
        <requirement key="_scheme">https</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('protegida', new Route('/protegida', array(
    '_controller' => 'AcmeDemoBundle:Principal:protegida',
), array(
    '_scheme' => 'https',
)));

return $coleccion;
```

La configuración anterior obliga a utilizar siempre la ruta `protegida` *HTTPS*.

Cuando se genera la *URL* `protegida`, y si el esquema actual es *HTTP*, *Symfony* generará automáticamente una *URL* absoluta con *HTTPS* como esquema:

```
# Si el esquema actual es HTTPS
{{ path('protegida') }}
# genera /protegida

# Si el esquema actual es HTTP
{{ path('protegida') }}
# genera https://ejemplo.com/protegida
```

El requisito también aplica para las peticiones entrantes. Si intentas acceder a la ruta `/protegida` con HTTP, automáticamente se te redirige a la misma *URL*, pero con el esquema *HTTPS*.

El ejemplo anterior utiliza `https` para el `_scheme`, pero también puedes obligar a que una *URL* siempre utilice `http`.

Nota: El componente `Security` proporciona otra manera de hacer cumplir el esquema HTTP a través de la creación de `requires_channel`. Este método alternativo es más adecuado para proteger "una amplia área" de tu sitio web (todas las direcciones *URL* en `/admin`) o cuando deseas proteger direcciones *URL* definidas en un paquete de terceros.

3.5 Cómo permitir un carácter "/" en un parámetro de ruta

A veces, es necesario componer las *URL* con parámetros que pueden contener una barra inclinada `/`. Por ejemplo, tomemos la ruta clásica `/hola/{nombre}`. Por omisión, `/hola/Fabien` coincidirá con esta ruta pero no `/hola/Fabien/Kris`. Esto se debe a que *Symfony* utiliza este carácter como separador entre las partes de la ruta.

Esta guía explica cómo puedes modificar una ruta para que `/hola/Fabien/Kris` coincida con la ruta `/hola/{nombre}`, donde `{nombre}` es igual a `Fabien/Kris`.

3.5.1 Configurando la ruta

De manera predeterminada, el componente de enrutado de *Symfony* requiere que los parámetros coincidan con los siguientes patrones de expresiones regulares: `[^/]+`. Esto significa que todos los caracteres están permitidos excepto `/`.

Debes permitir el carácter `/` explícitamente para que sea parte de tu parámetro especificando un patrón de expresión regular más permisivo.

- *YAML*

```
_hola:
    pattern: /hola/{nombre}
    defaults: { _controller: AcmeDemoBundle:Demo:hola }
    requirements:
        nombre: ".+"
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout
```

```
    <route id="_hola" pattern="/hola/{nombre}">
        <default key="_controller">AcmeDemoBundle:Demo:hola</default>
        <requirement key="nombre">.+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$coleccion = new RouteCollection();
$coleccion->add('_hola', new Route('/hola/{nombre}', array(
    '_controller' => 'AcmeDemoBundle:Demo:hola',
), array(
    'nombre' => '.+',
)));

return $coleccion;
```

- *Annotations*

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class DemoController
{
    /**
     * @Route("/hola/{nombre}", name="_hola", requirements={"nombre" = ".+"})
     */
    public function holaAction($nombre)
    {
        // ...
    }
}
```

¡Eso es todo! Ahora, el parámetro `{nombre}` puede contener el carácter `/`.

3.6 Cómo utilizar `Assetic` para gestionar activos

`Assetic` es una biblioteca para gestionar activos que se incluye con la distribución estándar de *Symfony2* y se puede utilizar fácilmente en *Symfony2* directamente desde las plantillas *Twig* o PHP.

`Assetic` combina dos ideas principales: *activos* y *filtros*. Los *activos* son archivos tal como los archivos *CSS*, *JavaScript* e imágenes. Los *filtros* son cosas que se pueden aplicar a estos archivos antes de servirlos al navegador. Esto te permite una separación entre los archivos de activos almacenados en tu aplicación y los archivos presentados realmente al usuario.

Sin `Assetic`, sólo sirves los archivos que están almacenados directamente en la aplicación:

- *Twig*

```
<script src="{{ asset('js/script.js') }}" type="text/javascript" />
```

- *PHP*

```
<script src="<?php echo $view['assets']->getUrl('js/script.js') ?>"
        type="text/javascript" />
```

Sin embargo, *con* `Assetic`, puedes manipular estos activos como quieras (o cargarlos desde cualquier lugar) antes de servirlos. Esto significa que puedes:

- Minimizarlos con `minify` y combinar todos tus archivos *CSS* y JS
- Ejecutar todos (o algunos) de tus archivos *CSS* o JS a través de algún tipo de compilador, como LESS, SASS o CoffeeScript
- Ejecutar optimizaciones de imagen en tus imágenes

3.6.1 Activos

`Assetic` ofrece muchas ventajas sobre los archivos que sirves directamente. Los archivos no se tienen que almacenar dónde son servidos y se pueden obtener de diversas fuentes, tal como desde dentro de un paquete:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*'
%}
<script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
<script type="text/javascript" src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

En este ejemplo, todos los archivos en el directorio `Resources/public/js/` del `AcmeFooBundle` se cargan y sirven desde un lugar diferente. En realidad la etiqueta reproducida simplemente podría ser:

```
<script src="/js/abcd123.js"></script>
```

También puedes combinar varios archivos en uno solo. Esto ayuda a reducir el número de peticiones *HTTP*, lo cual es bueno para un rendimiento frontal extremo, ya que la mayoría de los navegadores sólo procesan un número limitado a la vez frenando la carga de tus páginas. También te permite mantener los archivos con mayor facilidad dividiéndolos en partes manejables. Esto también te puede ayudar con la reutilización puesto que fácilmente puedes dividir los archivos de proyectos específicos de los que puedes utilizar en otras aplicaciones, pero aún los servirás como un solo archivo:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*'
               '@AcmeBarBundle/Resources/public/js/form.js'
               '@AcmeBarBundle/Resources/public/js/calendar.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*',
          '@AcmeBarBundle/Resources/public/js/form.js',
          '@AcmeBarBundle/Resources/public/js/calendar.js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Esto no sólo se aplica a tus propios archivos, también lo puede utilizar `Assetic` para combinar activos de terceros, tal como jQuery como un solo archivo en sí mismo:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js'
               '@AcmeFooBundle/Resources/public/js/*'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js',
          '@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

3.6.2 Filtros

Adicionalmente a esto `Assetic`, puede aplicar *filtros* a tus *activos* antes de servirlos. Esto incluye tareas tales como compresión de la salida para reducir el tamaño de los archivos, el cual es otro valor de optimización frontal. Otros filtros incluyen la compilación de archivos *JavaScript* desde archivos *CoffeeScript* y *SASS* a *CSS*.

Muchos de los filtros no hacen el trabajo directamente, sino que utilizan otras bibliotecas para hacerlo, esto es por lo que a menudo tienes que instalar ese software también. La gran ventaja de utilizar `Assetic` para invocar estas bibliotecas es que en lugar de tener que ejecutarlo manualmente cuando has trabajado en los archivos, `Assetic` se hará cargo de esto por ti y elimina por completo este paso de tu desarrollo y proceso de despliegue.

Para usar un filtro debes especificarlo en la configuración de `Assetic` ya que no están habilitados por omisión. Por ejemplo, para utilizar el Compresor de *JavaScript YUI* hay que añadir la siguiente configuración:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_js' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
    ),
));
```

A continuación, puedes especificar la utilización del filtro en la plantilla:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Puedes encontrar una guía más detallada de la configuración y uso de *filtros* `Assetic` así como detalles del modo de depuración de `Assetic` en *Cómo minimizar JavaScript y hojas de estilo con YUI Compressor* (Página 285).

3.6.3 Controlando la *URL* utilizada

Si quieres puedes controlar las *URL* que produce *Assetic*. Esto se hace desde la plantilla y es relativo a la raíz del documento público:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*'
   output='js/combined.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array(),
    array('output' => 'js/combined.js')
) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

3.6.4 Memorizando la salida

El proceso de creación de los archivos servidos puede ser bastante lento, especialmente cuando utilizas algunos de los filtros que invocan software de terceros para el trabajo real. Incluso cuando trabajas en el entorno de desarrollo se desacelera la carga de páginas, si esto se fuera a hacer cada vez, rápidamente sería frustrante. Afortunadamente en el entorno de desarrollo `Assetic` memoriza la salida para que esto no suceda, en lugar de tener que borrar la caché manualmente, sin embargo, monitorea los cambios en los *activos* y regenera los archivos según sea necesario. Esto significa que puedes trabajar en los archivos de activos y ver los resultados al cargar la página, pero sin tener que sufrir continuas cargas lentas de la página.

Para producción, en la que no vas a realizar cambios a los archivos de los activos, el rendimiento puede ser mayor, evitando el paso de la comprobación de cambios. Assetic te permite ir más allá y evitar el contacto con *Symfony2* e incluso PHP del todo al servir los archivos. Esto se hace vertiendo todos los archivos de salida con una orden de consola. Estos los puede suministrar directamente el servidor web como archivos estáticos, aumentando el rendimiento y permitiendo que el servidor web haga frente a las cabeceras de caché. La orden de la consola para verter los archivos es:

```
php app/console assetic:dump
```

Nota: Una vez que has vertido la salida tendrás que ejecutar la orden de consola de nuevo para ver los nuevos cambios. Si estás corriendo el servidor de desarrollo tendrás que eliminar los archivos a fin de empezar a permitir que `Assetic` de nuevo procese los *activos* sobre la marcha.

3.7 Cómo minimizar JavaScript y hojas de estilo con YUI Compressor

¡Yahoo! proporciona una excelente utilidad para minimizar (minify) JavaScript y hojas de estilo para que viaje por la red más rápido, el YUI Compressor. Gracias a `Assetic`, puedes tomar ventaja de esta herramienta con mucha facilidad.

3.7.1 Descargando el JAR de YUI Compressor

El YUI Compressor está escrito en Java y se distribuye como JAR. Descarga el JAR desde el sitio Yahoo! y guárdalo en `app/Resources/java/yuicompressor.jar`.

3.7.2 Configurando los filtros de YUI

Ahora debes configurar dos *filtros* `Assetic` en tu aplicación, uno para minimizar JavaScript con el compresor YUI y otro para minimizar hojas de estilo:

- *YAML*

```yaml
# app/config/config.yml
assetic:
    filters:
        yui_css:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```xml
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_css"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_css' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
```

```
            'yui_js' => array(
                'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
            ),
        ),
    ));
```

Ahora tienes acceso a dos nuevos *filtros* `Assetic` en tu aplicación: `yui_css` y `yui_js`. Estos utilizan el compresor de YUI para minimizar hojas de estilo y JavaScripts, respectivamente.

3.7.3 Minimizando tus activos

Ahora tienes configurado el compresor YUI, pero nada va a pasar hasta que apliques uno de estos filtros a un activo. Dado que tus activos son una parte de la capa de la vista, este trabajo se hace en tus plantillas:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Nota: El ejemplo anterior asume que tienes un paquete llamado `AcmeFooBundle` y tus archivos *JavaScript* están bajo el directorio `Resources/public/js` de tu paquete. No obstante, esto no es importante - puedes incluir tus archivos *Javascript* sin importar donde se encuentren.

Con la incorporación del filtro `yui_js` a las etiquetas de los activos anteriores, ahora deberías ver llegar tus JavaScripts minimizados a través del cable mucho más rápido. Puedes repetir el mismo proceso para minimizar tus hojas de estilo.

- *Twig*

```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='yui_css' %}
<link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
{% endstylesheets %}
```

- *PHP*

```
<?php foreach ($view['assetic']->stylesheets(
    array('@AcmeFooBundle/Resources/public/css/*'),
    array('yui_css')) as $url): ?>
<link rel="stylesheet" type="text/css" media="screen" href="<?php echo $view->escape($url) ?>" /
<?php endforeach; ?>
```

3.7.4 Desactivando la minimización en modo de depuración

El JavaScript y las hojas de estilo minimizadas son muy difíciles de leer, y mucho más de depurar. Debido a esto, `Assetic` te permite desactivar un determinado *filtro* cuando la aplicación está en modo de depuración. Para ello,

puedes prefijar el nombre del *filtro* en tu plantilla con un signo de interrogación: `?`. Esto le dice a `Assetic` que aplique este filtro sólo cuando el modo de depuración está desactivado.

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('?yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

3.8 Cómo utilizar `Assetic` para optimizar imágenes con funciones *Twig*

Entre sus muchos filtros, `Assetic` tiene cuatro filtros que puedes utilizar para optimizar imágenes al vuelo. Esto te permite obtener el beneficio de archivos de menor tamaño sin tener que usar un editor de imágenes para procesar cada imagen. Los resultados se almacenan en caché y se puede vaciar en producción para que no haya impacto en el rendimiento para los usuarios finales.

3.8.1 Usando *Jpegoptim*

Jpegoptim es una utilidad para la optimización de archivos JPEG. Para usarlo con `assetic`, añade lo siguiente a la configuración de `assetic`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: ruta/a/jpegoptim
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="ruta/a/jpegoptim" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'ruta/a/jpegoptim',
        ),
```

```
    ),
));
```

Nota: Ten en cuenta que al usar *jpegoptim*, ya lo debes tener instalado en tu sistema. La opción `bin` apunta a la ubicación de los binarios compilados.

Ahora lo puedes utilizar desde una plantilla:

- *Twig*

```
{% image '@AcmeFooBundle/Resources/public/images/ejemplo.jpg'
    filter='jpegoptim' output='/images/ejemplo.jpg'
%}
<img src="{{ asset_url }}" alt="Ejemplo"/>
{% endimage %}
```

- *PHP*

```
<?php foreach ($view['assetic']->images(
    array('@AcmeFooBundle/Resources/public/images/ejemplo.jpg'),
    array('jpegoptim')) as $url): ?>
<img src="<?php echo $view->escape($url) ?>" alt="Ejemplo"/>
<?php endforeach; ?>
```

Eliminando todos los datos `EXIF`

De manera predeterminada, al ejecutar este filtro sólo eliminas parte de la metainformación almacenada en el archivo. Todos los datos `EXIF` y comentarios no se eliminan, pero los puedes quitar usando la opción `strip_all`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: ruta/a/jpegoptim
            strip_all: true
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="ruta/a/jpegoptim"
        strip_all="true" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'ruta/a/jpegoptim',
            'strip_all' => 'true',
        ),
```

```
    ),
));
```

Reduciendo la calidad máxima

El nivel de calidad del *JPEG* de manera predeterminada no se ve afectado. Puedes obtener mayor reducción de tamaño del archivo estableciendo la configuración de calidad máxima más baja que el nivel actual de las imágenes. Esto, por supuesto, a expensas de la calidad de la imagen:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: ruta/a/jpegoptim
            max: 70
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="ruta/a/jpegoptim"
        max="70" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'ruta/a/jpegoptim',
            'max' => '70',
        ),
    ),
));
```

3.8.2 Sintaxis más corta: función *Twig*

Si estás utilizando *Twig*, es posible lograr todo esto con una sintaxis más corta habilitando y utilizando una función especial de *Twig*. Comienza por agregar la siguiente configuración:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: ruta/a/jpegoptim
    twig:
        functions:
            jpegoptim: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="ruta/a/jpegoptim" />
    <assetic:twig>
        <assetic:twig_function
            name="jpegoptim" />
    </assetic:twig>
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'ruta/a/jpegoptim',
        ),
    ),
    'twig' => array(
        'functions' => array('jpegoptim'),
        ),
    ),
));
```

Ahora,puedes cambiar la plantilla de *Twig* a lo siguiente:

```
<img src="{{ jpegoptim('@AcmeFooBundle/Resources/public/images/ejemplo.jpg') }}"
     alt="Ejemplo"/>
```

Puedes especificar el directorio de salida en la configuración de la siguiente manera:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: ruta/a/jpegoptim
    twig:
        functions:
            jpegoptim: { output: images/*.jpg }
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="ruta/a/jpegoptim" />
    <assetic:twig>
        <assetic:twig_function
            name="jpegoptim"
            output="images/*.jpg" />
    </assetic:twig>
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'ruta/a/jpegoptim',
        ),
    ),
    'twig' => array(
        'functions' => array(
            'jpegoptim' => array(
                output => 'images/*.jpg'
            ),
        ),
    ),
));
```

3.9 Cómo aplicar un *filtro* `Assetic` a una extensión de archivo especifica

Los *filtros* `Assetic` se pueden aplicar a archivos individuales, grupos de archivos o incluso, como veremos aquí, a archivos que tengan una determinada extensión. Para mostrarte cómo manejar cada opción, vamos a suponer que quieres usar el filtro `CoffeeScript` de `Assetic`, el cual compila archivos de `CoffeeScript` en *Javascript*.

La configuración principal sólo son las rutas a `coffee` y `node`. Estas por omisión son `/usr/bin/coffee` y `/usr/bin/node` respectivamente:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
        ),
    ),
));
```

3.9.1 Filtrando un solo archivo

Ahora puedes servir un solo archivo `CoffeeScript` como *JavaScript* dentro de tus plantillas:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/ejemplo.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/ejemplo.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

Esto es todo lo que se necesita para compilar este archivo `CoffeeScript` y servirlo como *JavaScript* compilado.

3.9.2 Filtrando múltiples archivos

También puedes combinar varios archivos `CoffeeScript` y producir un único archivo:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/ejemplo.coffee'
               '@AcmeFooBundle/Resources/public/js/otro.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/ejemplo.coffee',
          '@AcmeFooBundle/Resources/public/js/otro.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

Ahora, ambos archivos se sirven como un solo archivo compilado en *JavaScript* regular.

3.9.3 Filtrando en base a la extensión de archivo

Una de las grandes ventajas de usar `Assetic` es minimizar el número de archivos de activos para reducir las peticiones `HTTP`. Con el fin de usar esto completamente, sería bueno combinar *todos* los archivos *JavaScript* y `CoffeeScript` juntos puesto que en última instancia, todo se debe servir como *JavaScript*. Desafortunadamente sólo añadir los archivos *JavaScript* a los archivos combinados como el anterior no funciona puesto que los archivos *JavaScript* regulares no sobrevivirán a la compilación de `CoffeeScript`.

Este problema se puede evitar usando la opción `apply_to` en la configuración, lo cual te permite especificar que siempre se aplique un *filtro* a las extensiones de archivo en particular. En este caso puedes especificar que el *filtro* `Coffee` se aplique a todos los archivos `.coffee`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            apply_to: "\.coffee$"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node"
        apply_to="\.coffee$" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
            'apply_to' => '\.coffee$',
        ),
    ),
));
```

Con esto, ya no tendrás que especificar el *filtro* `coffee` en la plantilla. También puedes listar archivos *JavaScript* regulares, los cuales serán combinados y reproducidos como un único archivo *JavaScript* (con sólo ejecutar los archivos `.coffee` a través del *filtro* `CoffeeScript`.)

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/ejemplo.coffee'
               '@AcmeFooBundle/Resources/public/js/otro.coffee'
               '@AcmeFooBundle/Resources/public/js/regular.js'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/ejemplo.coffee',
          '@AcmeFooBundle/Resources/public/js/otro.coffee',
          '@AcmeFooBundle/Resources/public/js/regular.js'),
    as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

3.10 Cómo manejar archivos subidos con *Doctrine*

Manejar el envío de archivos con entidades *Doctrine* no es diferente a la manipulación de cualquier otra carga de archivo. En otras palabras, eres libre de mover el archivo en tu controlador después de manipular el envío de un formulario. Para ver ejemplos de cómo hacerlo, consulta el *Tipo de campo file* (Página 480) en la referencia.

Si lo deseas, también puedes integrar la carga de archivos en el ciclo de vida de tu entidad (es decir, creación, actualización y eliminación). En este caso, ya que tu entidad es creada, actualizada y eliminada desde *Doctrine*, el proceso de carga y remoción de archivos se llevará a cabo de forma automática (sin necesidad de hacer nada en el controlador);

Para que esto funcione, tendrás que hacerte cargo de una serie de detalles, los cuales serán cubiertos en este artículo del recetario.

3.10.1 Configuración básica

En primer lugar, crea una sencilla clase Entidad de *Doctrine* con la cual trabajar:

```
// src/Acme/DemoBundle/Entity/Document.php
namespace Acme\DemoBundle\Entity;

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

/**
 * @ORM\Entity
 */
class Documento
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    public $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank
     */
    public $nombre;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    public $ruta;

    public function getAbsolutePath()
    {
        return null === $this->ruta ? null : $this->getUploadRootDir().'/'.$this->ruta;
    }

    public function getWebPath()
    {
        return null === $this->ruta ? null : $this->getUploadDir().'/'.$this->ruta;
    }

    protected function getUploadRootDir()
```

```
    {
        // la ruta absoluta al directorio dónde se deben guardar los documentos cargados
        return __DIR__.'/../../../../web/'.$this->getUploadDir();
    }

    protected function getUploadDir()
    {
        // se libra del __DIR__ para no desviarse al mostrar 'doc/image' en la vista.
        return 'cargas/documentos';
    }
}
```

La entidad `Documento` tiene un nombre y está asociado con un archivo. La propiedad `ruta` almacena la ruta relativa al archivo y se persiste en la base de datos. El `getAbsolutePath()` es un método útil que devuelve la ruta absoluta al archivo, mientras que `getWebPath()` es un conveniente método que devuelve la ruta web, la cual se utiliza en una plantilla para enlazar el archivo cargado.

Truco: Si no lo has hecho, probablemente primero deberías leer el tipo *archivo* (Página 480) en la documentación para comprender cómo trabaja el proceso de carga básico.

Nota: Si estás utilizando anotaciones para especificar tus reglas de anotación (como muestra este ejemplo), asegúrate de que has habilitado la validación por medio de anotaciones (consulta *configurando la validación* (Página 152)).

Para manejar el archivo subido real en el formulario, utiliza un campo `file` "virtual". Por ejemplo, si estás construyendo tu formulario directamente en un controlador, podría tener este aspecto:

```
public function uploadAction()
{
    // ...

    $formulario = $this->createFormBuilder($document)
        ->add('nombre')
        ->add('file')
        ->getForm()
    ;

    // ...
}
```

A continuación, crea esta propiedad en tu clase `Documento` y agrega algunas reglas de validación:

```
// src/Acme/DemoBundle/Entity/Document.php

// ...
class Document
{
    /**
     * @Assert\File(maxSize="6000000")
     */
    public $file;

    // ...
}
```

Nota: Debido a que estás utilizando la restricción `File`, *Symfony2* automáticamente supone que el campo del formu-

lario es una entrada de carga archivo. Es por eso que no lo tienes que establecer explícitamente al crear el formulario anterior (`->add('file')`).

El siguiente controlador muestra cómo manipular todo el proceso:

```
use Acme\DemoBundle\Entity\Document;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
// ...

/**
 * @Template()
 */
public function uploadAction()
{
    $document = new Document();
    $formulario = $this->createFormBuilder($document)
        ->add('nombre')
        ->add('file')
        ->getForm()
    ;

    if ($this->getRequest()->getMethod() === 'POST') {
        $formulario->bindRequest($this->getRequest());
        if ($formulario->isValid()) {
            $em = $this->getDoctrine()->getEntityManager();

            $em->persist($document);
            $em->flush();

            $this->redirect($this->generateUrl('...'));
        }
    }

    return array('form' => $formulario->createView());
}
```

Nota: Al escribir la plantilla, no olvides fijar el atributo `enctype`:

```
<h1>Subir Archivo</h1>

<form action="#" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Cargar Documento" />
</form>
```

El controlador anterior automáticamente persistirá la entidad `Documento` con el nombre presentado, pero no hará nada sobre el archivo y la propiedad `path` quedará en blanco.

Una manera fácil de manejar la carga de archivos es que lo muevas justo antes de que se persista la entidad y a continuación, establece la propiedad `path` en consecuencia. Comienza por invocar a un nuevo método `upload()` en la clase `Documento`, el cual deberás crear en un momento para manejar la carga del archivo:

```
if ($formulario->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();

    $document->upload();
```

```
    $em->persist($document);
    $em->flush();

    $this->redirect('...');
}
```

El método `upload()` tomará ventaja del objeto `Symfony\Component\HttpFoundation\File\UploadedFile`, el cual es lo que devuelve después de que se presenta un campo `file`:

```
public function upload()
{
    // la propiedad 'file' puede estar vacía si el campo no es obligatorio
    if (null === $this->file) {
        return;
    }

    // aquí utilizamos el nombre de archivo original pero lo deberías
    // desinfectar por lo menos para evitar cualquier problema de seguridad

    // 'move' toma el directorio y nombre de archivo destino al cual trasladarlo
    $this->file->move($this->getUploadRootDir(), $this->file->getClientOriginalName());

    // fija la propiedad 'path' al nombre de archivo donde se guardó el archivo
    $this->setPath($this->file->getClientOriginalName());

    // limpia la propiedad 'file' ya que no la necesitas más
    $this->file = null;
}
```

3.10.2 Usando el ciclo de vida de las retrollamadas

Incluso si esta implementación trabaja, adolece de un defecto importante: ¿Qué pasa si hay un problema al persistir la entidad? El archivo ya se ha movido a su ubicación final, incluso aunque la propiedad `path` de la entidad no se persista correctamente.

Para evitar estos problemas, debes cambiar la implementación para que la operación de base de datos y el traslado del archivo sean atómicos: si hay un problema al persistir la entidad o si el archivo no se puede mover, entonces, no debe suceder *nada*.

Para ello, es necesario mover el archivo justo cuando *Doctrine* persista la entidad a la base de datos. Esto se puede lograr enganchando el ciclo de vida de la entidad a una retrollamada:

```
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
}
```

A continuación, reconstruye la clase `Documento` para que tome ventaja de estas retrollamadas:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
```

```
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            // haz lo que quieras para generar un nombre único
            $this->setPath(uniqid().'.'.$this->file->guessExtension());
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->archivo) {
            return;
        }

        // aquí debes lanzar una excepción si el archivo no se puede mover
        // para que la entidad no se persista en la base de datos
        // lo cual hace automáticamente el método move() del archivo subido
        $this->file->move($this->getUploadRootDir(), $this->path);

        unset($this->file);
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }
}
```

La clase ahora hace todo lo que necesitas: genera un nombre de archivo único antes de persistirlo, mueve el archivo después de persistirlo y elimina el archivo si la entidad es eliminada.

Nota: Los eventos retrollamados `@ORM\PrePersist()` y `@ORM\PostPersist()` se disparan antes y después de almacenar la entidad en la base de datos. Por otro lado, los eventos retrollamados `@ORM\PreUpdate()` y `@ORM\PostUpdate()` se llama al actualizar la entidad.

Prudencia: Las retrollamadas `PreUpdate` y `PostUpdate` sólo se activan si se persiste algún cambio en uno de los campos de la entidad. Esto significa que, de manera predeterminada, si sólo modificas la propiedad `$archivo`, estos eventos no se activarán, puesto que esa propiedad no se persiste directamente a través de *Doctrine*. Una solución sería usar un campo `actualizado` que *Doctrine* persista, y modificarlo manualmente al cambiar el archivo.

3.10.3 Usando el `id` como nombre de archivo

Si deseas utilizar el `id` como el nombre del archivo, la implementación es un poco diferente conforme sea necesaria para guardar la extensión en la propiedad `path`, en lugar del nombre de archivo real:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            $this->setPath($this->file->guessExtension());
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->archivo) {
            return;
        }

        // aquí debes lanzar una excepción si el archivo no se puede mover
        // para que la entidad no se conserve en la base de datos
        // lo cual hace el método move() del archivo subido
        $this->file->move($this->getUploadRootDir(), $this->id.'.'.$this->file->guessExtension());

        unset($this->file);
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }

    public function getAbsolutePath()
    {
        return null === $this->ruta ? null : $this->getUploadRootDir().'/'.$this->id.'.'.$this->path;
    }
}
```

3.11 Extensiones *Doctrine*: `Timestampable`: `Sluggable`, `Translatable`, etc.

Doctrine2 es muy flexible, y la comunidad ya ha creado una serie de útiles extensiones *Doctrine* para ayudarte con las tareas habituales, tareas relacionadas con la entidad.

Un paquete en particular - DoctrineExtensionsBundle - proporciona integración con una biblioteca de extensiones que ofrecen comportamientos Sluggable, Translatable, Timestampable, Loggable y Tree.

Ve el paquete para más detalles.

3.12 Registrando escuchas y suscriptores de eventos

Doctrine cuenta con un rico sistema de eventos que lanza eventos en casi todo lo que sucede dentro del sistema. Para ti, esto significa que puedes crear *servicios* (Página 237) arbitrarios y decirle a *Doctrine* que los notifique a los objetos cada vez que ocurra una determinada acción (por ejemplo, `preSave`) dentro de *Doctrine*. Esto podría ser útil, por ejemplo, para crear un índice de búsqueda independiente cuando se guarde un objeto en tu base de datos.

Doctrine define dos tipos de objetos que pueden escuchar los eventos de *Doctrine*: escuchas y suscriptores. Ambos son muy similares, pero los escuchas son un poco más sencillos. Para más información, consulta el Sistema de eventos en el sitio web de *Doctrine*.

3.12.1 Configurando escuchas/suscriptores

Para registrar un servicio para que actúe como un escucha o suscriptor de eventos sólo lo tienes que *etiquetar* (Página 251) con el nombre apropiado. Dependiendo de tu caso de uso, puedes enganchar un escucha en cada conexión *DBAL* y gestor de entidad *ORM* o simplemente en una conexión *DBAL* específica y todos los gestores de entidad que utilicen esta conexión.

- *YAML*

```
doctrine:
    dbal:
        default_connection: default
        connections:
            default:
                driver: pdo_sqlite
                memory: true

services:
    mi.escucha:
        class: Acme\SearchBundle\Listener\SearchIndexer
        tags:
            - { name: doctrine.event_listener, event: postSave }
    mi.escucha2:
        class: Acme\SearchBundle\Listener\SearchIndexer2
        tags:
            - { name: doctrine.event_listener, event: postSave, connection: default }
    mi.suscriptor:
        class: Acme\SearchBundle\Listener\SearchIndexerSubsriber
        tags:
            - { name: doctrine.event_subscriber, connection: default }
```

- *XML*

```xml
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine">

    <doctrine:config>
        <doctrine:dbal default-connection="default">
            <doctrine:connection driver="pdo_sqlite" memory="true" />
        </doctrine:dbal>
    </doctrine:config>

    <services>
        <service id="mi.escucha" class="Acme\SearchBundle\Listener\SearchIndexer">
            <tag name="doctrine.event_listener" event="postSave" />
        </service>
        <service id="mi.escucha2" class="Acme\SearchBundle\Listener\SearchIndexer2">
            <tag name="doctrine.event_listener" event="postSave" connection="default" />
        </service>
        <service id="mi.suscriptor" class="Acme\SearchBundle\Listener\SearchIndexerSubsriber">
            <tag name="doctrine.event_subscriber" connection="default" />
        </service>
    </services>
</container>
```

3.12.2 Creando la clase Escucha

En el ejemplo anterior, configuramos un servicio `mi.escucha` como un escucha del evento `postSave` de *Doctrine*. El cual detrás de esa clase servicio debe tener un método `postSave`, que se llamará cuando sea lanzado el evento:

```
// src/Acme/SearchBundle/Listener/SearchIndexer.php
namespace Acme\SearchBundle\Listener;

use Doctrine\ORM\Event\LifecycleEventArgs;

class SearchIndexer
{
    public function postSave(LifecycleEventArgs $args)
    {
        $entidad = $args->getEntity();
        $em = $args->getEntityManager();

        // tal vez sólo quieres actuar en alguna entidad "Producto"
        if ($entidad instanceof Acme\TiendaBundle\Entity\Producto) {
            // haz algo con el producto
        }
    }
}
```

En cada caso, tienes acceso a un objeto `LifecycleEventArgs`, el cual te da acceso tanto al objeto entidad del evento como al gestor de la entidad misma.

Una cosa importante a resaltar es que un escucha debe estar atento a *todas* las entidades en tu aplicación. Por lo tanto, si estás interesado sólo en manejar un tipo de entidad específico (por ejemplo, una entidad `Producto`, pero no en una entidad `BlogComunicado`), debes verificar el nombre de clase de la entidad en tu método (como arriba).

3.13 Cómo generar entidades de una base de datos existente

Cuando empiezas a trabajar en el proyecto de una nueva marca que utiliza una base de datos, es algo natural que sean dos situaciones diferentes. En la mayoría de los casos, el modelo de base de datos está diseñado y construido desde cero. A veces, sin embargo, comenzará con un modelo de base de datos existente y probablemente inmutable. Afortunadamente, *Doctrine* viene con un montón de herramientas para ayudarte a generar las clases del modelo desde tu base de datos existente.

Nota: Como dicen las herramientas de documentación de Doctrine, la ingeniería inversa es un proceso de una sola vez para empezar a trabajar en un proyecto. *Doctrine* es capaz de convertir aproximadamente el 70-80 % de la información asignada basándose en los campos, índices y restricciones de clave externa. *Doctrine* no puede descubrir asociaciones inversas, tipos de herencia, entidades con claves externas como claves principales u operaciones semánticas en asociaciones tales como eventos en cascada o ciclo de vida de los eventos. Posteriormente, será necesario algún trabajo adicional sobre las entidades generadas para diseñar cada una según tus características específicas del modelo de dominio.

Esta guía asume que estás usando una simple aplicación de blog con las siguientes dos tablas: `blog_post` y `blog_coment`. Un registro de comentarios está vinculado a un registro de comentario gracias a una restricción de clave externa.

```
CREATE TABLE `blog_post` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `titulo` varchar(100) COLLATE utf8_unicode_ci NOT NULL,
  `contenido` longtext COLLATE utf8_unicode_ci NOT NULL,
  `creado_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

CREATE TABLE `blog_comment` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) NOT NULL,
  `autor` varchar(20) COLLATE utf8_unicode_ci NOT NULL,
  `contenido` longtext COLLATE utf8_unicode_ci NOT NULL,
  `creado_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `blog_comment_post_id_idx` (`post_id`),
  CONSTRAINT `blog_post_id` FOREIGN KEY (`post_id`) REFERENCES `blog_post` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Antes de zambullirte en la receta, asegúrate de que los parámetros de conexión de tu base de datos están configurados correctamente en el archivo `app/config/parameters.ini` (o cualquier otro lugar donde mantienes la configuración de base de datos) y que has iniciado un paquete que será la sede de tu futura clase entidad. En esta guía, vamos a suponer que existe un `AcmeBlogBundle` y se encuentra en el directorio `src/Acme/BlogBundle`.

El primer paso para crear clases de entidad de una base de datos existente es pedir a *Doctrine* que introspeccione la base de datos y genere los archivos de metadatos correspondientes. Los archivos de metadatos describen la clase entidad para generar tablas basándose en los campos.

```
php app/console doctrine:mapping:convert xml ./src/Acme/BlogBundle/Resources/config/doctrine/metadata
```

Esta herramienta de línea de ordenes le pide a *Doctrine* que inspeccione la estructura de la base de datos y genere los archivos *XML* de metadatos bajo el directorio `src/Acme/BlogBundle/Resources/config/doctrine/metadata/orm` de tu paquete.

Truco: También es posible generar los metadatos de clase en formato *YAML* cambiando el primer argumento a *yml*.

El archivo de metadatos generado `BlogPost.dcm.xml` es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>
  <entity name="BlogPost" table="blog_post">
    <change-tracking-policy>DEFERRED_IMPLICIT</change-tracking-policy>
    <id name="id" type="bigint" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="title" type="string" column="title" length="100"/>
    <field name="contenido" type="text" column="contenido"/>
    <field name="esPublicado" type="boolean" column="es_publicado"/>
    <field name="createdAt" type="datetime" column="creado_at"/>
    <field name="updatedAt" type="datetime" column="updated_at"/>
    <field name="ficha" type="string" column="ficha" length="255"/>
    <lifecycle-callbacks/>
  </entity>
</doctrine-mapping>
```

Una vez generados los archivos de metadatos, puedes pedir a *Doctrine* que importe el esquema y construya las clases relacionadas con la entidad, ejecutando las dos siguientes ordenes.

```
php app/console doctrine:mapping:import AcmeBlogBundle annotation
php app/console doctrine:generate:entities AcmeBlogBundle
```

La primer orden genera las clases de entidad con una asignación de anotaciones, pero por supuesto puedes cambiar el argumento `annotation` a `xml` o `yml`. La clase entidad `BlogComment` recién creada se ve de la siguiente manera:

```
<?php

// src/Acme/BlogBundle/Entity/BlogComment.php
namespace Acme\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Acme\BlogBundle\Entity\BlogComment
 *
 * @ORM\Table(name="blog_comment")
 * @ORM\Entity
 */
class BlogComment
{
    /**
     * @var bigint $id
     *
     * @ORM\Column(name="id", type="bigint", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

    /**
     * @var string $autor
     *
     * @ORM\Column(name="autor", type="string", length=100, nullable=false)
     */
    private $autor;
```

```
    /**
     * @var text $contenido
     *
     * @ORM\Column(name="contenido", type="text", nullable=false)
     */
    private $contenido;

    /**
     * @var datetime $createdAt
     *
     * @ORM\Column(name="creado_at", type="datetime", nullable=false)
     */
    private $createdAt;

    /**
     * @var BlogPost
     *
     * @ORM\ManyToOne(targetEntity="BlogPost")
     * @ORM\JoinColumn(name="post_id", referencedColumnName="id")
     */
    private $comunicado;
}
```

Como puedes ver, *Doctrine* convierte todos los campos de la tabla a propiedades privadas puras y anotaciones de clase. Lo más impresionante es que también descubriste la relación con la clase entidad `BlogPost` basándote en la restricción de la clave externa. Por lo tanto, puedes encontrar una propiedad privada `$comunicado` asignada a una entidad `BlogPost` en la clase entidad `BlogComment`.

La última orden genera todos los captadores y definidores de tus dos propiedades de la clase entidad `BlogPost` y `BlogComment`. Las entidades generadas ahora están listas para utilizarse. ¡Que te diviertas!

3.14 Cómo utiliza *Doctrine* la capa *DBAL*

Nota: Este artículo es sobre la capa DBAL de *Doctrine*. Normalmente, vas a trabajar con el nivel superior de la capa *ORM* de *Doctrine*, la cual simplemente utiliza DBAL detrás del escenario para comunicarse realmente con la base de datos. Para leer más sobre el *ORM* de *Doctrine*, consulta "*Bases de datos y Doctrine ("El modelo")* (Página 115)".

Doctrine la capa de abstracción de base de datos (DataBase Abstraction Layer - DBAL) es una capa que se encuentra en la parte superior de PDO y ofrece una *API* intuitiva y flexible para comunicarse con las bases de datos relacionales más populares. En otras palabras, la biblioteca DBAL facilita la ejecución de consultas y realización de otras acciones de bases de datos.

Truco: Lee la documentación oficial de DBAL para conocer todos los detalles y las habilidades de la biblioteca *DBAL* de *Doctrine*.

Para empezar, configura los parámetros de conexión a la base de datos:

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        driver:   pdo_mysql
        dbname:   Symfony2
```

```
        user:     root
        password: null
        charset:  UTF8
```

- *XML*

```
// app/config/config.xml
<doctrine:config>
    <doctrine:dbal
        name="default"
        dbname="Symfony2"
        user="root"
        password="null"
        driver="pdo_mysql"
    />
</doctrine:config>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('doctrine', array(
    'dbal' => array(
        'driver'    => 'pdo_mysql',
        'dbname'    => 'Symfony2',
        'user'      => 'root',
        'password'  => null,
    ),
));
```

Para ver todas las opciones de configuración DBAL, consulta *Configurando DBAL Doctrine* (Página 450).

A continuación, puedes acceder a la conexión *Doctrine DBAL* accediendo al servicio `database_connection`:

```
class UserController extends Controller
{
    public function indexAction()
    {
        $conn = $this->get('database_connection');
        $usuarios = $conn->fetchAll('SELECT * FROM users');

        // ...
    }
}
```

3.14.1 Registrando tipos de asignación personalizados

Puedes registrar tipos de asignación personalizados a través de la configuración de *Symfony*. Ellos se sumarán a todas las conexiones configuradas. Para más información sobre los tipos de asignación personalizados, lee la sección Tipos de asignación personalizados de la documentación de *Doctrine*.

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        types:
            custom_first: Acme\HolaBundle\Type\CustomFirst
            custom_second: Acme\HolaBundle\Type\CustomSecond
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:dbal>
        <doctrine:dbal default-connection="default">
            <doctrine:connection>
                <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
            </doctrine:connection>
        </doctrine:dbal>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('doctrine', array(
    'dbal' => array(
        'connections' => array(
            'default' => array(
                'mapping_types' => array(
                    'enum'  => 'string',
                ),
            ),
        ),
    ),
));
```

3.14.2 Registrando tipos de asignación personalizados en SchemaTool

La SchemaTool se utiliza al inspeccionar la base de datos para comparar el esquema. Para lograr esta tarea, es necesario saber qué tipo de asignación se debe utilizar para cada tipo de la base de datos. Por medio de la configuración puedes registrar nuevos tipos.

Vamos a asignar el tipo ENUM (no apoyado por DBAL de manera predeterminada) al tipo `string`:

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        connection:
            default:
                // otros parámetros de conexión
                mapping_types:
                    enum: string
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
```

```
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

        <doctrine:config>
            <doctrine:dbal>
                <doctrine:type name="custom_first" class="Acme\HolaBundle\Type\CustomFirst" />
                <doctrine:type name="custom_second" class="Acme\HolaBundle\Type\CustomSecond" />
            </doctrine:dbal>
        </doctrine:config>
    </container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('doctrine', array(
    'dbal' => array(
        'types' => array(
            'custom_first'  => 'Acme\HolaBundle\Type\CustomFirst',
            'custom_second' => 'Acme\HolaBundle\Type\CustomSecond',
        ),
    ),
));
```

3.15 Cómo trabajar con varios gestores de entidad

En una aplicación *Symfony2* puedes utilizar múltiples gestores de entidad. Esto es necesario si estás utilizando diferentes bases de datos e incluso proveedores con conjuntos de entidades totalmente diferentes. En otras palabras, un gestor de entidad que se conecta a una base de datos deberá administrar algunas entidades, mientras que otro gestor de entidad conectado a otra base de datos puede manejar el resto.

Nota: Usar varios gestores de entidad es bastante fácil, pero más avanzado y generalmente no se requiere. Asegúrate de que realmente necesitas varios gestores de entidad antes de añadir complejidad a ese nivel.

El siguiente código de configuración muestra cómo puedes configurar dos gestores de entidad:

- *YAML*

```
doctrine:
    orm:
        default_entity_manager:   default
        entity_managers:
            default:
                connection:       default
                mappings:
                    AcmeDemoBundle: ~
                    AcmeGuardaBundle: ~
            customer:
                connection:       customer
                mappings:
                    AcmeCustomerBundle: ~
```

En este caso, hemos definido dos gestores de entidad y los llamamos `default` y `customer`. El gestor de entidad `default` administra cualquier entidad en los paquetes `AcmeDemoBundle` y `AcmeGuardaBundle`, mientras que el gestor de entidad `customer` gestiona cualquiera en el paquete `AcmeCustomerBundle`.

Cuando trabajas con tus gestores de entidad, entonces debes ser explícito acerca de cual gestor de entidad deseas. Si *no* omites el nombre del gestor de entidad al consultar por él, se devuelve el gestor de entidad predeterminado (es decir, `default`):

```
class UserController extends Controller
{
    public function indexAction()
    {
        // ambas devuelven el em "default"
        $em = $this->get('doctrine')->getEntityManager();
        $em = $this->get('doctrine')->getEntityManager('default');

        $customerEm =  $this->get('doctrine')->getEntityManager('customer');
    }
}
```

Ahora puedes utilizar *Doctrine* tal como lo hiciste antes - con el gestor de entidad `default` para persistir y recuperar las entidades que gestiona y el gestor de entidad `customer` para persistir y recuperar sus entidades.

3.16 Registrando funciones `DQL` personalizadas

Doctrine te permite especificar las funciones `DQL` personalizadas. Para más información sobre este tema, lee el artículo "DQL Funciones definidas por el usuario" de *Doctrine*.

En *Symfony*, puedes registrar tus funciones `DQL` personalizadas de la siguiente manera:

- *YAML*

```
# app/config/config.yml
doctrine:
    orm:
        # ...
        entity_managers:
            default:
                # ...
                dql:
                    string_functions:
                        test_string: Acme\HolaBundle\DQL\StringFunction
                        second_string: Acme\HolaBundle\DQL\SecondStringFunction
                    numeric_functions:
                        test_numeric: Acme\HolaBundle\DQL\NumericFunction
                    datetime_functions:
                        test_datetime: Acme\HolaBundle\DQL\DatetimeFunction
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:orm>
            <!-- ... -->
            <doctrine:entity-manager name="default">
                <!-- ... -->
```

```
                <doctrine:dql>
                    <doctrine:string-function name="test_string>Acme\HolaBundle\DQL\StringFuncti
                    <doctrine:string-function name="second_string>Acme\HolaBundle\DQL\SecondStri
                    <doctrine:numeric-function name="test_numeric>Acme\HolaBundle\DQL\NumericFun
                    <doctrine:datetime-function name="test_datetime>Acme\HolaBundle\DQL\Datetime
                </doctrine:dql>
            </doctrine:entity-manager>
        </doctrine:orm>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('doctrine', array(
    'orm' => array(
        // ...
        'entity_managers' => array(
            'default' => array(
                // ...
                'dql' => array(
                    'string_functions' => array(
                        'test_string'   => 'Acme\HolaBundle\DQL\StringFunction',
                        'second_string' => 'Acme\HolaBundle\DQL\SecondStringFunction',
                    ),
                    'numeric_functions' => array(
                        'test_numeric' => 'Acme\HolaBundle\DQL\NumericFunction',
                    ),
                    'datetime_functions' => array(
                        'test_datetime' => 'Acme\HolaBundle\DQL\DatetimeFunction',
                    ),
                ),
            ),
        ),
    ),
));
```

3.17 Cómo personalizar la reproducción de un formulario

Symfony ofrece una amplia variedad de formas para personalizar cómo se reproduce un formulario. En esta guía, aprenderás cómo personalizar cada parte posible de tu formulario con el menor esfuerzo posible si utilizas *Twig* o PHP como tu motor de plantillas.

3.17.1 Fundamentos de la reproducción de formularios

Recuerda que `label`, error y los elementos gráficos *HTML* de un campo de formulario se pueden reproducir fácilmente usando la función `form_row` de *Twig* o el método ayudante `row` de *PHP*:

- *Twig*

```
{{ form_row(form.edad) }}
```

- *PHP*

```
<?php echo $view['form']->row($formulario['edad']) }} ?>
```

También puedes reproducir cada una de las tres partes del campo individualmente:

- *Twig*

```
<div>
    {{ form_label(form.edad) }}
    {{ form_errors(form.edad) }}
    {{ form_widget(form.edad) }}
</div>
```

- *PHP*

```
<div>
    <?php echo $view['form']->label($form['edad']) }} ?>
    <?php echo $view['form']->errors($form['edad']) }} ?>
    <?php echo $view['form']->widget($form['edad']) }} ?>
</div>
```

En ambos casos, la etiqueta, errores y elementos gráficos del formulario *HTML* se reproducen con un conjunto de marcas que se incluyen de serie con *Symfony*. Por ejemplo, ambas plantillas anteriores reproducirán:

```
<div>
    <label for="form_edad">Edad</label>
    <ul>
        <li>Este campo es obligatorio</li>
    </ul>
    <input type="number" id="form_edad" name="form[edad]" />
</div>
```

para crear prototipos rápidamente y probar un formulario, puedes reproducir el formulario completo con una sola línea:

- *Twig*

```
{{ form_widget(form) }}
```

- *PHP*

```
<?php echo $view['form']->widget($formulario) }} ?>
```

El resto de esta receta debe explicar cómo se puede modificar cada parte del marcado del formulario en varios niveles diferentes. Para más información sobre la forma de reproducción en general, consulta *Reproduciendo un formulario en una plantilla* (Página 169).

3.17.2 ¿Qué son los temas de formulario?

Symfony utiliza fragmentos de formulario - una parte de una plantilla que sólo reproduce una pequeña parte de un formulario - para reproducir todas las partes de un formulario - - etiquetas de campo, errores, campos de texto `input`, etiquetas `select`, etc.

Los fragmentos se definen como bloques en *Twig* y como archivos de plantilla en *PHP*.

A *tema* no es más que un conjunto de fragmentos que deseas utilizar al reproducir un formulario. En otras palabras, si deseas personalizar una parte de cómo reproducir un formulario, importa el *tema* que contiene una personalización apropiada de los fragmentos del formulario.

Symfony viene con un tema predeterminado (form_div_base.html.twig en *Twig* y `FrameworkBundle:Form` en *PHP*) que define todos y cada uno de los fragmentos necesarios para reproducir todas las partes de un formulario.

En la siguiente sección aprenderás cómo personalizar un tema redefiniendo todos o algunos de sus fragmentos.

Por ejemplo, cuando reproduces el elemento gráfico de un campo de tipo `entero`, se genera un campo `input` como `número`.

- *Twig*

```
{{ form_widget(form.edad) }}
```

- *PHP*

```
<?php echo $view['form']->widget($formulario['edad']) ?>
```

reproduce:

```
<input type="number" id="form_edad" name="form[edad]" required="required" value="33" />
```

Internamente, *Symfony* utiliza el fragmento `integer_widget` para reproducir el campo. Esto se debe a que el tipo de campo es `entero` y estás reproduciendo su `elemento gráfico` (en comparación a `label` o `errores`).

En *Twig* de manera predeterminada en el bloque `integer_widget` de la plantilla form_div_base.html.twig.

En PHP sería más bien el archivo `integer_widget.html.php` ubicado en el directorio `FrameworkBundle/Resources/views/Form`.

La implementación predeterminada del fragmento `integer_widget` tiene el siguiente aspecto:

- *Twig*

```
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('field_widget') }}
{% endblock integer_widget %}
```

- *PHP*

```
<!-- integer_widget.html.php -->

<?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nu
```

Como puedes ver, este fragmento reproduce otro fragmento - `field_widget`:

- *Twig*

```
{% block field_widget %}
    {% set type = type|default('text') %}
    <input type="{{ type }}" {{ block('widget_attributes') }} value="{{ value }}" />
{% endblock field_widget %}
```

- *PHP*

```
<!-- FrameworkBundle/Resources/views/Form/field_widget.html.php -->

<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($valor) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>
```

El punto es que los fragmentos dictan la salida *HTML* de cada parte de un formulario. Para personalizar la salida del formulario, sólo tienes que identificar y redefinir el fragmento correcto. Un conjunto de estos fragmentos personalizados del formulario se conoce como un "tema" de formulario. Al reproducir un formulario, puedes elegir el/los tema(s) que deseas aplicar al formulario.

En *Twig* un tema es un sólo archivo de plantilla y los fragmentos son los bloques definidos en ese archivo.

En PHP un tema es un directorio y los fragmentos son archivos de plantilla individuales en ese directorio.

Saber cual bloque personalizar

En este ejemplo, el nombre del fragmento personalizado es `integer_widget` debido a que deseas reemplazar el `elemento gráfico` *HTML* para todos los campos de tipo `entero`. Si necesitas personalizar los campos `textarea`, debes personalizar el `textarea_widget`.
Como puedes ver, el nombre del bloque es una combinación del tipo de campo y qué parte del campo se está reproduciendo (por ejemplo, `widget`, `label`, `errores`, `row`). Como tal, para personalizar cómo se reproducen los errores, tan sólo para campos de entrada `text`, debes personalizar el fragmento `text_errors`.
Muy comúnmente, sin embargo, deseas personalizar cómo se muestran los errores en *todos* los campos. Puedes hacerlo personalizando el fragmento `field_errors`. Este aprovecha la herencia del tipo de campo. Especialmente, ya que el tipo‘‘text‘‘ se extiende desde el tipo `field`, el componente *form* busca el bloque del tipo específico (por ejemplo, `text_errors`) antes de caer de nuevo al nombre del fragmento padre si no existe (por ejemplo, `field_errors`).
Para más información sobre este tema, consulta *Nombrando fragmentos de formulario* (Página 177).

3.17.3 Tematizando formularios

Para ver el poder del tematizado de formularios, supongamos que deseas envolver todos los campos de entrada `número` con una etiqueta `div`. La clave para hacerlo es personalizar el fragmento `text_widget`.

3.17.4 Tematizando formularios en *Twig*

Cuando personalizamos el bloque de campo de formulario en *Twig*, tienes dos opciones en *donde* puede vivir el bloque personalizado del formulario:

Método	Pros	Contras
Dentro de la misma plantilla que el formulario	Rápido y fácil	No se puede reutilizar en otra plantilla
Dentro de una plantilla separada	Se puede reutilizar en muchas plantillas	Requiere la creación de una plantilla extra

Ambos métodos tienen el mismo efecto, pero son mejores en diferentes situaciones.

Método 1: Dentro de la misma plantilla que el formulario

La forma más sencilla de personalizar el bloque `integer_widget` es personalizarlo directamente en la plantilla que realmente reproduce el formulario.

```
{% extends '::base.html.twig' %}

{% form_tema form _self %}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

```
{% block contenido %}
    {# reproduce el formulario #}

    {{ form_row(form.age) }}
{% endblock %}
```

Al usar las etiquetas especiales `{% form_theme form _self %}`, *Twig* busca dentro de la misma plantilla cualquier bloque de formulario a sustituir. Suponiendo que el campo `form.edad` es un campo de tipo `entero`, cuando se reproduzca el elemento gráfico, utilizará el bloque personalizado `integer_widget`.

La desventaja de este método es que los bloques personalizados del formulario no se pueden reutilizar en otros formularios reproducidos en otras plantillas. En otras palabras, este método es más útil cuando haces personalizaciones en forma que sean específicas a un único formulario en tu aplicación. Si deseas volver a utilizar una personalización a través de varios (o todos) los formularios de tu aplicación, lee la siguiente sección.

Método 2: Dentro de una plantilla separada

También puedes optar por poner el bloque personalizado `integer_widget` del formulario en una plantilla completamente independiente. El código y el resultado final son el mismo, pero ahora puedes volver a utilizar la personalización de un formulario a través de muchas plantillas:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

Ahora que has creado el bloque personalizado, es necesario decirle a *Symfony* que lo utilice. Dentro de la plantilla en la que estás reproduciendo tu formulario realmente, dile a *Symfony* que utilice la plantilla por medio de la etiqueta `form_theme`:

```
{% form_theme form 'AcmeDemoBundle:Form:fields.html.twig' %}

{{ form_widget(form.edad) }}
```

Cuando se reproduzca el `form.edad`, *Symfony* utilizará el bloque `integer_widget` de la nueva plantilla y la etiqueta `input` será envuelta en el elemento `div` especificado en el bloque personalizado.

3.17.5 Tematizando formularios en *PHP*

Cuando usas PHP como motor de plantillas, el único método para personalizar un fragmento es crear un nuevo archivo de plantilla - esto es similar al segundo método utilizado por *Twig*.

El archivo de plantilla se debe nombrar después del fragmento. Debes crear un archivo `integer_widget.html.php` a fin de personalizar el fragmento `integer_widget`.

```
<!-- src/Acme/DemoBundle/Resources/views/Form/integer_widget.html.php -->

<div class="integer_widget">
    <?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nur
</div>
```

Ahora que has creado la plantilla del formulario personalizado, necesitas decirlo a *Symfony* para utilizarlo. Dentro de la plantilla en la que estás reproduciendo tu formulario realmente, dile a *Symfony* que utilice la plantilla por medio del método ayudante `setTheme`:

```
<?php $view['form']->setTheme($formulario, array('AcmeDemoBundle:Form')) ;?>

<?php $view['form']->widget($formulario['edad']) ?>
```

Al reproducir el elemento gráfico `form.edad`, *Symfony* utilizará la plantilla personalizada `integer_widget.html.php` y la etiqueta `input` será envuelta en el elemento `div`.

3.17.6 Refiriendo bloques del formulario base (específico de *Twig*)

Hasta ahora, para sustituir un bloque `form` particular, el mejor método consiste en copiar el bloque predeterminado desde form_div_base.html.twig, pegarlo en una plantilla diferente y entonces, personalizarlo. En muchos casos, puedes evitarte esto refiriendo al bloque base cuando lo personalizas.

Esto se logra fácilmente, pero varía ligeramente dependiendo de si el bloque del formulario personalizado se encuentra en la misma plantilla que el formulario o en una plantilla separada.

Refiriendo bloques dentro de la misma plantilla que el formulario

Importa los bloques añadiendo una etiqueta `use` en la plantilla donde estás reproduciendo el formulario:

```
{% use 'form_div_base.html.twig' with integer_widget as base_integer_widget %}
```

Ahora, cuando importes bloques desde form_div_base.html.twig, el bloque `integer_widget` es llamado `base_integer_widget`. Esto significa que cuando redefines el bloque `integer_widget`, puedes referir el marcado predeterminado a través de `base_integer_widget`:

```
{% block integer_widget %}
    <div class="integer_widget">
        {{ block('base_integer_widget') }}
    </div>
{% endblock %}
```

Refiriendo bloques base desde una plantilla externa

Si tus personalizaciones del formulario viven dentro de una plantilla externa, puedes referir al bloque base con la función `parent()` de *Twig*:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% extends 'form_div_base.html.twig' %}

{% block integer_widget %}
    <div class="integer_widget">
        {{ parent() }}
    </div>
{% endblock %}
```

Nota: No es posible hacer referencia al bloque base cuando usas PHP como motor de plantillas. Tienes que copiar manualmente el contenido del bloque base a tu nuevo archivo de plantilla.

3.17.7 Personalizando toda tu aplicación

Si deseas que una personalización en cierto formulario sea global en tu aplicación, lo puedes lograr haciendo las personalizaciones del formulario en una plantilla externa y luego importarla dentro de la configuración de tu aplicación:

Twig

Al utilizar la siguiente configuración, los bloques personalizados del formulario dentro de la plantilla `AcmeDemoBundle:Form:fields.html.twig` se utilizarán globalmente al reproducir un formulario.

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
        <twig:form>
            <resource>AcmeDemoBundle:Form:fields.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```
// app/config/config.php

$contenedor->loadFromExtension('twig', array(
    'form' => array('resources' => array(
        'AcmeDemoBundle:Form:fields.html.twig',
     ))
    // ...
));
```

De forma predeterminada, *Twig* utiliza un diseño con *div* al reproducir formularios. Algunas personas, sin embargo, pueden preferir reproducir formularios en un diseño con *tablas*. Usa el recurso `form_table_base.html.twig` para utilizarlo como diseño:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources: ['form_table_base.html.twig']
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->
```

```
<twig:config ...>
        <twig:form>
            <resource>form_table_base.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```
// app/config/config.php

$contenedor->loadFromExtension('twig', array(
    'form' => array('resources' => array(
        'form_table_base.html.twig',
     ))
    // ...
));
```

Si sólo quieres hacer el cambio en una plantilla, añade la siguiente línea a tu archivo de plantilla en lugar de agregar la plantilla como un recurso:

```
{% form_theme form 'form_table_base.html.twig' %}
```

Ten en cuenta que la variable `form` en el código anterior es la variable de la vista del formulario pasada a la plantilla.

PHP

Al utilizar la siguiente configuración, cualquier fragmento de formulario personalizado dentro del directorio `src/Acme/DemoBundle/Resources/views/Form` se usará globalmente al reproducir un formulario.

- *YAML*

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'AcmeDemoBundle:Form'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>AcmeDemoBundle:Form</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
// app/config/config.php

// PHP
```

```
$contenedor->loadFromExtension('framework', array(
    'templating' => array('form' =>
        array('resources' => array(
            'AcmeDemoBundle:Form',
     )))
    // ...
));
```

De manera predeterminada, el motor PHP utiliza un diseño *div* al reproducir formularios. Algunas personas, sin embargo, pueden preferir reproducir formularios en un diseño con *tablas*. Utiliza el recurso `FrameworkBundle:FormTable` para utilizar este tipo de diseño:

- *YAML*

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'FrameworkBundle:FormTable'
```

- *XML*

```
<!-- app/config/config.xml -->

<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>FrameworkBundle:FormTable</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
// app/config/config.php

$contenedor->loadFromExtension('framework', array(
    'templating' => array('form' =>
        array('resources' => array(
            'FrameworkBundle:FormTable',
     )))
    // ...
));
```

Si sólo quieres hacer el cambio en una plantilla, añade la siguiente línea a tu archivo de plantilla en lugar de agregar la plantilla como un recurso:

```
<?php $view['form']->setTheme($formulario, array('FrameworkBundle:FormTable')); ?>
```

Ten en cuenta que la variable `$formulario` en el código anterior es la variable de la vista del formulario que pasaste a tu plantilla.

3.17.8 Cómo personalizar un campo individual

Hasta ahora, hemos visto diferentes formas en que puedes personalizar elementos gráficos de todos los tipos de campo de texto. También puedes personalizar campos individuales. Por ejemplo, supongamos que tienes dos campos `text`

- nombre_de_pila y apellido -, pero sólo quieres personalizar uno de los campos. Esto se puede lograr personalizando un fragmento cuyo nombre es una combinación del atributo id del campo y cual parte del campo estás personalizando. Por ejemplo:

- *Twig*

```
{% form_tema form _self %}

{% block _producto_nombre_reproductor %}
    <div class="texto_reproductor">
        {{ block('campo_reproductor') }}
    </div>
{% endblock %}

{{ form_reproductor(form.nombre) }}
```

- *PHP*

```
<!-- Plantilla principal -->

<?php echo $view['form']->setTheme($formulario, array('AcmeDemoBundle:Form')); ?>

<?php echo $view['form']->widget($formulario['nombre']); ?>

<!-- src/Acme/DemoBundle/Resources/views/Form/_producto_nombre_reproductor.html.php -->

<div class="texto_reproductor">
      echo $view['form']->renderBlock('campo_reproductor') ?>
</div>
```

Aquí, el fragmento _nombre_producto_reproductor define la plantilla a utilizar para el campo cuyo id es producto_nombre (y nombre es producto[nombre]).

Truco: La parte producto del campo es el nombre del formulario, el cual puedes ajustar manualmente o generar automáticamente a partir del nombre del tipo en el formulario (por ejemplo, ProductoType equivale a producto). Si no estás seguro cual es el nombre del formulario, solo ve el código fuente del formulario generado.

También puedes sustituir el marcado de toda la fila de un campo usando el mismo método:

- *Twig*

```
{% form_tema form _self %}

{% block _producto_nombre_fila %}
    <div class="nombre_fila">
        {{ form_label(formulario) }}
        {{ form_errors(formulario) }}
        {{ form_widget(formulario) }}
    </div>
{% endblock %}
```

- *PHP*

```
<!-- _producto_nombre_fila.html.php -->

<div class="nombre_fila">
    <?php echo $view['form']->label($formulario) ?>
    <?php echo $view['form']->errors($formulario) ?>
```

```
        <?php echo $view['form']->widget($formulario) ?>
    </div>
```

3.17.9 Otras personalizaciones comunes

Hasta el momento, esta receta ha mostrado varias formas de personalizar una sola pieza de cómo se reproduce un formulario. La clave está en personalizar un fragmento específico que corresponde a la porción del formulario que deseas controlar (consulta *nombrando bloques de formulario* (Página 312)).

En las siguientes secciones, verás cómo puedes hacer varias personalizaciones de formulario comunes. Para aplicar estas personalizaciones, utiliza uno de los dos métodos descritos en la sección *cookbook-form-twig-two-methods*.

Personalizando la salida de error

Nota: El componente *form* sólo se ocupa de la *forma* en que los errores de validación se reproducen, y no los mensajes de error de validación reales. Los mensajes de error están determinados por las restricciones de validación que apliques a tus objetos. Para más información, ve el capítulo *Validando* (Página 149).

Hay muchas maneras de personalizar el modo en que se representan los errores cuando se envía un formulario con errores. Los mensajes de error de un campo se reproducen cuando se utiliza el ayudante `form_errors`:

- *Twig*

  ```
  {{ form_errors(form.edad) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->errors($formulario['edad']); ?>
  ```

De forma predeterminada, los errores se representan dentro de una lista desordenada:

```
<ul>
    <li>Este campo es obligatorio</li>
</ul>
```

Para redefinir cómo se reproducen los errores para *todos* los campos, simplemente copia, pega y personaliza el fragmento `field_errors`.

- *Twig*

  ```
  {% block field_errors %}
  {% spaceless %}
      {% if errors|length > 0 %}
      <ul class="error_list">
          {% for error in errors %}
              <li>{{ error.messageTemplate|trans(error.messageParameters, 'validators') }}</li>
          {% endfor %}
      </ul>
      {% endif %}
  {% endspaceless %}
  {% endblock field_errors %}
  ```

- *PHP*

```
<!-- fields_errors.html.php -->

<?php if (count($errors) > 0): ?>
    <ul class="error_list">
        <?php foreach ($errors as $error): ?>
            <li><?php echo $view['translator']->trans(
                $error->getMessageTemplate(),
                $error->getMessageParameters(),
                'validators'
            ) ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif ?>
```

Truco: Consulta *Tematizando formularios* (Página 312) para ver cómo aplicar esta personalización.

También puedes personalizar la salida de error de sólo un tipo de campo específico. Por ejemplo, algunos errores que son más globales en tu formulario (es decir, no específicos a un solo campo) se reproducen por separado, por lo general en la parte superior de tu formulario:

- *Twig*

  ```
  {{ form_errors(form) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->render($formulario); ?>
  ```

Para personalizar *sólo* el formato utilizado por estos errores, sigue las mismas instrucciones que el anterior, pero ahora llamamos al bloque `form_errors` (*Twig*) / el archivo `form_errors.html.php` (PHP). Ahora, al reproducir errores del tipo `form`, se utiliza el fragmento personalizado en lugar del `field_errors` predeterminado.

Personalizando una "fila del formulario"

Cuando consigas manejarla, la forma más fácil para reproducir un campo de formulario es a través de la función `form_row`, la cual reproduce la etiqueta, errores y el elemento gráfico *HTML* de un campo. Para personalizar el formato utilizado para reproducir *todas* las filas de los campos del formulario, redefine el fragmento `field_row`. Por ejemplo, supongamos que deseas agregar una clase al elemento `div` alrededor de cada fila:

- *Twig*

  ```
  {% block field_row %}
      <div class="form_row">
          {{ form_label(formulario) }}
          {{ form_errors(formulario) }}
          {{ form_widget(formulario) }}
      </div>
  {% endblock field_row %}
  ```

- *PHP*

  ```
  <!-- field_row.html.php -->

  <div class="form_row">
      <?php echo $view['form']->label($formulario) ?>
      <?php echo $view['form']->errors($formulario) ?>
  ```

```
        <?php echo $view['form']->widget($formulario) ?>
    </div>
```

Truco: Consulta *Tematizando formularios* (Página 312) para ver cómo aplicar esta personalización.

Añadiendo un asterisco "Requerido" a las etiquetas de campo

Si deseas denotar todos los campos obligatorios con un asterisco requerido (*), lo puedes hacer personalizando el fragmento `field_label`.

En *Twig*, si estás haciendo la personalización del formulario dentro de la misma plantilla que tu formulario, modifica la etiqueta `use` y añade lo siguiente:

```
{% use 'form_div_base.html.twig' with field_label as base_field_label %}

{% block field_label %}
    {{ block('base_field_label') }}

    {% if required %}
        <span class="required" titulo="This field is required">*</span>
    {% endif %}
{% endblock %}
```

En *Twig*, si estás haciendo la personalización del formulario dentro de una plantilla separada, utiliza lo siguiente:

```
{% extends 'form_div_base.html.twig' %}

{% block field_label %}
    {{ parent() }}

    {% if required %}
        <span class="required" titulo="This field is required">*</span>
    {% endif %}
{% endblock %}
```

Cuando usas PHP como motor de plantillas tienes que copiar el contenido desde la plantilla original:

```
<!-- field_label.html.php -->

<!-- contenido original -->
<label for="<?php echo $view->escape($id) ?>" <?php foreach($attr as $k => $v) { printf('%s="%s" ', $

<!-- personalización -->
<?php if ($required) : ?>
    <span class="required" titulo="Este campo es obligatorio">*</span>
<?php endif ?>
```

Truco: Consulta *Tematizando formularios* (Página 312) para ver cómo aplicar esta personalización.

Añadiendo mensajes de "ayuda"

También puedes personalizar los elementos gráficos del formulario para que tengan un mensaje de "ayuda" opcional.

En *Twig*, si estás haciendo la personalización del formulario dentro de la misma plantilla que tu formulario, modifica la etiqueta `use` y añade lo siguiente:

```
{% use 'form_div_base.html.twig' with field_widget as base_field_widget %}

{% block field_widget %}
    {{ block('base_field_widget') }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

En *Twig*, si estás haciendo la personalización del formulario dentro de una plantilla separada, utiliza lo siguiente:

```
{% extends 'form_div_base.html.twig' %}

{% block field_widget %}
    {{ parent() }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

Cuando usas PHP como motor de plantillas tienes que copiar el contenido desde la plantilla original:

```
<!-- field_widget.html.php -->

<!-- contenido original -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($valor) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>

<!-- Personalización -->
<?php if (isset($help)) : ?>
    <span class="help"><?php echo $view->escape($help) ?></span>
<?php endif ?>
```

Para reproducir un mensaje de ayuda debajo de un campo, pásalo en una variable `help`:

- *Twig*

  ```
  {{ form_widget(form.titulo, { 'help': 'foobar' }) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->widget($formulario['titulo'], array('help' => 'foobar')) ?>
  ```

Truco: Consulta *Tematizando formularios* (Página 312) para ver cómo aplicar esta personalización.

3.18 Cómo crear un tipo de campo de formulario personalizado

Este artículo no se ha escrito todavía, pero será muy pronto. Si estás interesado en escribir esta entrada, consulta *Colaborando en la documentación* (Página 633).

3.19 Cómo crear una restricción de validación personalizada

Puedes crear una restricción personalizada extendiendo de la clase base "constraint", `Symfony\Component\Validator\Constraint`. Las opciones para tu restricción se representan como propiedades públicas de la clase 'constraint'. Por ejemplo, la restricción *URL* (Página 526) incluye las propiedades `message` y `protocols`:

```
namespace Symfony\Component\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class Url extends Constraint
{
    public $mensaje = 'Este valor no es una URL válida';
    public $protocolos = array('http', 'https', 'ftp', 'ftps');
}
```

Nota: La anotación `@Annotation` es necesaria para esta nueva restricción con el fin de hacerla disponible para su uso en clases vía anotaciones.

Como puedes ver, una clase de restricción es bastante mínima. La validación real la realiza otra clase "validador de restricción". La clase validador de restricción se especifica por el método de la restricción `validatedBy()`, que por omisión incluye alguna lógica simple:

```
// en la clase base Symfony\Component\Validator\Constraint
public function validatedBy()
{
    return get_class($this).'Validator';
}
```

En otras palabras, si creas una `restricción` personalizada (por ejemplo, `MyConstraint`), *Symfony2* automáticamente buscará otra clase, `MyConstraintValidator` cuando realmente lleva a cabo la validación.

La clase "validator" también es simple, y sólo tiene un método requerido: `isValid`. Tomemos al `NotBlankValidator` como ejemplo:

```
class NotBlankValidator extends ConstraintValidator
{
    public function isValid($valor, Constraint $constraint)
    {
        if (null === $valor || '' === $valor) {
            $this->setMessage($constraint->message);

            return false;
        }

        return true;
    }
}
```

3.19.1 Restricción de validadores con dependencias

Si la restricción del validador tiene dependencias, tal como una conexión de base de datos, se tendrá que configurar como un servicio en el contenedor de inyección de dependencias. Este servicio debe incluir la etiqueta `validator.constraint_validator` y un atributo `alias`:

- *YAML*

```
services:
    validator.unique.your_validator_name:
        class: Fully\Qualified\Validator\Class\Name
        tags:
            - { name: validator.constraint_validator, alias: alias_name }
```

- *XML*

```
<service id="validator.unique.your_validator_name" class="Fully\Qualified\Validator\Class\Name">
    <argument type="service" id="doctrine.orm.default_entity_manager" />
    <tag name="validator.constraint_validator" alias="alias_name" />
</service>
```

- *PHP*

```
$contenedor
    ->register('validator.unique.your_validator_name', 'Fully\Qualified\Validator\Class\Name')
    ->addTag('validator.constraint_validator', array('alias' => 'alias_name'))
;
```

Tu clase de restricción ahora podrá utilizar este alias para referirse al validador correspondiente:

```
public function validatedBy()
{
    return 'alias_name';
}
```

3.20 Cómo dominar y crear nuevos entornos

Cada aplicación es la combinación de código y un conjunto de configuración que dicta la forma en que el código debería funcionar. La configuración puede definir la base de datos en uso, si o no se debe almacenar en caché, o cómo se debe detallar el registro. En *Symfony2*, la idea de "entornos" es la idea de que el mismo código base se puede ejecutar con varias configuraciones diferentes. Por ejemplo, el entorno `dev` debería usar la configuración que facilita el desarrollo y lo hace agradable, mientras que el entorno `prod` debe usar un conjunto de configuración optimizado para la velocidad.

3.20.1 Diferentes entornos, diferentes archivos de configuración

Una típica aplicación *Symfony2* comienza con tres ambientes: `dev`, `prod` y `test`. Como mencionamos, cada "entorno" simplemente representa una manera de ejecutar el mismo código base con diferente configuración. No debería ser una sorpresa entonces que cada entorno cargue su propio archivo de configuración individual. Si estás utilizando el formato de configuración YAML, utiliza los siguientes archivos:

- para el entorno `dev`: `app/config/config_dev.yml`
- para el entorno `prod`: `app/config/config_prod.yml`
- para el entorno `test`: `app/config/config_test.yml`

Esto funciona a través de un estándar sencillo que se utiliza por omisión dentro de la clase `AppKernel`:

```
// app/AppKernel.php
// ...

class AppKernel extends Kernel
{
    // ...

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

Como puedes ver, cuando se carga *Symfony2*, utiliza el entorno especificado para determinar qué archivo de configuración cargar. Esto cumple con el objetivo de múltiples entornos en una manera elegante, potente y transparente.

Por supuesto, en realidad, cada entorno difiere un poco de los demás. En general, todos los entornos comparten una gran base de configuración común. Abriendo el archivo de configuración "dev", puedes ver cómo se logra esto fácil y transparentemente:

- *YAML*

  ```
  imports:
      - { resource: config.yml }

  # ...
  ```

- *XML*

  ```
  <imports>
      <import resource="config.xml" />
  </imports>

  <!-- ... -->
  ```

- *PHP*

  ```
  $cargador->import('config.php');

  // ...
  ```

Para compartir configuración común, el archivo de configuración de cada entorno simplemente importa primero los ajustes más comunes desde un archivo de configuración central (`config.yml`). El resto del archivo se puede desviar de la configuración predeterminada sustituyendo parámetros individuales. Por ejemplo, de manera predeterminada, la barra de herramientas `web_profiler` está desactivada. Sin embargo, en el entorno `dev`, la barra de herramientas se activa modificando el valor predeterminado en el archivo de configuración `dev`:

- *YAML*

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

  web_profiler:
      toolbar: true
      # ...
  ```

- *XML*

```
<!-- app/config/config_dev.xml -->
<imports>
    <import resource="config.xml" />
</imports>

<webprofiler:config
    toolbar="true"
    # ...
/>
```

- *PHP*

```
// app/config/config_dev.php
$cargador->import('config.php');

$contenedor->loadFromExtension('web_profiler', array(
    'toolbar' => true,
    // ..
));
```

3.20.2 Ejecutando una aplicación en entornos diferentes

Para ejecutar la aplicación en cada entorno, carga la aplicación usando como controlador frontal o bien `app.php` (para el entorno `prod`) o `app_dev.php` (para el entorno `dev`):

```
http://localhost/app.php      -> entorno *prod*
http://localhost/app_dev.php  -> entorno *dev*
```

Nota: Las direcciones *URL* dadas suponen que tu servidor web está configurado para utilizar el directorio `web/` de la aplicación como su raíz. Lee más en *Instalando Symfony2* (Página 53).

Si abres uno de estos archivos, rápidamente verás que el entorno utilizado por cada uno se fija explícitamente:

```
<?php

require_once __DIR__.'/../app/bootstrap_cache.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppCache(new AppKernel('prod', false));
$kernel->handle(Request::createFromGlobals())->send();
```

Como puedes ver, la clave `prod` especifica que este entorno se ejecutará en el entorno `producción`. Una aplicación *Symfony2* se puede ejecutar en cualquier entorno usando este código y simplemente cambiando la cadena de entorno.

Nota: El entorno `test` se utiliza al escribir las pruebas funcionales y no es accesible en el navegador directamente a través de un controlador frontal. En otras palabras, a diferencia de los otros entornos, no hay archivo controlador frontal `app_test.php`.

Modo *Debug*

Importante, pero irrelevante al tema de *entornos* es la clave `false` en la línea 8 del controlador frontal anterior. Esto especifica si o no la aplicación se debe ejecutar en "modo de depuración". Independientemente del entorno, una aplicación *Symfony2* se puede ejecutar con el modo de depuración establecido en `true` o `false`. Esto afecta a muchas cosas en la aplicación, tal como cuando o no se deben mostrar los errores o si los archivos de caché se reconstruyen de forma dinámica en cada petición. Aunque no es un requisito, el modo de depuración generalmente se fija a `true` para los entornos `dev` y `test` y `false` para el entorno `prod`.
Internamente, el valor del modo de depuración viene a ser el parámetro `kernel.debug` utilizado en el interior del *contenedor de servicios* (Página 237). Si miras dentro del archivo de configuración de tu aplicación, puedes encontrar el parámetro utilizado, por ejemplo, para activar o desactivar el registro cuando se utiliza el *DBAL Doctrine*:

- *YAML*

```
doctrine:
    dbal:
        logging: %kernel.debug%
        # ...
```

- *XML*

```
<doctrine:dbal logging="%kernel.debug%" ... />
```

- *PHP*

```
$contenedor->loadFromExtension('doctrine', array(
    'dbal' => array(
        'logging'  => '%kernel.debug%',
        // ...
    ),
    // ...
));
```

3.20.3 Creando un nuevo entorno

De forma predeterminada, una aplicación *Symfony2* tiene tres entornos que se encargan de la mayoría de los casos. Por supuesto, debido a que un entorno no es más que una cadena que corresponde a un conjunto de configuración, la creación de un nuevo entorno es muy fácil.

Supongamos, por ejemplo, que antes de implantarla, es necesario comparar tu aplicación. Una forma de comparar la aplicación es usando la configuración cercana a la producción, pero con el `web_profiler` de *Symfony2* habilitado. Esto permite a *Symfony2* registrar información sobre tu aplicación durante la evaluación comparativa.

La mejor manera de lograrlo es a través de un nuevo entorno llamado, por ejemplo, `benchmark`. Comienza creando un nuevo archivo de configuración:

- *YAML*

```
# app/config/config_benchmark.yml

imports:
    - { resource: config_prod.yml }

framework:
    profiler: { only_exceptions: false }
```

- *XML*

```xml
<!-- app/config/config_benchmark.xml -->

<imports>
    <import resource="config_prod.xml" />
</imports>

<framework:config>
    <framework:profiler only-exceptions="false" />
</framework:config>
```

- *PHP*

```php
// app/config/config_benchmark.php

$cargador->import('config_prod.php')

$contenedor->loadFromExtension('framework', array(
    'profiler' => array('only-exceptions' => false),
));
```

Y con esta simple adición, la aplicación ahora es compatible con un nuevo entorno llamado `benchmark`.

Este nuevo archivo de configuración importa la configuración del entorno `prod` y la modifica. Esto garantiza que el nuevo entorno es idéntico al entorno `prod`, a excepción de los cambios echos explícitamente aquí.

Debido a que deseas que este entorno sea accesible a través de un navegador, también debes crear un controlador frontal para el mismo. Copia el archivo `web/app.php` a `web/app_benchmark.php` y edita el entorno para que sea `benchmark`:

```php
<?php

require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('benchmark', false);
$kernel->handle(Request::createFromGlobals())->send();
```

El nuevo entorno ahora es accesible a través de:

http://localhost/app_benchmark.php

Nota: Algunos entornos, como el entorno `dev`, no están destinados a ser visitados en algún servidor empleado para el público en general. Esto se debe a que ciertos entornos, con fines de depuración, pueden dar demasiada información sobre la aplicación o infraestructura subyacente. Para estar seguros de que estos entornos no son accesibles, el controlador frontal suele ser protegido de direcciones IP externas a través del siguiente código en la parte superior del controlador:

```
if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1'))) {
    die('You are not allowed to access this file. Check '.basename(__FILE__).' for more informat
}
```

3.20.4 Entornos y el directorio de caché

Symfony2 aprovecha la memorización en caché de muchas maneras: la configuración de la aplicación, la configuración de enrutado, las plantillas *Twig* y más, se memorizan en objetos caché de *PHP* en archivos del sistema de archivos.

Por omisión, estos archivos se memorizan principalmente en el directorio `app/cache`. Sin embargo, cada entorno memoriza su propio conjunto de archivos:

```
app/cache/dev   - directorio cachá para el entorno *dev*
app/cache/prod  - directorio caché para el entorno *prod*
```

A veces, cuando depuramos, puede ser útil inspeccionar un archivo memorizado para entender cómo está funcionando algo. Al hacerlo, recuerda buscar en el directorio del entorno que estás utilizando (comúnmente `dev` mientras desarrollas y depuras). Aunque puede variar, el directorio `app/cache/dev` incluye lo siguiente:

- `appDevDebugProjectContainer.php` - el "contenedor del servicio" memorizado que representa la configuración de la aplicación en caché;
- `appdevUrlGenerator.php` - la clase *PHP* generada a partir de la configuración de enrutado y usada cuando genera las *URL*;
- `appdevUrlMatcher.php` - la clase *PHP* usada para emparejar rutas - busca aquí para ver la lógica de las expresiones regulares compiladas utilizadas para concordar las *URL* entrantes con diferentes rutas;
- `twig/` - este directorio contiene todas las plantillas *Twig* en caché.

3.20.5 Prosigue

Lee el artículo en *Cómo configurar parámetros externos en el contenedor de servicios* (Página 329).

3.21 Cómo configurar parámetros externos en el contenedor de servicios

En el capítulo *Cómo dominar y crear nuevos entornos* (Página 324), aprendiste cómo gestionar la configuración de tu aplicación. A veces, puedes beneficiar a tu aplicación almacenando ciertas credenciales fuera del código de tu proyecto. La configuración de la base de datos es tal ejemplo. La flexibilidad del contenedor de servicios de *Symfony* te permite hacer esto fácilmente.

3.21.1 Variables de entorno

Symfony grabará cualquier variable de entorno con el prefijo `SYMFONY__` y lo configurará como parámetro en el contenedor de servicios. Los subrayados dobles son reemplazados por un punto, ya que un punto no es un carácter válido en un nombre de variable de entorno.

Por ejemplo, si estás usando Apache, las variables de entorno se pueden fijar usando la siguiente configuración de `VirtualHost`:

```
<VirtualHost *:80>
    ServerName      Symfony2
    DocumentRoot    "/ruta/a/mi_aplic_symfony2/web"
    DirectoryIndex  index.php index.html
    SetEnv          SYMFONY__DATABASE__USER user
    SetEnv          SYMFONY__DATABASE__PASSWORD secret

    <Directory "/ruta/a/mi_aplic_symfony2/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

Nota: El ejemplo anterior es una configuración para Apache, con la directiva SetEnv. Sin embargo, esta funcionará para cualquier servidor web compatible con la configuración de variables de entorno.

Ahora que has declarado una variable de entorno, estará presente en la variable global PHP `$_SERVER`. Entonces *Symfony* automáticamente fija todas las variables `$_SERVER` con el prefijo `SYMFONY__` como parámetros en el contenedor de servicios.

Ahora puedes referirte a estos parámetros donde los necesites.

- *YAML*

```
doctrine:
    dbal:
        driver    pdo_mysql
        dbname:   symfony2_project
        user:     %database.user%
        password: %database.password%
```

- *XML*

```
<!-- xmlns:doctrine="http://symfony.com/schema/dic/doctrine" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/do

<doctrine:config>
    <doctrine:dbal
        driver="pdo_mysql"
        dbname="symfony2_projet"
        user="%database.user%"
        password="%database.password%"
    />
</doctrine:config>
```

- *PHP*

```
$contenedor->loadFromExtension('doctrine', array('dbal' => array(
    'driver'   => 'pdo_mysql',
    'dbname'   => 'symfony2_project',
    'user'     => '%database.user%',
    'password' => '%database.password%',
));
```

3.21.2 Constantes

El contenedor también cuenta con apoyo para fijar constantes PHP como parámetros. Para aprovechar esta característica, asigna el nombre de tu constante a un parámetro clave, y define el tipo como `constant`.

```
<?xml version="1.0" encoding="UTF-8"?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>

    <parameters>
        <parameter key="global.constant.value" type="constant">GLOBAL_CONSTANT</parameter>
        <parameter key="my_class.constant.value" type="constant">My_Class::CONSTANT_NAME</parame
    </parameters>
</container>
```

Nota: Esto sólo funciona para la configuración XML. Si *no* estás usando XML, sólo tienes que importar un archivo XML para tomar ventaja de esta funcionalidad:

```
// app/config/config.yml
imports:
    - { resource: parameters.xml }
```

3.21.3 Otra configuración

La directiva `imports` se puede utilizar para extraer parámetros almacenados en otro lugar. Importar un archivo *PHP* te da la flexibilidad de añadir al contenedor lo que sea necesario. La siguiente directiva importa un archivo llamado `parameters.php`.

- *YAML*

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

- *XML*

  ```
  <!-- app/config/config.xml -->
  <imports>
      <import resource="parameters.php" />
  </imports>
  ```

- *PHP*

  ```
  // app/config/config.php
  $cargador->import('parameters.php');
  ```

Nota: Un archivo de recursos puede tener uno de muchos tipos. Los recursos *PHP*, *XML*, *YAML*, *INI* y cierre son compatibles con la directiva `imports`.

En `parameters.php`, dile al contenedor de servicios los parámetros que deseas configurar. Esto es útil cuando la configuración importante está en un formato no estándar. El siguiente ejemplo incluye la configuración de una base de datos Drupal en el contenedor de servicios de *Symfony*.

```
// app/config/parameters.php

include_once('/ruta/a/drupal/sites/all/default/settings.php');
$contenedor->setParameter('drupal.database.url', $db_url);
```

3.22 Cómo utilizar el patrón fábrica para crear servicios

Los contenedores de servicios de *Symfony2* proporcionan una forma eficaz de controlar la creación de objetos, lo cual te permite especificar los argumentos pasados al constructor, así como llamar a los métodos y establecer parámetros. A veces, sin embargo, esto no te proporcionará todo lo necesario para construir tus objetos. Por esta situación, puedes utilizar una fábrica para crear el objeto y decirle al contenedor de servicios que llame a un método en la fábrica y no directamente una instancia del objeto.

Supongamos que tienes una fábrica que configura y devuelve un nuevo objeto BoletinGestor:

```
namespace Acme\HolaBundle\Newletter;

class NewsletterFactory
{
    public function get()
    {
        $newsletterManager = new BoletinGestor();

        // ...

        return $newsletterManager;
    }
}
```

Para que el objeto `BoletinGestor` esté disponible como servicio, puedes configurar el contenedor de servicios para usar la clase de fábrica `NewsletterFactory`:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor
    boletin_factory.class: Acme\HolaBundle\Boletin\NewsletterFactory
services:
    boletin_gestor:
        class:          %boletin_gestor.class%
        factory_class:  %newsletter_factory.class%
        factory_method: get
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
    <parameter key="boletin_factory.class">Acme\HolaBundle\Boletin\NewsletterFactory</parameter>
</parameters>

<services>
    <service id="boletin_gestor"
             class="%boletin_gestor.class%"
             factory-class="%newsletter_factory.class%"
             factory-method="get"
    />
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');
$contenedor->setParameter('boletin_factory.class', 'Acme\HolaBundle\Boletin\NewsletterFactory');

$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%'
))->setFactoryClass(
    '%newsletter_factory.class%'
```

```
)->setFactoryMethod(
    'get'
);
```

Cuando especificas la clase que utiliza la fábrica (a través de `factory_class`), el método será llamado estáticamente. Si la fábrica debe crear una instancia y se llama al método del objeto resultante (como en este ejemplo), configura la fábrica misma como un servicio:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor
    boletin_factory.class: Acme\HolaBundle\Boletin\NewsletterFactory
services:
    boletin_factory:
        class:           %newsletter_factory.class%
    boletin_gestor:
        class:           %boletin_gestor.class%
        factory_service:  boletin_factory
        factory_method:   get
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
    <parameter key="boletin_factory.class">Acme\HolaBundle\Boletin\NewsletterFactory</parameter>
</parameters>

<services>
    <service id="boletin_factory" class="%newsletter_factory.class%"/>
    <service id="boletin_gestor"
             class="%boletin_gestor.class%"
             factory-service="boletin_factory"
             factory-method="get"
    />
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');
$contenedor->setParameter('boletin_factory.class', 'Acme\HolaBundle\Boletin\NewsletterFactory');

$contenedor->setDefinition('boletin_factory', new Definition(
    '%newsletter_factory.class%'
))
$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%'
))->setFactoryService(
    'boletin_factory'
)->setFactoryMethod(
    'get'
```

```
);
```

Nota: El servicio fábrica se indica por su nombre de id y no una referencia al propio servicio. Por lo tanto, no es necesario utilizar la sintaxis @.

3.22.1 Pasando argumentos al método fábrica

Si tienes que pasar argumentos al método fábrica, puedes utilizar la opción `arguments` dentro del contenedor de servicios. Por ejemplo, supongamos que el método `get` en el ejemplo anterior tiene el servicio de `templating` como argumento:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor
    boletin_factory.class: Acme\HolaBundle\Boletin\NewsletterFactory
services:
    boletin_factory:
        class:           %newsletter_factory.class%
    boletin_gestor:
        class:           %boletin_gestor.class%
        factory_service:  boletin_factory
        factory_method:   get
        arguments:
            -             @templating
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
    <parameter key="boletin_factory.class">Acme\HolaBundle\Boletin\NewsletterFactory</parameter>
</parameters>

<services>
    <service id="boletin_factory" class="%newsletter_factory.class%"/>
    <service id="boletin_gestor"
             class="%boletin_gestor.class%"
             factory-service="boletin_factory"
             factory-method="get"
    >
        <argument type="service" id="templating" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');
$contenedor->setParameter('boletin_factory.class', 'Acme\HolaBundle\Boletin\NewsletterFactory');
```

```
    $contenedor->setDefinition('boletin_factory', new Definition(
        '%newsletter_factory.class%'
    ))
    $contenedor->setDefinition('boletin_gestor', new Definition(
        '%boletin_gestor.class%',
        array(new Reference('templating'))
    ))->setFactoryService(
        'boletin_factory'
    )->setFactoryMethod(
        'get'
    );
```

3.23 Cómo gestionar dependencias comunes con servicios padre

A medida que agregas más funcionalidad a tu aplicación, puedes comenzar a tener clases relacionadas que comparten algunas de las mismas dependencias. Por ejemplo, puedes tener un gestor de boletines que utiliza inyección para definir sus dependencias:

```
namespace Acme\HolaBundle\Mail;

use Acme\HolaBundle\Mailer;
use Acme\HolaBundle\EmailFormatter;

class BoletinGestor
{
    protected $cartero;
    protected $emailFormatter;

    public function setCartero(Mailer $cartero)
    {
        $this->cartero = $cartero;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

y también una clase Tarjeta de Saludo que comparte las mismas dependencias:

```
namespace Acme\HolaBundle\Mail;

use Acme\HolaBundle\Mailer;
use Acme\HolaBundle\EmailFormatter;

class GreetingCardManager
{
    protected $cartero;
    protected $emailFormatter;

    public function setCartero(Mailer $cartero)
    {
        $this->cartero = $cartero;
    }
```

```
    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

La configuración del servicio de estas clases se vería algo como esto:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Mail\BoletinGestor
    greeting_card_manager.class: Acme\HolaBundle\Mail\GreetingCardManager
services:
    mi_cartero:
        # ...
    my_correo_formatter:
        # ...
    boletin_gestor:
        class:    %boletin_gestor.class%
        calls:
            - [ setCartero, [ @mi_cartero ] ]
            - [ setEmailFormatter, [ @my_correo_formatter] ]

    greeting_card_manager:
        class:    %greeting_card_manager.class%
        calls:
            - [ setCartero, [ @mi_cartero ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Mail\BoletinGestor</parameter>
    <parameter key="greeting_card_manager.class">Acme\HolaBundle\Mail\GreetingCardManager</param
</parameters>

<services>
    <service id="mi_cartero" ... >
      <!-- ... -->
    </service>
    <service id="my_correo_formatter" ... >
      <!-- ... -->
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%">
        <call method="setCartero">
             <argument type="service" id="mi_cartero" />
        </call>
        <call method="setEmailFormatter">
             <argument type="service" id="my_correo_formatter" />
        </call>
    </service>
    <service id="greeting_card_manager" class="%greeting_card_manager.class%">
        <call method="setCartero">
```

```
            <argument type="service" id="mi_cartero" />
        </call>
        <call method="setEmailFormatter">
            <argument type="service" id="my_correo_formatter" />
        </call>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Mail\BoletinGestor');
$contenedor->setParameter('greeting_card_manager.class', 'Acme\HolaBundle\Mail\GreetingCardManag

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('my_correo_formatter', ... );
$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%'
))->addMethodCall('setCartero', array(
    new Reference('mi_cartero')
))->addMethodCall('setEmailFormatter', array(
    new Reference('my_correo_formatter')
));
$contenedor->setDefinition('greeting_card_manager', new Definition(
    '%greeting_card_manager.class%'
))->addMethodCall('setCartero', array(
    new Reference('mi_cartero')
))->addMethodCall('setEmailFormatter', array(
    new Reference('my_correo_formatter')
));
```

Hay mucha repetición, tanto en las clases como en la configuración. Esto significa que si cambias, por ejemplo, las clases de correo de la aplicación `Mailer` de `EmailFormatter` para inyectarlas a través del constructor, tendrías que actualizar la configuración en dos lugares. Del mismo modo, si necesitas hacer cambios en los métodos definidores tendrías que hacerlo en ambas clases. La forma típica de hacer frente a los métodos comunes de estas clases relacionadas es extraerlas en una superclase:

```
namespace Acme\HolaBundle\Mail;

use Acme\HolaBundle\Mailer;
use Acme\HolaBundle\EmailFormatter;

abstract class MailManager
{
    protected $cartero;
    protected $emailFormatter;

    public function setCartero(Mailer $cartero)
    {
        $this->cartero = $cartero;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
```

```
    }
    // ...
}
```

Entonces `BoletinGestor` y `GreetingCardManager` pueden extender esta superclase:

```
namespace Acme\HolaBundle\Mail;

class BoletinGestor extends MailManager
{
    // ...
}
```

y:

```
namespace Acme\HolaBundle\Mail;

class GreetingCardManager extends MailManager
{
    // ...
}
```

De manera similar, el contenedor de servicios de *Symfony2* también apoya la extensión de servicios en la configuración por lo que también puedes reducir la repetición especificando un padre para un servicio.

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Mail\BoletinGestor
    greeting_card_manager.class: Acme\HolaBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HolaBundle\Mail\MailManager
services:
    mi_cartero:
        # ...
    my_correo_formatter:
        # ...
    mail_manager:
        class:    %mail_manager.class%
        abstract:  true
        calls:
            - [ setCartero, [ @mi_cartero ] ]
            - [ setEmailFormatter, [ @my_correo_formatter] ]

    boletin_gestor:
        class:    %boletin_gestor.class%
        parent: mail_manager

    greeting_card_manager:
        class:    %greeting_card_manager.class%
        parent: mail_manager
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Mail\BoletinGestor</parameter>
    <parameter key="greeting_card_manager.class">Acme\HolaBundle\Mail\GreetingCardManager</param
```

```
    <parameter key="mail_manager.class">Acme\HolaBundle\Mail\MailManager</parameter>
</parameters>

<services>
    <service id="mi_cartero" ... >
      <!-- ... -->
    </service>
    <service id="my_correo_formatter" ... >
      <!-- ... -->
    </service>
    <service id="mail_manager" class="%mail_manager.class%" abstract="true">
        <call method="setCartero">
            <argument type="service" id="mi_cartero" />
        </call>
        <call method="setEmailFormatter">
            <argument type="service" id="my_correo_formatter" />
        </call>
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%" parent="mail_manager"/>
    <service id="greeting_card_manager" class="%greeting_card_manager.class%" parent="mail_mana
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Mail\BoletinGestor');
$contenedor->setParameter('greeting_card_manager.class', 'Acme\HolaBundle\Mail\GreetingCardManag
$contenedor->setParameter('mail_manager.class', 'Acme\HolaBundle\Mail\MailManager');

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('my_correo_formatter', ... );
$contenedor->setDefinition('mail_manager', new Definition(
    '%mail_manager.class%'
))->SetAbstract(
    true
)->addMethodCall('setCartero', array(
    new Reference('mi_cartero')
))->addMethodCall('setEmailFormatter', array(
    new Reference('my_correo_formatter')
));
$contenedor->setDefinition('boletin_gestor', new DefinitionDecorator(
    'mail_manager'
))->setClass(
    '%boletin_gestor.class%'
);
$contenedor->setDefinition('greeting_card_manager', new DefinitionDecorator(
    'mail_manager'
))->setClass(
    '%greeting_card_manager.class%'
);
```

En este contexto, tener un servicio `padre` implica que los argumentos y las llamadas a métodos del servicio padre se deben utilizar en los servicios descendientes. En concreto, los métodos definidores especificados para el servicio padre serán llamados cuando se crean instancias del servicio descendiente.

Nota: Si quitas la clave de configuración del `padre`, el servicio todavía seguirá siendo una instancia, por supuesto, extendiendo la clase `MailManager`. La diferencia es que la omisión del `padre` en la clave de configuración significa que las `llamadas` definidas en el servicio `mail_manager` no se ejecutarán al crear instancias de los servicios descendientes.

La clase padre es abstracta, ya que no se deben crear instancias directamente. Al establecerla como abstracta en el archivo de configuración como se hizo anteriormente, significa que sólo se puede utilizar como un servicio primario y no se puede utilizar directamente como un servicio para inyectar y retirar en tiempo de compilación. En otras palabras, existe sólo como una "plantilla" que otros servicios pueden utilizar.

3.23.1 Sustituyendo dependencias padre

Puede haber ocasiones en las que deses sustituir que clase se pasa a una dependencia en un servicio hijo único. Afortunadamente, añadiendo la llamada al método de configuración para el servicio hijo, las dependencias establecidas por la clase principal se sustituyen. Así que si necesitas pasar una dependencia diferente sólo para la clase `BoletinGestor`, la configuración sería la siguiente:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Mail\BoletinGestor
    greeting_card_manager.class: Acme\HolaBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HolaBundle\Mail\MailManager
services:
    mi_cartero:
        # ...
    my_alternative_mailer:
        # ...
    my_correo_formatter:
        # ...
    mail_manager:
        class:    %mail_manager.class%
        abstract:  true
        calls:
            - [ setCartero, [ @mi_cartero ] ]
            - [ setEmailFormatter, [ @my_correo_formatter] ]

    boletin_gestor:
        class:    %boletin_gestor.class%
        parent: mail_manager
        calls:
            - [ setCartero, [ @my_alternative_mailer ] ]

    greeting_card_manager:
        class:    %greeting_card_manager.class%
        parent: mail_manager
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Mail\BoletinGestor</parameter>
    <parameter key="greeting_card_manager.class">Acme\HolaBundle\Mail\GreetingCardManager</param
    <parameter key="mail_manager.class">Acme\HolaBundle\Mail\MailManager</parameter>
```

```
</parameters>

<services>
    <service id="mi_cartero" ... >
      <!-- ... -->
    </service>
    <service id="my_alternative_mailer" ... >
      <!-- ... -->
    </service>
    <service id="my_correo_formatter" ... >
      <!-- ... -->
    </service>
    <service id="mail_manager" class="%mail_manager.class%" abstract="true">
        <call method="setCartero">
             <argument type="service" id="mi_cartero" />
        </call>
        <call method="setEmailFormatter">
             <argument type="service" id="my_correo_formatter" />
        </call>
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%" parent="mail_manager">
         <call method="setCartero">
             <argument type="service" id="my_alternative_mailer" />
        </call>
    </service>
    <service id="greeting_card_manager" class="%greeting_card_manager.class%" parent="mail_mana
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Mail\BoletinGestor');
$contenedor->setParameter('greeting_card_manager.class', 'Acme\HolaBundle\Mail\GreetingCardManag
$contenedor->setParameter('mail_manager.class', 'Acme\HolaBundle\Mail\MailManager');

$contenedor->setDefinition('mi_cartero', ... );
$contenedor->setDefinition('my_alternative_mailer', ... );
$contenedor->setDefinition('my_correo_formatter', ... );
$contenedor->setDefinition('mail_manager', new Definition(
    '%mail_manager.class%'
))->SetAbstract(
    true
)->addMethodCall('setCartero', array(
    new Reference('mi_cartero')
))->addMethodCall('setEmailFormatter', array(
    new Reference('my_correo_formatter')
));
$contenedor->setDefinition('boletin_gestor', new DefinitionDecorator(
    'mail_manager'
))->setClass(
    '%boletin_gestor.class%'
)->addMethodCall('setCartero', array(
    new Reference('my_alternative_mailer')
));
$contenedor->setDefinition('boletin_gestor', new DefinitionDecorator(
```

```
        'mail_manager'
    ))->setClass(
        '%greeting_card_manager.class%'
    );
```

El `GreetingCardManager` recibirá las mismas dependencias que antes, pero la `BoletinGestor` será pasada a `my_alternative_mailer` en lugar del servicio `mi_cartero`.

3.23.2 Colección de dependencias

Cabe señalar que el método definidor sustituido en el ejemplo anterior en realidad se llama dos veces - una vez en la definición del padre y otra más en la definición del hijo. En el ejemplo anterior, esto estaba muy bien, ya que la segunda llamada a `setCartero` sustituye al objeto mailer establecido por la primera llamada.

En algunos casos, sin embargo, esto puede ser un problema. Por ejemplo, si la sustitución a la llamada al método consiste en añadir algo a una colección, entonces se agregarán dos objetos a la colección. A continuación mostramos tal caso, si la clase padre se parece a esto:

```
namespace Acme\HolaBundle\Mail;

use Acme\HolaBundle\Mailer;
use Acme\HolaBundle\EmailFormatter;

abstract class MailManager
{
    protected $filtros;

    public function setFilter($filtro)
    {
        $this->filters[] = $filtro;
    }
    // ...
}
```

Si tiene la siguiente configuración:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    # ...
    boletin_gestor.class: Acme\HolaBundle\Mail\BoletinGestor
    mail_manager.class: Acme\HolaBundle\Mail\MailManager
services:
    my_filter:
        # ...
    another_filter:
        # ...
    mail_manager:
        class:    %mail_manager.class%
        abstract:  true
        calls:
            - [ setFilter, [ @my_filter ] ]

    boletin_gestor:
        class:    %boletin_gestor.class%
        parent: mail_manager
```

```
        calls:
            - [ setFilter, [ @another_filter ] ]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Mail\BoletinGestor</parameter>
    <parameter key="mail_manager.class">Acme\HolaBundle\Mail\MailManager</parameter>
</parameters>

<services>
    <service id="my_filter" ... >
      <!-- ... -->
    </service>
    <service id="another_filter" ... >
      <!-- ... -->
    </service>
    <service id="mail_manager" class="%mail_manager.class%" abstract="true">
        <call method="setFilter">
             <argument type="service" id="my_filter" />
        </call>
    </service>
    <service id="boletin_gestor" class="%boletin_gestor.class%" parent="mail_manager">
         <call method="setFilter">
             <argument type="service" id="another_filter" />
        </call>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Mail\BoletinGestor');
$contenedor->setParameter('mail_manager.class', 'Acme\HolaBundle\Mail\MailManager');

$contenedor->setDefinition('my_filter', ... );
$contenedor->setDefinition('another_filter', ... );
$contenedor->setDefinition('mail_manager', new Definition(
    '%mail_manager.class%'
))->SetAbstract(
    true
)->addMethodCall('setFilter', array(
    new Reference('my_filter')
));
$contenedor->setDefinition('boletin_gestor', new DefinitionDecorator(
    'mail_manager'
))->setClass(
    '%boletin_gestor.class%'
)->addMethodCall('setFilter', array(
    new Reference('another_filter')
));
```

En este ejemplo, el `setFilter` del servicio `boletin_gestor` se llamará dos veces, dando lugar a que el array

`$filtros` contenga ambos objetos `my_filter` y `another_filter`. Esto es genial si sólo quieres agregar filtros adicionales para subclases. Si deseas reemplazar los filtros pasados a la subclase, elimina de la matriz el ajuste de la configuración, esto evitará que la clase `setFilter` base sea llamada.

3.24 Cómo utilizar PdoSessionStorage para almacenar sesiones en la base de datos

El almacenamiento de sesiones predeterminado de *Symfony2* escribe la información de la sesión en archivo(s). La mayoría de medianos a grandes sitios web utilizan una base de datos para almacenar valores de sesión en lugar de archivos, porque las bases de datos son más fáciles de usar y escalar en un entorno multiservidor web.

Symfony2 ha incorporado una solución de almacenamiento en base de datos denominada `Symfony\Component\HttpFoundation\SessionStorage\PdoSessionStorage`. Para usarla, sólo tienes que cambiar algunos parámetros en `config.yml` (o el formato de configuración de tu elección):

- *YAML*

```
# app/config/config.yml
framework:
    session:
        # ...
        storage_id:     session.storage.pdo

parameters:
    pdo.db_options:
        db_table:    sesion
        db_id_col:   sesion_id
        db_data_col: sesion_value
        db_time_col: sesion_time

services:
    pdo:
        class: PDO
        arguments:
            dsn:      "mysql:dbname=mydatabase"
            user:     miusuario
            password: mipase

    session.storage.pdo:
        class:     Symfony\Component\HttpFoundation\SessionStorage\PdoSessionStorage
        arguments: [@pdo, %session.storage.options%, %pdo.db_options%]
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:session storage-id="session.storage.pdo" default-locale="en" lifetime="3600" autc
</framework:config>

<parameters>
    <parameter key="pdo.db_options" type="collection">
        <parameter key="db_table">session</parameter>
        <parameter key="db_id_col">session_id</parameter>
        <parameter key="db_data_col">session_value</parameter>
        <parameter key="db_time_col">session_time</parameter>
    </parameter>
</parameters>
```

```
<services>
    <service id="pdo" class="PDO">
        <argument>mysql:dbname=mydatabase</argument>
        <argument>miusuario</argument>
        <argument>mipase</argument>
    </service>

    <service id="session.storage.pdo" class="Symfony\Component\HttpFoundation\SessionStorage\Pdo
        <argument type="service" id="pdo" />
        <argument>%session.storage.options%</argument>
        <argument>%pdo.db_options%</argument>
    </service>
</services>
```

- *PHP*

```
// app/config/config.yml
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

$contenedor->loadFromExtension('framework', array(
    // ...
    'session' => array(
        // ...
        'storage_id' => 'session.storage.pdo',
    ),
));

$contenedor->setParameter('pdo.db_options', array(
    'db_table'      => 'session',
    'db_id_col'     => 'session_id',
    'db_data_col'   => 'session_value',
    'db_time_col'   => 'session_time',
));

$pdoDefinition = new Definition('PDO', array(
    'mysql:dbname=mydatabase',
    'miusuario',
    'mipase',
));
$contenedor->setDefinition('pdo', $pdoDefinition);

$storageDefinition = new Definition('Symfony\Component\HttpFoundation\SessionStorage\PdoSessionS
    new Reference('pdo'),
    '%session.storage.options%',
    '%pdo.db_options%',
));
$contenedor->setDefinition('session.storage.pdo', $storageDefinition);
```

- `db_table`: El nombre de la tabla de sesión en tu base de datos
- `db_id_col`: El nombre de la columna id en tu tabla de sesión (VARCHAR(255) o mayor)
- `db_data_col`: El nombre del valor de columna en tu tabla sesión (TEXT o CLOB)
- `db_time_col`: El nombre de la columna hora en tu tabla sesión (INTEGER)

3.24.1 Compartiendo información de conexión a tu base de datos

Con la configuración dada, la configuración de conexión de la base de datos únicamente se define para la conexión de almacenamiento de sesión. Esto está bien cuando utilizas una base de datos para los datos de sesión.

Pero si deseas almacenar los datos de sesión en la misma base que el resto de los datos de tu proyecto, puedes utilizar la configuración de conexión de *parameter.ini* refiriendo los parámetros relacionados con la base de datos definidos allí:

- *YAML*

```
pdo:
    class: PDO
    arguments:
        - "mysql:dbname=%database_name%"
        - %database_user%
        - %database_password%
```

- *XML*

```
<service id="pdo" class="PDO">
    <argument>mysql:dbname=%database_name%</argument>
    <argument>%database_user%</argument>
    <argument>%database_password%</argument>
</service>
```

- *XML*

```
$pdoDefinition = new Definition('PDO', array(
    'mysql:dbname=%database_name%',
    '%database_user%',
    '%database_password%',
));
```

3.24.2 Ejemplo de declaración *MySQL*

La declaración `SQL` para crear la tabla de base de datos necesaria podría ser similar a la siguiente (*MySQL*):

```
CREATE TABLE `session` (
    `session_id` varchar(255) NOT NULL,
    `session_value` text NOT NULL,
    `session_time` int(11) NOT NULL,
    PRIMARY KEY (`session_id`),
    UNIQUE KEY `session_id_idx` (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

3.25 Estructura de un paquete y buenas prácticas

Un paquete es un directorio que tiene una estructura bien definida y puede alojar cualquier cosa, desde clases hasta controladores y recursos web. A pesar de que los paquetes son tan flexibles, se deben seguir algunas recomendaciones si deseas distribuirlos.

3.25.1 Nombre de paquete

Un paquete también es un espacio de nombres PHP. El espacio de nombres debe seguir los estándares de interoperabilidad técnica de los espacios de nombres y nombres de clases de *PHP* 5.3: comienza con un segmento de proveedor, seguido por cero o más segmentos de categoría, y termina con el nombre corto del espacio de nombres, mismo que debe terminar con un sufijo `Bundle`.

Un espacio de nombres se convierte en un paquete tan pronto como se agrega una clase `bundle` al mismo. El nombre de la clase `bundle` debe seguir estas sencillas reglas:

- Solamente usa caracteres alfanuméricos y subraya;
- Utiliza un nombre con mayúsculas intercaladas;
- Usa un nombre corto y descriptivo (no más de 2 palabras);
- Prefija el nombre con la concatenación del proveedor (y opcionalmente la categoría del espacio de nombres);
- El nombre tiene el sufijo `Bundle`.

Estos son algunos espacios de nombres y nombres de clase `bundle` válidos:

Espacio de nombres	Nombre de clase `Bundle`
`Acme\Bundle\BlogBundle`	`AcmeBlogBundle`
`Acme\Bundle\Social\BlogBundle`	`AcmeSocialBlogBundle`
`Acme\BlogBundle`	`AcmeBlogBundle`

Por convención, el método `getNombre()` de la clase `bundle` debe devolver el nombre de la clase.

Nota: Si compartes tu paquete públicamente, debes utilizar el nombre de la clase `bundle` como nombre del repositorio (`AcmeBlogBundle` y no `BlogBundle` por ejemplo).

Nota: Los paquetes del núcleo de *Symfony2* no prefijan la clase `bundle` con `Symfony` y siempre agregan un subespacio de nombres `Bundle`, por ejemplo: `Symfony\Bundle\FrameworkBundle\FrameworkBundle`.

3.25.2 Estructura del directorio

La estructura básica del directorio del paquete `HolaBundle` se debe leer de la siguiente manera:

```
XXX/...
    HolaBundle/
        HolaBundle.php
        Controller/
        Resources/
            meta/
                LICENSE
            config/
            doc/
                index.rst
            translations/
            views/
            public/
        Tests/
```

Los directorios `XXX` reflejan la estructura del espacio de nombres del paquete.

Los siguientes archivos son obligatorios:

- `HolaBundle.php`;
- `Resources/meta/LICENSE`: La licencia completa del código;
- `Resources/doc/index.rst`: El archivo raíz de la documentación del paquete.

Nota: Estos convenios garantizan que las herramientas automatizadas pueden trabajar confiablemente en esta estructura predeterminada.

La profundidad de los subdirectorios se debe reducir al mínimo en la mayoría de las clases y archivos utilizados (2 niveles como máximo). Puedes definir más niveles para archivos no estratégicos, los menos utilizados.

El directorio del paquete es de sólo lectura. Si necesitas escribir archivos temporales, guárdalos en el directorio `cache/` o `log/` de la aplicación anfitriona. Las herramientas pueden generar archivos en la estructura de directorios del paquete, pero sólo si los archivos generados van a formar parte del repositorio.

Las siguientes clases y archivos tienen emplazamientos específicos:

Tipo	Directorio
Controladores	`Controller/`
Archivos de traducción	`Resources/translations/`
Plantillas	`Resources/views/`
Pruebas unitarias y funcionales	`Tests/`
Recursos Web	`Resources/public/`
Configuración	`Resources/config/`
Ordenes	`Command/`

3.25.3 Clases

La estructura del directorio de un paquete se utiliza como la jerarquía del espacio de nombres. Por ejemplo, un controlador `HolaController` se almacena en `/HolaBundle/Controller/HolaController.php` y el nombre de clase completamente cualificado es `Bundle\HolaBundle\Controller\HolaController`.

Todas las clases y archivos deben seguir los *estándares* (Página 629) de codificación *Symfony2*.

Algunas clases se deben ver como fachada y deben ser lo más breves posible, al igual que las ordenes, ayudantes, escuchas y controladores.

Las clases que conectan el Evento al Despachador deben llevar el posfijo `Listener`.

Las clases de excepciones se deben almacenar en un subespacio de nombres `Exception`.

3.25.4 Terceros

Un paquete no debe integrar bibliotecas *PHP* de terceros. Se debe confiar en la carga automática estándar de *Symfony2* en su lugar.

Un paquete no debería integrar bibliotecas de terceros escritas en *JavaScript*, *CSS* o cualquier otro lenguaje.

3.25.5 Pruebas

Un paquete debe venir con un banco de pruebas escritas con *PHPUnit* , las cuales se deben almacenar en el directorio `Test/`. Las pruebas deben seguir los siguientes principios:

- El banco de pruebas se debe ejecutar con una simple orden `PHPUnit` desde una aplicación de ejemplo;

- Las pruebas funcionales sólo se deben utilizar para probar la salida de la respuesta y alguna información del perfil si tienes alguno;
- La cobertura de código debe cubrir cuando menos el 95 % del código base.

Nota: Un banco de pruebas no debe contener archivos `AllTests.php`, sino que se debe basar en la existencia de un archivo `phpunit.xml.dist`.

3.25.6 Documentación

Todas las clases y funciones deben venir con *PHPDoc* completo.

También deberá proporcionar abundante documentación provista en formato *reStructuredText* (Página 634), bajo el directorio `Resources/doc/`, el archivo `Resources/doc/index.rst` es el único archivo obligatorio.

3.25.7 Controladores

Como práctica recomendada, los controladores en un paquete que está destinado a ser distribuido a otros no debe extender la clase base `Symfony\Bundle\FrameworkBundle\Controller\Controller`. Puede implementar la `Symfony\Component\DependencyInjection\ContainerAwareInterface` o en su lugar extender la clase `Symfony\Component\DependencyInjection\ContainerAware`.

Nota: Si echas un vistazo a los métodos de la clase `Symfony\Bundle\FrameworkBundle\Controller\Controller`, podrás ver que sólo son buenos accesos directos para facilitar la curva de aprendizaje.

3.25.8 Plantillas

Si un paquete proporciona plantillas, debe utilizar Twing. Un paquete no debe proporcionar un diseño principal, salvo si ofrece una aplicación completa.

3.25.9 Archivos de traducción

Si un paquete proporciona traducción de mensajes, se deben definir en formato XLIFF, el dominio se debe nombrar después del nombre del paquete (`bundle.hola`).

Un paquete no debe reemplazar los mensajes de otro paquete existente.

3.25.10 Configurando

Para proporcionar mayor flexibilidad, un paquete puede proporcionar opciones configurables utilizando los mecanismos integrados de *Symfony2*.

Para ajustes de configuración simples, confía en los `parámetros` predeterminados de la configuración de *Symfony2*. Los parámetros de *Symfony2* simplemente son pares clave/valor; un valor es cualquier valor PHP válido. Cada nombre de parámetro debe comenzar con una versión corta en minúscula del nombre del paquete usando guiones bajos (`acme_hola` para `AcmeHolaBundle` o `acme_social_blog` para `Acme\Social\BlogBundle` por ejemplo) , aunque esto sólo es una sugerencia de buenas prácticas. El resto del nombre del parámetro utiliza un punto (`.`) para separar las diferentes partes (por ejemplo, `acme_hola.correo.from`).

El usuario final puede proporcionar valores en cualquier archivo de configuración:

- *YAML*

```
# app/config/config.yml
parameters:
    acme_hola.correo.from: fabien@ejemplo.com
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="acme_hola.correo.from">fabien@ejemplo.com</parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$contenedor->setParameter('acme_hola.correo.from', 'fabien@ejemplo.com');
```

- *INI*

```
[parameters]
acme_hola.correo.from = fabien@ejemplo.com
```

Recupera los parámetros de configuración en tu código desde el contenedor:

```
$contenedor->getParameter('acme_hola.correo.from');
```

Incluso si este mecanismo es bastante simple, te animamos a usar la configuración semántica descrita en el recetario.

3.25.11 Aprende más en el recetario

- *Cómo exponer la configuración semántica de un paquete* (Página 350)

3.26 Cómo utilizar la herencia de paquetes para redefinir partes de un paquete

Este artículo no se ha escrito todavía, pero será muy pronto. Si estás interesado en escribir esta entrada, consulta *Colaborando en la documentación* (Página 633).

Este tema es para mostrar cómo se puede hacer que un paquete "extienda" a otro y utilizar este para redefinir diferentes aspectos de ese paquete.

3.27 Cómo exponer la configuración semántica de un paquete

Si abres el archivo de configuración de tu aplicación (por lo general `app/config/config.yml`), puedes encontrar una serie de configuraciones de diferentes "espacios de nombres", como `framework`, `twig` y `doctrine`. Cada una de estas configura un paquete específico, lo cual te permite configurar las cosas a nivel superior y luego dejar que el paquete haga todo lo de bajo nivel, haciendo los cambios complejos que resulten.

Por ejemplo, el siguiente fragmento le dice a `FrameworkBundle` que habilite la integración de formularios, lo cual implica la definición de unos cuantos servicios, así como la integración de otros componentes relacionados:

- *YAML*

```
framework:
    # ...
    form:             true
```

- *XML*

```
<framework:config>
    <framework:form />
</framework:config>
```

- *PHP*

```
$contenedor->loadFromExtension('framework', array(
    // ...
    'form'             => true,
    // ...
));
```

Cuando creas un paquete, tienes dos opciones sobre cómo manejar la configuración:

1. **Configuración normal del servicio** (*fácil*):

   Puedes especificar tus servicios en un archivo de configuración (por ejemplo, `services.yml`) que vive en tu paquete y luego importarlo desde la configuración principal de tu aplicación. Esto es realmente fácil, rápido y completamente eficaz. Si usas *parámetros* (Página 239), entonces todavía tienes cierta flexibilidad para personalizar el paquete desde la configuración de tu aplicación. Consulta "*Importando configuración con imports* (Página 241)" para más detalles.

2. **Exponiendo la configuración semántica** (*avanzado*):

   Esta es la forma de configuración que se hace con los paquetes básicos (como se describió anteriormente). La idea básica es que, en lugar de permitir al usuario sustituir parámetros individuales, permites al usuario configurar unos cuantos, en concreto la creación de opciones. A medida que desarrollas el paquete, vas analizando la configuración y cargas tus servicios en una clase "Extensión". Con este método, no tendrás que importar ningún recurso de configuración desde la configuración principal de tu aplicación: la clase Extensión puede manejar todo esto.

La segunda opción - de la cual aprenderás en este artículo - es mucho más flexible, pero también requiere más tiempo de configuración. Si te preguntas qué método debes utilizar, probablemente sea una buena idea empezar con el método #1, y más adelante, si es necesario, cambiar al #2.

El segundo método tiene varias ventajas específicas:

- Es mucho más poderoso que la simple definición de parámetros: un valor de opción específico podría inducir la creación de muchas definiciones de servicios;
- La habilidad de tener jerarquías de configuración
- La fusión inteligente de varios archivos de configuración (por ejemplo, `config_dev.yml` y `config.yml`) sustituye los demás ajustes;
- Configurando la validación (si utilizas una *clase Configuración* (Página 357));
- Autocompletado en tu IDE cuando creas un XSD y los desarrolladores del IDE utilizan XML.

Sustituyendo parámetros del paquete

Si un paquete proporciona una clase *Extensión*, entonces, generalmente *no debes* reemplazar los parámetros del contenedor de servicios de ese paquete. La idea es que si está presente una clase Extensión, cada ajuste que deba ser configurable debe estar presente en la configuración disponible en esa clase. En otras palabras, la clase Extensión define como públicas todas las opciones de configuración apoyadas para las cuales, por mantenimiento, existe compatibilidad hacia atrás.

3.27.1 Creando una clase Extensión

Si eliges exponer una configuración semántica de tu paquete, primero tendrás que crear una nueva clase "Extensión", la cual debe manipular el proceso. Esta clase debe vivir en el directorio `DependencyInjection` de tu paquete y su nombre se debe construir sustituyendo el sufijo `Bundle` del nombre de clase del paquete con `Extension`. Por ejemplo, la clase Extensión de `AcmeHolaBundle` se llamaría `AcmeHolaExtension`:

```
// Acme/HolaBundle/DependencyInjection/HolaExtension.php
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class AcmeHolaExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $contenedor)
    {
        // dónde se lleva a cabo toda la lógica
    }

    public function getXsdValidationBasePath()
    {
        return __DIR__.'/../Resources/config/';
    }

    public function getNamespace()
    {
        return 'http://www.ejemplo.com/symfony/schema/';
    }
}
```

Nota: Los métodos `getXsdValidationBasePath` y `getNamespace` sólo son necesarios si el paquete opcional XSD proporciona la configuración.

La presencia de la clase anterior significa que ahora puedes definir una configuración de espacio de nombres `acme_hola` en cualquier archivo de configuración. El espacio de nombres `acme_hola` se construyó a partir de la extensión del nombre de la clase eliminando la palabra `Extensión` y a continuación, en minúsculas y un subrayando el resto del nombre. En otras palabras, `AcmeHolaExtension` se convierte en `acme_hola`.

Puedes empezar de inmediato, especificando la configuración en este espacio de nombres:

- *YAML*

  ```
  # app/config/config.yml
  acme_hola: ~
  ```

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hola="http://www.ejemplo.com/symfony/schema/"
    xsi:schemaLocation="http://www.ejemplo.com/symfony/schema/ http://www.ejemplo.com/symfony/sc

   <acme_hola:config />
   ...

</container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('acme_hola', array());
```

Truco: Si sigues las convenciones de nomenclatura mencionadas anteriormente, entonces el método `load()` el cual carga el código de tu extensión es llamado siempre que tu paquete sea registrado en el núcleo. En otras palabras, incluso si el usuario no proporciona ninguna configuración (es decir, la entrada `acme_hola` ni siquiera figura), el método `load()` será llamado y se le pasará una matriz `$configs` vacía. Todavía puedes proporcionar algunos parámetros predeterminados para tu paquete si lo deseas.

3.27.2 Analizando la matriz `$configs`

Cada vez que un usuario incluya el espacio de nombres `acme_hola` en un archivo de configuración, la configuración bajo este se agrega a una gran matriz de configuraciones y se pasa al método `load()` de tu extensión (*Symfony2* convierte automáticamente XML y YAML en una matriz).

Tomemos la siguiente configuración:

- *YAML*

```
# app/config/config.yml
acme_hola:
    foo: fooValue
    bar: barValue
```

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hola="http://www.ejemplo.com/symfony/schema/"
    xsi:schemaLocation="http://www.ejemplo.com/symfony/schema/ http://www.ejemplo.com/symfony/sc

    <acme_hola:config foo="fooValue">
        <acme_hola:bar>barValue</acme_hola:bar>
    </acme_hola:config>

</container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('acme_hola', array(
    'foo' => 'fooValue',
    'bar' => 'barValue',
));
```

La matriz pasada a tu método `load()` se verá así:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    )
)
```

Ten en cuenta que se trata de una *matriz de matrices*, y no sólo una única matriz plana con los valores de configuración. Esto es intencional. Por ejemplo, si `acme_hola` aparece en otro archivo de configuración - digamos en `config_dev.yml` - con diferentes valores bajo él, entonces la matriz entrante puede tener este aspecto:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    ),
    array(
        'foo' => 'fooDevValue',
        'baz' => 'newConfigEntry',
    ),
)
```

El orden de las dos matrices depende de cuál es el primer conjunto.

Entonces, es tu trabajo, decidir cómo se fusionan estas configuraciones. Es posible que, por ejemplo, después tengas que sustituir valores anteriores o alguna combinación de ellos.

Más tarde, en la sección *clase Configuración* (Página 357), aprenderás una forma realmente robusta para manejar esto. Pero por ahora, sólo puedes combinarlos manualmente:

```
public function load(array $configs, ContainerBuilder $contenedor)
{
    $config = array();
    foreach ($configs as $subConfig) {
        $config = array_merge($config, $subConfig);
    }

    // Ahora usa la matriz simple $config
}
```

Prudencia: Asegúrate de que la técnica de fusión anterior tenga sentido para tu paquete. Este es sólo un ejemplo, y debes tener cuidado de no usarlo a ciegas.

3.27.3 Usando el método `load()`

Dentro de `load()`, la variable `$contenedor` se refiere a un contenedor que sólo sabe acerca de esta configuración de espacio de nombres (es decir, no contiene información de los servicios cargados por otros paquetes). El objetivo del método `load()` es manipular el contenedor, añadir y configurar cualquier método o servicio necesario por tu paquete.

Cargando la configuración de recursos externos

Una de las cosas comunes por hacer es cargar un archivo de configuración externo que puede contener la mayor parte de los servicios que necesita tu paquete. Por ejemplo, supongamos que tienes un archivo `services.xml` el cual contiene gran parte de la configuración de los servicios en tu paquete:

```
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\Config\FileLocator;

public function load(array $configs, ContainerBuilder $contenedor)
{
    // prepara tu variable $configs

    $cargador = new XmlFileLoader($contenedor, new FileLocator(__DIR__.'/../Resources/config'));
    $cargador->load('services.xml');
}
```

Incluso lo podrías hacer condicionalmente, basándote en uno de los valores de configuración. Por ejemplo, supongamos que sólo deseas cargar un conjunto de servicios si una opción `habilitado` es pasada y fijada en `true`:

```
public function load(array $configs, ContainerBuilder $contenedor)
{
    // prepara tu variable $configs

    $cargador = new XmlFileLoader($contenedor, new FileLocator(__DIR__.'/../Resources/config'));

    if (isset($config['enabled']) && $config['enabled']) {
        $cargador->load('services.xml');
    }
}
```

Configurando servicios y ajustando parámetros

Una vez que hayas cargado alguna configuración de servicios, posiblemente necesites modificar la configuración basándote en alguno de los valores entrantes. Por ejemplo, supongamos que tienes un servicio cuyo primer argumento es una cadena "tipo" utilizada internamente. Quisieras que el usuario del paquete lo configurara fácilmente, por lo que en el archivo de configuración de tu servicio (por ejemplo, `services.xml`), defines este servicio y utilizas un parámetro en blanco - `acme_hola.my_service_options` - como primer argumento:

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services

    <parameters>
        <parameter key="acme_hola.my_service_type" />
    </parameters>

    <services>
        <service id="acme_hola.my_service" class="Acme\HolaBundle\MyService">
            <argument>%acme_hola.my_service_type%</argument>
        </service>
    </services>
</container>
```

Pero ¿por qué definir un parámetro vacío y luego pasarlo a tu servicio? La respuesta es que vas a establecer este parámetro en tu clase Extensión, basándote en los valores de configuración entrantes. Supongamos, por ejemplo, que

deseas permitir al usuario definir esta opción de *tipo* en una clave denominada `my_type`. Para hacerlo agrega lo siguiente al método `load()`:

```
public function load(array $configs, ContainerBuilder $contenedor)
{
    // prepara tu variable $configs

    $cargador = new XmlFileLoader($contenedor, new FileLocator(__DIR__.'/../Resources/config'));
    $cargador->load('services.xml');

    if (!isset($config['my_type'])) {
        throw new \InvalidArgumentException('The "my_type" option must be set');
    }

    $contenedor->setParameter('acme_hola.my_service_type', $config['my_type']);
}
```

Ahora, el usuario puede configurar eficientemente el servicio especificando el valor de configuración `my_type`:

- *YAML*

  ```
  # app/config/config.yml
  acme_hola:
      my_type: foo
      # ...
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <?xml version="1.0" ?>

  <container xmlns="http://symfony.com/schema/dic/services"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:acme_hola="http://www.ejemplo.com/symfony/schema/"
      xsi:schemaLocation="http://www.ejemplo.com/symfony/schema/ http://www.ejemplo.com/symfony/sc

      <acme_hola:config my_type="foo">
          <!-- ... -->
      </acme_hola:config>

  </container>
  ```

- *PHP*

  ```
  // app/config/config.php
  $contenedor->loadFromExtension('acme_hola', array(
      'my_type' => 'foo',
      // ...
  ));
  ```

Parámetros globales

Cuando configures el contenedor, tienes que estar consciente de que los siguientes parámetros globales están disponibles para que los utilices:

- `kernel.name`
- `kernel.environment`
- `kernel.debug`

- kernel.root_dir
- kernel.cache_dir
- kernel.logs_dir
- kernel.bundle_dirs
- kernel.bundles
- kernel.charset

Prudencia: Todos los nombres de los parámetros y servicios que comienzan con un subrayado _ están reservados para la plataforma, y no los debes definir en tus nuevos paquetes.

3.27.4 Validación y fusión con una clase configuración

Hasta ahora, has fusionado manualmente las matrices de configuración y las has comprobado por medio de la presencia de los valores de configuración utilizando la función `isset()` de *PHP*. También hay disponible un sistema de *configuración* opcional, el cual puede ayudar con la fusión, validación, valores predeterminados y normalización de formato.

Nota: *Normalización de formato* se refiere al hecho de que ciertos formatos - en su mayoría XML - resultan en matrices de configuración ligeramente diferentes, y que estas matrices se deben "normalizar" para que coincidan con todo lo demás.

Para aprovechar las ventajas de este sistema, debes crear una clase `Configuración` y construir un árbol que define tu configuración en esa clase:

```
// src/Acme/HolaBundle/DependencyExtension/Configuration.php
namespace Acme\HolaBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hola');

        $rootNode
            ->children()
                ->scalarNode('my_type')->defaultValue('bar')->end()
            ->end()
        ;

        return $treeBuilder;
    }
```

Se trata de un ejemplo *muy* sencillo, pero ahora puedes utilizar esta clase en el método `load()` para combinar tu configuración y forzar su validación. Si se pasan las demás opciones salvo `my_type`, el usuario recibirá una notificación con una excepción de que se ha pasado una opción no admitida:

```
use Symfony\Component\Config\Definition\Processor;
// ...
```

```
public function load(array $configs, ContainerBuilder $contenedor)
{
    $processor = new Processor();
    $configuration = new Configuration();
    $config = $processor->processConfiguration($configuration, $configs);

    // ...
}
```

El método `processConfiguration()` utiliza el árbol de configuración que has definido en la clase `Configuración` y lo utilizas para validar, normalizar y fusionar todas las matrices de configuración.

La clase `Configuración` puede ser mucho más complicada de lo que se muestra aquí, apoyando matrices de nodos, nodos "prototipo", validación avanzada, normalización XML específica y fusión avanzada. La mejor manera de ver esto en acción es revisando algunas de las clases configuración del núcleo, como la configuración del FrameworkBundle o la configuración del TwigBundle.

3.27.5 Convenciones de extensión

Al crear una extensión, sigue estas simples convenciones:

- La extensión se debe almacenar en el subespacio de nombres `DependencyInjection`;
- La extensión se debe nombrar después del nombre del paquete y con el sufijo `Extension` (`AcmeHolaExtension` para `AcmeHolaBundle`);
- La extensión debe proporcionar un esquema XSD.

Si sigues estas simples convenciones, *Symfony2* registrará automáticamente las extensiones. Si no es así, sustituye el método **:method:'Symfony\\Component\\HttpKernel\\Bundle\\Bundle::build'** en tu paquete:

```
use Acme\HolaBundle\DependencyInjection\ExtensionHola;

class AcmeHolaBundle extends Bundle
{
    public function build(ContainerBuilder $contenedor)
    {
        parent::build($contenedor);

        // registra manualmente las extensiones que no siguen la convención
        $contenedor->registerExtension(new ExtensionHola());
    }
}
```

En este caso, la clase Extensión también debe implementar un método `getAlias()` que devuelva un alias único nombrado después del paquete (por ejemplo, `acme_hola`). Esto es necesario porque el nombre de clase no sigue la norma de terminar en `Extension`.

Además, el método `load()` de tu extensión *sólo* se llama si el usuario especifica el alias `acme_hola` en por lo menos un archivo de configuración. Una vez más, esto se debe a que la clase Extensión no se ajusta a las normas establecidas anteriormente, por lo tanto nada sucede automáticamente.

3.28 Cómo enviar correo electrónico

El envío de correo electrónico es una tarea clásica para cualquier aplicación web, y la cual tiene complicaciones especiales y peligros potenciales. En lugar de recrear la rueda, una solución para enviar mensajes de correo electrónico

es usando `SwiftmailerBundle`, la cual aprovecha el poder de la biblioteca SwiftMailer.

Nota: No olvides activar el paquete en tu núcleo antes de usarlo:

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

3.28.1 Configurando

Antes de usar SwiftMailer, asegúrate de incluir su configuración. El único parámetro de configuración obligatorio es `transport`:

- *YAML*

```
# app/config/config.yml
swiftmailer:
    transport:  smtp
    encryption: ssl
    auth_mode:  login
    host:       smtp.gmail.com
    username:   your_username
    password:   your_password
```

- *XML*

```
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    transport="smtp"
    encryption="ssl"
    auth-mode="login"
    host="smtp.gmail.com"
    username="your_username"
    password="your_password" />
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('swiftmailer', array(
    'transport'  => "smtp",
    'encryption' => "ssl",
    'auth_mode'  => "login",
    'host'       => "smtp.gmail.com",
    'username'   => "your_username",
```

```
        'password'   => "your_password",
    ));
```

La mayoría de los atributos de configuración de SwiftMailer tratan con la forma en que se deben entregar los mensajes.

Los atributos de configuración disponibles son las siguientes:

- `transport` (`smtp`, `mail`, `sendmail` o `gmail`)
- `username`
- `password`
- `host`
- `port`
- `encryption` (`tls` o `ssl`)
- `auth_mode` (`plain`, `login` o `cram-md5`)
- `spool`
  - `type` (cómo formar los mensajes, actualmente sólo es compatible con `file`)
  - `path` (dónde almacenar los mensajes)
- `delivery_domicilio` (una dirección de correo electrónico de donde enviar todos los mensajes)
- `disable_delivery` (fija a true para desactivar la entrega por completo)

3.28.2 Enviando correo electrónico

La biblioteca *SwiftMailer* trabaja creando, configurando y luego enviando objetos `Swift_Message`. El "mailer" es responsable de la entrega real del mensaje y es accesible a través del servicio `mailer`. En general, el envío de un correo electrónico es bastante sencillo:

```
public function indexAction($nombre)
{
    $mensaje = \Swift_Message::newInstance()
        ->setSubject('Hola correo-e')
        ->setFrom('send@ejemplo.com')
        ->setTo('recipient@ejemplo.com')
        ->setCuerpo($this->renderView('HolaBundle:Hola:email.txt.twig', array('nombre' => $nombre)))
    ;
    $this->get('mailer')->send($mensaje);

    return $this->render(...);
}
```

Para mantener las cosas disociadas, el cuerpo del correo electrónico se ha almacenado en una plantilla y reproducido con el método `RenderView()`.

El objeto `$message` admite muchas más opciones, como incluir archivos adjuntos, agregar contenido *HTML*, y mucho más. Afortunadamente, SwiftMailer cubre el tema con gran detalle en Creando mensajes de su documentación.

Truco: Hay disponibles varios artículos en el recetario relacionados con el envío de mensajes de correo electrónico en *Symfony2*:

- *Cómo utilizar Gmail para enviar mensajes de correo electrónico* (Página 361)
- *Cómo trabajar con correos electrónicos durante el desarrollo* (Página 361)

- *Cómo organizar el envío de correo electrónico* (Página 363)

3.29 Cómo utilizar Gmail para enviar mensajes de correo electrónico

Durante el desarrollo, en lugar de utilizar un servidor SMTP regular para enviar mensajes de correo electrónico, verás que es más fácil y más práctico utilizar Gmail. El paquete SwiftMailer hace que sea muy fácil.

Truco: En lugar de utilizar tu cuenta normal de Gmail, por supuesto, recomendamos crear una cuenta especial.

En el archivo de configuración de desarrollo, cambia el ajuste `transporte` a `gmail` y establece el `nombre de usuario` y `contraseña` a las credenciales de Google:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    transport: gmail
    username:  your_gmail_username
    password:  your_gmail_password
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    transport="gmail"
    username="your_gmail_username"
    password="your_gmail_password" />
```

- *PHP*

```
// app/config/config_dev.php
$contenedor->loadFromExtension('swiftmailer', array(
    'transport' => "gmail",
    'username'  => "your_gmail_username",
    'password'  => "your_gmail_password",
));
```

¡Ya está!

Nota: El transporte `gmail` simplemente es un acceso directo que utiliza el transporte `smtp` y establece `encryption`, `auth_mode` y `host` para trabajar con Gmail.

3.30 Cómo trabajar con correos electrónicos durante el desarrollo

Cuando estás creando una aplicación que envía mensajes de correo electrónico, a menudo, mientras desarrollas, no quieres enviar realmente los correos electrónicos al destinatario especificado. Si estás utilizando el

`SwiftmailerBundle` con *Symfony2*, puedes lograr fácilmente esto a través de ajustes de configuración sin tener que realizar ningún cambio en el código de tu aplicación en absoluto. Hay dos opciones principales cuando se trata del manejo de correos electrónicos durante el desarrollo: (a) desactivar el envío de correos electrónicos por completo o (b) enviar todos los mensajes de correo electrónico a una dirección especificada.

3.30.1 Desactivando el envío

Puedes desactivar el envío de correos electrónicos estableciendo la opción `disable_delivery` a `true`. Este es el predeterminado en el entorno `test` de la distribución estándar. Si haces esto en la configuración específica `test`, los mensajes de correo electrónico no se enviarán cuando se ejecutan pruebas, pero se seguirán enviando en los entornos `prod` y `dev`:

- *YAML*

```
# app/config/config_test.yml
swiftmailer:
    disable_delivery:  true
```

- *XML*

```
<!-- app/config/config_test.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    disable-delivery="true" />
```

- *PHP*

```
// app/config/config_test.php
$contenedor->loadFromExtension('swiftmailer', array(
    'disable_delivery'  => "true",
));
```

Si también deseas inhabilitar el envío en el entorno `dev`, sólo tienes que añadir esta configuración en el archivo `config_dev.yml`.

3.30.2 Enviando a una dirección específica

También puedes optar por que todos los correos sean enviados a una dirección específica, en vez de la dirección real especificada cuando se envía el mensaje. Esto se puede hacer a través de la opción `delivery_domicilio`:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    delivery_domicilio:  dev@ejemplo.com
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
```

```
    http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
    -->

    <swiftmailer:config
        delivery-domicilio="dev@ejemplo.com" />
```

- *PHP*

```
// app/config/config_dev.php
$contenedor->loadFromExtension('swiftmailer', array(
    'delivery_domicilio'  => "dev@ejemplo.com",
));
```

Ahora, supongamos que estás enviando un correo electrónico a `recipiente@ejemplo.com`.

```
public function indexAction($nombre)
{
    $mensaje = \Swift_Message::newInstance()
        ->setSubject('Hola correo-e')
        ->setFrom('send@ejemplo.com')
        ->setTo('recipient@ejemplo.com')
        ->setCuerpo($this->renderView('HolaBundle:Hola:email.txt.twig', array('nombre' => $nombre)))
    ;
    $this->get('mailer')->send($mensaje);

    return $this->render(...);
}
```

En el entorno `dev`, el correo electrónico será enviado a `dev@ejemplo.com`. `SwiftMailer` añadirá un encabezado adicional para el correo electrónico, `X-Swift-To` conteniendo la dirección reemplazada, por lo tanto todavía serás capaz de ver que se habría enviado.

Nota: Además de las direcciones `para`, también se detendrá el correo electrónico que se envíe a cualquier dirección `CC` y `BCC` establecida. `SwiftMailer` agregará encabezados adicionales al correo electrónico con las direcciones reemplazada en ellos. Estas son `X-Swift-CC` y `X-Swift-CCO` para las direcciones `CC` y `BCC`, respectivamente.

3.30.3 Visualizando desde la barra de depuración web

Puedes ver cualquier correo electrónico enviado por una página cuando estás en el entorno `dev` usando la barra de depuración web. El icono de correo electrónico en la barra de herramientas mostrará cuántos correos electrónicos fueron enviados. Si haces clic en él, se abrirá un informe mostrando los detalles de los mensajes de correo electrónico.

Si estás enviando un correo electrónico e inmediatamente después lo desvías, tendrás que establecer la opción `intercept_redirects` a `true` en el archivo `config_dev.yml` para que puedas ver el correo electrónico en la barra de depuración antes de que sea redirigido.

3.31 Cómo organizar el envío de correo electrónico

Cuando estás utilizando el `SwiftmailerBundle` para enviar correo electrónico desde una aplicación *Symfony2*, de manera predeterminada el mensaje será enviado inmediatamente. Sin embargo, posiblemente quieras evitar el impacto en el rendimiento de la comunicación entre `SwiftMailer` y el transporte de correo electrónico, lo cual podría hacer que el usuario tuviera que esperar la carga de la siguiente página, mientras que se envía el correo electrónico. Puedes evitar todo esto eligiendo "carrete", los correos electrónicos en lugar de enviarlos directamente. Esto significa que

`SwiftMailer` no intenta enviar el correo electrónico, sino que guarda el mensaje en alguna parte, tal como un archivo. Otro proceso puede leer de la cola y hacerse cargo de enviar los correos que están organizados en la cola. Actualmente, sólo cola de archivo es compatible con `SwiftMailer`.

Para utilizar la cola, utiliza la siguiente configuración:

- *YAML*

```yaml
# app/config/config.yml
swiftmailer:
    # ...
    spool:
        type: file
        path: /ruta/a/la/cola
```

- *XML*

```xml
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config>
     <swiftmailer:spool
         type="file"
         path="/ruta/a/la/cola" />
</swiftmailer:config>
```

- *PHP*

```php
// app/config/config.php
$contenedor->loadFromExtension('swiftmailer', array(
     // ...
    'spool' => array(
        'type' => 'file',
        'path' => '/ruta/a/la/cola',
    )
));
```

Truco: Si deseas almacenar la cola de correo en algún lugar en el directorio de tu proyecto, recuerda que puedes utilizar el parámetro *%kernel.root_dir %* para referirte a la raíz del proyecto:

```
path: %kernel.root_dir %/cola
```

Ahora, cuando tu aplicación envía un correo electrónico, no se enviará realmente, sino que se añade a la cola de correo. El envío de los mensajes desde la cola se hace por separado. Hay una orden de consola para enviar los mensajes en la cola de correo:

```
php app/console swiftmailer:spool:send
```

Tiene una opción para limitar el número de mensajes que se enviarán:

```
php app/console swiftmailer:spool:send --message-limit=10
```

También puedes establecer el límite de tiempo en segundos:

```
php app/console swiftmailer:spool:send --time-limit=10
```

Por supuesto que en realidad no deseas ejecutar esto manualmente. En cambio, la orden de consola se debe activar por un trabajo cronometrado o tarea programada y ejecutarse a intervalos regulares.

3.32 Cómo simular autenticación *HTTP* en una prueba funcional

Si tu aplicación necesita autenticación *HTTP*, pasa el nombre de usuario y contraseña como variables del servidor a `createClient()`:

```
$cliente = static::createClient(array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

También lo puedes sustituir en base a la petición:

```
$cliente->request('DELETE', '/comunicado/12', array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

3.33 Cómo probar la interacción de varios clientes

Si necesitas simular la interacción entre diferentes clientes (piensa en un chat, por ejemplo), crea varios clientes:

```
$harry = static::createClient();
$sally = static::createClient();

$harry->request('POST', '/say/sally/Hola');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hola/', $sally->getResponse()->getContent());
```

Esto funciona, excepto cuando el código mantiene un estado global o si depende de bibliotecas de terceros que tienen algún tipo de estado global. En tal caso, puedes aislar a tus clientes:

```
$harry = static::createClient();
$sally = static::createClient();

$harry->insulate();
$sally->insulate();

$harry->request('POST', '/say/sally/Hola');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hola/', $sally->getResponse()->getContent());
```

Los clientes con aislamiento transparente ejecutan sus peticiones en un proceso PHP específico y limpio, evitando así efectos secundarios.

Truco: Como un cliente con aislamiento es más lento, puedes mantener a un cliente en el proceso principal, y aislar a los demás.

3.34 Cómo utilizar el generador de perfiles en una prueba funcional

Es altamente recomendable que una prueba funcional sólo pruebe la respuesta. Pero si escribes pruebas funcionales que controlan los servidores en producción, posiblemente desees escribir pruebas en los datos del generador de perfiles, ya que te da una gran manera de ver diferentes cosas y hacer cumplir algunas métricas.

El `Profiler` de *Symfony2* reúne una gran cantidad de datos para cada petición. Utiliza estos datos para comprobar el número de llamadas a la base de datos, el tiempo invertido en la plataforma, ... Pero antes de escribir aserciones, siempre verifica que el generador de perfiles realmente está disponible (está activado por omisión en el entorno de prueba —`test`):

```
class HolaControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $cliente = static::createClient();
        $impulsor = $cliente->request('GET', '/hola/Fabien');

        // Escribe algunas afirmaciones sobre Response
        // ...

        // Comprueba que el generador de perfiles esté activado
        if ($perfil = $cliente->getProfile()) {
            // comprueba el número de peticiones
            $this->assertTrue($perfil->get('db')->getQueryCount() < 10);

            // comprueba el tiempo gastado en la plataforma
            $this->assertTrue( $perfil->get('timer')->getTime() < 0.5);
        }
    }
}
```

Si una prueba falla debido a los datos del generador de perfiles (demasiadas consultas a la BD, por ejemplo), posiblemente desees utilizar el Generador de perfiles Web para analizar la petición después de terminar las pruebas. Es fácil conseguirlo si incorporas el símbolo en el mensaje de error:

```
$this->assertTrue(
    $perfil->get('db')->getQueryCount() < 30,
    sprintf('Checks that query count is less than 30 (token %s)', $perfil->getToken())
);
```

Prudencia: El almacén del generador de perfiles puede ser diferente en función del entorno (sobre todo si utilizas el almacén de datos SQLite, el cual es el valor configurado por omisión).

Nota: La información del generador de perfiles está disponible incluso si aíslas al cliente o si utilizas una capa HTTP para tus pruebas.

Truco: Lee la *API* para incorporar *colectores de datos* (Página 425) para aprender más acerca de tus interfaces.

3.35 Cómo probar repositorios *Doctrine*

Probar unidades de repositorios *Doctrine* en un proyecto *Symfony* no es una tarea sencilla. De hecho, para cargar un repositorio es necesario cargar las entidades, un gestor para la entidad, y algunas otras cosas como una conexión.

Para probar tu repositorio, tienes dos diferentes opciones:

1. **Prueba de funcionamiento**: Esta incluye el uso de una conexión de base de datos real con los objetos de la base de datos real. Es fácil instalarla y puedes probar cualquier cosa, pero es de muy lenta ejecución. Consulta *Probando la funcionalidad* (Página 369).
2. **Prueba de unidad**: La prueba de unidad es de ejecución más rápida y más precisa en la forma de probar. Esta requiere una configuración un poco más pequeña, la cual cubre este documento. También puedes probar sólo los métodos que, por ejemplo, crean consultas, no los métodos que se ejecutan realmente.

3.35.1 Pruebas unitarias

Puesto que *Symfony* y *Doctrine* comparten la misma plataforma de pruebas, es muy fácil implementar las pruebas unitarias en un proyecto *Symfony*. El *ORM* viene con su propio conjunto de herramientas para facilitar las pruebas unitarias y simular todo lo que necesites, tal como una conexión, un gestor de entidades, etc. Al usar los componentes de prueba proporcionados por *Doctrine* - junto con algunas configuraciones básicas - puedes aprovechar las herramientas de *Doctrine* para probar unidades de tus repositorios.

Ten en cuenta que si deseas probar la ejecución real de tus consultas, necesitarás una prueba de funcionamiento (consulta *Probando la funcionalidad* (Página 369)). Las pruebas unitarias sólo son posibles cuando pruebas un método que construye una consulta.

Configurando

En primer lugar, necesitas agregar el espacio de nombres `Doctrine\Tests` a tu cargador automático:

```
// app/autoload.php
$loader->registerNamespaces(array(
    //...
    'Doctrine\\Tests'                => __DIR__.'/../vendor/doctrine/tests',
));
```

En seguida, tendrás que configurar un gestor de entidades en cada prueba para que *Doctrine* pueda cargar entidades y repositorios por ti.

Debido a que *Doctrine* por omisión no es capaz de cargar los metadatos de anotaciones desde tus entidades, tendrás que configurar el lector de anotaciones para que pueda analizar y cargar las entidades:

```
// src/Acme/ProductoBundle/Tests/Entity/ProductoRepositoryTest.php
namespace Acme\ProductoBundle\Tests\Entity;

use Doctrine\Tests\OrmTestCase;
use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\ORM\Mapping\Driver\DriverChain;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;

class ProductoRepositoryTest extends OrmTestCase
{
    private $_em;

    protected function setUp()
```

```
    {
        $lector = new AnnotationReader();
        $lector->setIgnoreNotImportedAnnotations(true);
        $lector->setEnableParsePhpImports(true);

        $controladorMetadatos = new AnnotationDriver(
            $lector,
            // proporciona el espacio de nombres de las entidades que deseas probar
            'Acme\\ProductoBundle\\Entity'
        );

        $this->_em = $this->_getTestEntityManager();

        $this->_em->getConfiguration()
            ->setMetadataDriverImpl($controladorMetadatos);

        // te permite utilizar la sintaxis AcmeProductoBundle:Producto
        $this->_em->getConfiguration()->setEntityNamespaces(array(
            'AcmeProductoBundle' => 'Acme\\ProductoBundle\\Entity'
        ));
    }
}
```

Si te fijas en el código, puedes notar que:

- Extiendes desde `\Doctrine\Tests\OrmTestCase`, el cual proporciona útiles métodos para las pruebas unitarias;
- Debes configurar el `AnnotationReader` para poder analizar y cargar las entidades;
- Creas el gestor de entidades llamando a `_getTestEntityManager`, el cual devuelve un gestor de entidades simulado con una conexión simulada.

¡Eso es todo! Ya estás listo para escribir las pruebas unitarias para tus repositorios de *Doctrine*.

Escribiendo tus pruebas unitarias

Recuerda que los métodos de repositorio de *Doctrine* sólo pueden probar si estás construyendo y devolviendo una consulta (pero no ejecutando una consulta realmente). Tomemos el siguiente ejemplo:

```
// src/Acme/AcmeTiendaBundle/Entity/ProductoRepository
namespace Acme\TiendaBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductoRepository extends EntityRepository
{
    public function creaBusquedaPorNombreQueryBuilder($nombre)
    {
        return $this->createQueryBuilder('p')
            ->where('p.nombre LIKE :nombre', $nombre)
    }
}
```

En este ejemplo, el método devuelve una instancia de `QueryBuilder`. Puedes probar el resultado de este método en una variedad de maneras:

```
class ProductoRepositoryTest extends \Doctrine\Tests\OrmTestCase
{
```

```
    /* ... */

    public function testCreaBusquedaPorNombreQueryBuilder()
    {
        $constructorDeConsulta = $this->_em->getRepository('AcmeProductoBundle:Producto')
            ->creaBusquedaPorNombreQueryBuilder('foo');

        $this->assertEquals('p.nombre LIKE :nombre', (string) $constructorDeConsulta->getDqlPart('whe
        $this->assertEquals(array('nombre' => 'foo'), $constructorDeConsulta->getParameters());
    }
}
```

En esta prueba, diseccionas el objeto `QueryBuilder`, buscando que cada parte sea como tú esperas. Si estuvieras agregando otras cosas al generador de consultas, podrías verificar las partes *DQL*: `select`, `from`, `join`, `set`, `groupBy`, `having` u `orderBy`.

Si sólo tienes un objeto `Query` crudo o prefieres probar la consulta real, puedes probar la cadena de consulta *DQL* directamente:

```
public function testCreaBusquedaPorNombreQueryBuilder()
{
    $constructorDeConsulta = $this->_em->getRepository('AcmeProductoBundle:Producto')
        ->creaBusquedaPorNombreQueryBuilder('foo');

    $consulta = $constructorDeConsulta->getQuery();

    // prueba DQL
    $this->assertEquals(
        'SELECT p FROM Acme\ProductoBundle\Entity\Producto p WHERE p.nombre LIKE :nombre',
        $consulta->getDql()
    );
}
```

3.35.2 Probando la funcionalidad

Si realmente necesitas ejecutar una consulta, tendrás que arrancar el `núcleo` para conseguir una conexión válida. En este caso, debes extender a `WebTestCase`, el cual hace todo esto muy fácil:

```
// src/Acme/ProductoBundle/Tests/Entity/ProductoRepositoryFunctionalTest.php
namespace Acme\ProductoBundle\Tests\Entity;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ProductoRepositoryFunctionalTest extends WebTestCase
{
    /**
     * @var \Doctrine\ORM\EntityManager
     */
    private $_em;

    public function setUp()
    {
            $kernel = static::createKernel();
            $kernel->boot();
        $this->_em = $kernel->getContainer()
            ->get('doctrine.orm.entity_manager');
    }
```

```
    public function testProductoPorNombreDeCategoria()
    {
        $resultados = $this->_em->getRepository('AcmeProductoBundle:Producto')
            ->buscaProductosPorNombreQuery('foo')
            ->getResult();

        $this->assertEquals(count($resultados), 1);
    }
}
```

3.36 Cómo agregar la funcionalidad "recuérdame" al inicio de sesión

Una vez que un usuario está autenticado, normalmente sus credenciales se almacenan en la sesión. Esto significa que cuando termina la sesión se desechará la sesión y tienes que proporcionar de nuevo tus datos de acceso la próxima vez que desees acceder a la aplicación. Puedes permitir a tus usuarios que puedan optar por permanecer conectados durante más tiempo del que dure la sesión con una `cookie` con la opción `remember_me` del cortafuegos. El cortafuegos necesita tener configurada una clave secreta, la cual se utiliza para cifrar el contenido de la `cookie`. También tiene varias opciones con los valores predeterminados que se muestran a continuación:

- *YAML*

```
# app/config/security.yml
firewalls:
    main:
        remember_me:
            key:      aSecretKey
            lifetime: 3600
            path:     /
            domain:   ~ # El valor predeterminado es el dominio actual de $_SERVER
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <remember-me
            key="aSecretKey"
            lifetime="3600"
            path="/"
            domain="" <!-- El valor predeterminado es el dominio actual de $_SERVER -->
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('remember_me' => array(
            'key'                    => '/login_check',
            'lifetime'               => 3600,
            'path'                   => '/',
            'domain'                 => '', // El valor predeterminado es el dominio actual de
        )),
```

```
        ),
    ));
```

Es buena idea ofrecer al usuario la opción de utilizar o no la funcionalidad recuérdame, ya que no siempre es adecuada. La forma habitual de hacerlo consiste en añadir una casilla de verificación en el formulario de acceso. Al dar a la casilla de verificación el nombre `_remember_me`, la `cookie` se ajustará automáticamente cuando la casilla esté marcada y el usuario inicia sesión satisfactoriamente. Por lo tanto, tu formulario de acceso específico en última instancia, podría tener este aspecto:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="nombreusuario">Nombre:</label>
    <input type="text" id="nombreusuario" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="nombreusuario">Nombre:</label>
    <input type="text" id="nombreusuario"
           name="_username" value="<?php echo $last_username ?>" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

El usuario entonces, se registra automáticamente en las subsecuentes visitas, mientras que la `cookie` sea válida.

3.36.1 Forzando al usuario a volver a autenticarse antes de acceder a ciertos recursos

Cuando el usuario vuelve a tu sitio, se autentica automáticamente en función de la información almacenada en la `cookie` recuérdame. Esto permite al usuario acceder a recursos protegidos como si el usuario se hubiera autenticado en realidad al visitar el sitio.

En algunos casos, sin embargo, puedes obligar al usuario a realmente volver a autenticarse antes de acceder a ciertos recursos. Por ejemplo, podrías permitir a un usuario de "recuérdame" ver la información básica de la cuenta, pero luego obligarlo a volver a autenticarse realmente antes de modificar dicha información.

El componente de seguridad proporciona una manera fácil de hacerlo. Además de los roles asignados expresamente, a los usuarios se les asigna automáticamente uno de los siguientes roles, dependiendo de cómo se haya autenticado:

- `IS_AUTHENTICATED_ANONYMOUSLY` - asignado automáticamente a un usuario que está en una parte del sitio protegida por el cortafuegos, pero que no ha iniciado sesión. Esto sólo es posible si se le ha permitido el acceso anónimo.
- `IS_AUTHENTICATED_REMEMBERED` - asignado automáticamente a un usuario autenticado a través de una `cookie` recuérdame.
- `IS_AUTHENTICATED_FULLY` - asignado automáticamente a un usuario que haya proporcionado sus datos de acceso durante la sesión actual.

Las puedes utilizar para controlar el acceso más allá de los roles asignados explícitamente.

Nota: Si tienes el rol `IS_AUTHENTICATED_REMEMBERED`, entonces también tienes el rol `IS_AUTHENTICATED_ANONYMOUSLY`. Si tienes el rol `IS_AUTHENTICATED_FULLY`, entonces también tienes los otros dos roles. En otras palabras, estos roles representan tres niveles de "fortaleza" autenticación incremental.

Puedes utilizar estos roles adicionales para un control más granular sobre el acceso a partes de un sitio. Por ejemplo, posiblemente desees que el usuario pueda ver su cuenta en `/cuenta` cuando está autenticado por `cookie`, pero tiene que proporcionar sus datos de acceso para poder editar la información de la cuenta. Lo puedes hacer protegiendo acciones específicas del controlador usando estos roles. La acción de edición en el controlador se puede proteger usando el servicio contexto.

En el siguiente ejemplo, la acción sólo es permitida si el usuario tiene el rol `IS_AUTHENTICATED_FULLY`.

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException
// ...

public function editAction()
{
    if (false === $this->get('security.context')->isGranted(
        'IS_AUTHENTICATED_FULLY'
    )) {
        throw new AccessDeniedException();
    }

    // ...
}
```

También puedes optar por instalar y utilizar el opcional JMSSecurityExtraBundle, el cual puede proteger tu controlador usando anotaciones:

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="IS_AUTHENTICATED_FULLY")
 */
public function editAction($nombre)
{
    // ...
}
```

Truco: Si, además, hubiera un control de acceso en la configuración de seguridad que requiere que el usuario tenga un rol `ROLE_USER` a fin de acceder a cualquier área de la cuenta, entonces tendríamos la siguiente situación:

- Si un usuario no autenticado (o un usuario autenticado anónimamente) intenta acceder al área de la cuenta, el usuario se tendrá que autenticar.
- Una vez que el usuario ha introducido su nombre de usuario y contraseña, asumiendo que el usuario recibe el rol `ROLE_USER` de tu configuración, el usuario tendrá el rol `IS_AUTHENTICATED_FULLY` y podrá acceder a cualquier página en la sección de cuenta, incluyendo el controlador `editAction`.
- Si termina la sesión del usuario, cuando el usuario vuelve al sitio, podrá acceder a cada página de la cuenta - a excepción de la página de edición - sin verse obligado a volver a autenticarse. Sin embargo, cuando intenta acceder al controlador `editAction`, se verá obligado a volver a autenticarse, ya que no está, sin embargo, totalmente autenticado.

Para más información sobre proteger servicios o métodos de esta manera, consulta *Cómo proteger cualquier servicio o método de tu aplicación* (Página 389).

3.37 Cómo implementar tu propio votante para agregar direcciones IP a la lista negra

El componente `Security` de *Symfony2* ofrece varias capas para autenticar a los usuarios. Una de las capas se llama `voter` ("votante" en adelante). Un votante o es una clase dedicada que comprueba si el usuario tiene el derecho a conectarse con la aplicación. Por ejemplo, *Symfony2* proporciona una capa que comprueba si el usuario está plenamente autenticado o si se espera que tenga algún rol.

A veces es útil crear un votante personalizado para tratar un caso específico que la plataforma no maneja. En esta sección, aprenderás cómo crear un votante que te permitirá añadir usuarios a la lista negra por su IP.

3.37.1 La interfaz Votante

Un votante personalizado debe implementar la clase `Symfony\Component\Security\Core\Authorization\Voter\Vote` la cual requiere los tres siguientes métodos:

```
interface VoterInterface
{
    function supportsAttribute($attribute);
    function supportsClass($class);
    function vote(TokenInterface $muestra, $objeto, array $attributes);
}
```

El método `supportsAttribute()` se utiliza para comprobar si el votante admite el atributo usuario dado (es decir: un rol, una ACL ("access control list", en adelante: lista de control de acceso), etc.).

El método `supportsClass()` se utiliza para comprobar si el votante apoya a la clase simbólica usuario actual.

El método `vote()` debe implementar la lógica del negocio que verifica cuando o no se concede acceso al usuario. Este método debe devolver uno de los siguientes valores:

- `VoterInterface::ACCESS_GRANTED`: El usuario puede acceder a la aplicación
- `VoterInterface::ACCESS_ABSTAIN`: El votante no puede decidir si se concede acceso al usuario o no
- `VoterInterface::ACCESS_DENIED`: El usuario no está autorizado a acceder a la aplicación

En este ejemplo, vamos a comprobar si la dirección IP del usuario coincide con una lista de direcciones en la lista negra. Si la IP del usuario está en la lista negra, devolveremos `VoterInterface::ACCESS_DENIED`, de lo contrario devolveremos `VoterInterface::ACCESS_ABSTAIN` porque la finalidad del votante sólo es para negar el acceso, no para permitir el acceso.

3.37.2 Creando un votante personalizado

Para poner a un usuario en la lista negra basándonos en su IP, podemos utilizar el servicio `Petición` y comparar la dirección IP contra un conjunto de direcciones IP en la lista negra:

```
namespace Acme\DemoBundle\Security\Authorization\Voter;

use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\Security\Core\Authorization\Voter\VoterInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

class ClientIpVoter implements VoterInterface
{
    public function __construct(ContainerInterface $container, array $blacklistedIp = array())
    {
        $this->container     = $container;
        $this->blacklistedIp = $blacklistedIp;
    }

    public function supportsAttribute($attribute)
    {
        // no vamos a verificar contra un atributo de usuario, por lo
        // tanto devuelve true
        return true;
    }

    public function supportsClass($class)
    {
        // nuestro votante apoya todo tipo de clases, por lo que
        // devuelve true
        return true;
    }

    function vote(TokenInterface $token, $objeto, array $attributes)
    {
        $peticion = $this->container->get('request');
        if (in_array($this->request->getClientIp(), $this->blacklistedIp)) {
            return VoterInterface::ACCESS_DENIED;
        }

        return VoterInterface::ACCESS_ABSTAIN;
    }
}
```

¡Eso es todo! El votante está listo. El siguiente paso es inyectar el votante en el nivel de seguridad. Esto se puede hacer fácilmente a través del contenedor de servicios.

3.37.3 Declarando el votante como servicio

Para inyectar al votante en la capa de seguridad, se debe declarar como servicio, y la etiqueta como "security.voter":

- *YAML*

```
# src/Acme/AcmeBundle/Resources/config/services.yml

services:
    security.access.blacklist_voter:
        class:      Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter
        arguments:  [@service_container, [123.123.123.123, 171.171.171.171]]
        public:     false
        tags:
            -       { name: security.voter }
```

- *XML*

```
<!-- src/Acme/AcmeBundle/Resources/config/services.xml -->

<service id="security.access.blacklist_voter"
         class="Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter" public="false">
    <argument type="service" id="service_container" strict="false" />
    <argument type="collection">
        <argument>123.123.123.123</argument>
        <argument>171.171.171.171</argument>
    </argument>
    <tag name="security.voter" />
</service>
```

- *PHP*

```
// src/Acme/AcmeBundle/Resources/config/services.php

use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

$definition = new Definition(
    'Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter',
    array(
        new Reference('service_container'),
        array('123.123.123.123', '171.171.171.171'),
    ),
);
$definition->addTag('security.voter');
$definition->setPublic(false);

$contenedor->setDefinition('security.access.blacklist_voter', $definicion);
```

Truco: Asegúrate de importar este archivo de configuración desde el archivo de configuración principal de tu aplicación (por ejemplo, `app/config/config.yml`). Para más información, consulta *Importando configuración con imports* (Página 241). Para leer más acerca de definir los servicios en general, consulta el capítulo *Contenedor de servicios* (Página 237).

3.37.4 Cambiando la estrategia de decisión de acceso

A fin de que los cambios del nuevo votante tengan efecto, tenemos que cambiar la estrategia de decisión de acceso predeterminada, que, por omisión, concede el acceso si *cualquier* votante permite el acceso.

En nuestro caso, vamos a elegir la estrategia `unánime`. A diferencia de la estrategia `afirmativa` (predeterminada), con la estrategia `unánime`, aunque un votante sólo niega el acceso (por ejemplo, el `ClientIpVoter`), no otorga

acceso al usuario final.

Para ello, sustituye la sección `access_decision_manager` predeterminada del archivo de configuración de tu aplicación con el siguiente código.

- *YAML*

```
# app/config/security.yml
security:
    access_decision_manager:
        # Strategy puede ser: affirmative, unanimous o consensus
        strategy: unanimous
```

¡Eso es todo! Ahora, a la hora de decidir si un usuario debe tener acceso o no, el nuevo votante deniega el acceso a cualquier usuario en la lista negra de direcciones IP.

3.38 Listas de control de acceso (ACL)

En aplicaciones complejas, a menudo te enfrentas al problema de que las decisiones de acceso no se pueden basar únicamente en la persona (`Ficha`) que está solicitando el acceso, sino también implica un objeto dominio al cual se está solicitando acceso. Aquí es donde entra en juego el sistema ACL.

Imagina que estás diseñando un sistema de blog donde los usuarios pueden comentar tus mensajes. Ahora, deseas que un usuario pueda editar sus propios comentarios, pero no los de otros usuarios, además, tú mismo quieres ser capaz de editar todos los comentarios. En este escenario, `Comentario` sería nuestro objeto dominio al cual deseas restringir el acceso. Podrías tomar varios enfoques para lograr esto usando *Symfony2*, dos enfoques básicos (no exhaustivos) son:

- *Hacer cumplir la seguridad en tus métodos de negocio*: Básicamente, significa mantener una referencia dentro de cada `comentario` a todos los usuarios que tienen acceso, y luego comparar estos usuarios a la `Ficha` provista.
- *Hacer cumplir la seguridad con roles*: En este enfoque, debes agregar un rol a cada objeto `comentario`, es decir, `ROLE_COMMENT_1`, `ROLE_COMMENT_2`, etc.

Ambos enfoques son perfectamente válidos. Sin embargo, su pareja lógica de autorización a tu código del negocio lo hace menos reutilizable en otros lugares, y también aumenta la dificultad de las pruebas unitarias. Además, posiblemente tengas problemas de rendimiento si muchos usuarios tuvieran acceso a un único objeto dominio.

Afortunadamente, hay una manera mejor, de la cual vamos a hablar ahora.

3.38.1 Proceso de arranque

Ahora, antes de que finalmente puedas entrar en acción, tenemos que hacer algún proceso de arranque. En primer lugar, tenemos que configurar la conexión al sistema ACL que se supone vamos a emplear:

- *YAML*

```
# app/config/security.yml
security:
    acl:
        connection: default
```

- *XML*

```
<!-- app/config/security.xml -->
<acl>
    <connection>default</connection>
</acl>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', 'acl', array(
    'connection' => 'default',
));
```

Nota: El sistema ACL requiere al menos configurar una conexión DBAL con *Doctrine*. Sin embargo, eso no significa que tengas que utilizar *Doctrine* para asignar tus objetos del dominio. Puedes usar cualquier asignador de objetos que te guste, ya sea el *ORM* de *Doctrine*, Mongo ODM, Propel, o `SQL` crudo, la elección es tuya.

Después de configurar la conexión, tenemos que importar la estructura de la base de datos. Afortunadamente, tenemos una tarea para eso. Basta con ejecutar la siguiente orden:

```
php app/console init:acl
```

3.38.2 Cómo empezar

Volviendo a nuestro pequeño ejemplo desde el principio, vamos a implementar ACL para ello.

Crea una ACL, y añade una ACE

```
// BlogController.php
public function addCommentAction(Comunicado $comunicado)
{
    $comment = new Comment();

    // configura $formulario, y vincula datos
    // ...

    if ($formulario->isValid()) {
        $entityManager = $this->get('doctrine.orm.default_entity_manager');
        $entityManager->persist($comment);
        $entityManager->flush();

        // creando la ACL
        $aclProvider = $this->get('security.acl.provider');
        $objectIdentity = ObjectIdentity::fromDomainObject($comment);
        $acl = $aclProvider->createAcl($objectIdentity);

        // recupera la identidad de seguridad del usuario registrado actual
        $securityContext = $this->get('security.context');
        $user = $securityContext->getToken()->getUser();
        $securityIdentity = UserSecurityIdentity::fromAccount($user);

        // otorga permiso de propietario
        $acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
        $aclProvider->updateAcl($acl);
    }
}
```

Hay un par de decisiones de implementación importantes en este fragmento de código. Por ahora, sólo quiero destacar dos:

En primer lugar, te habrás dado cuenta de que `->createAcl()` no acepta objetos de dominio directamente, sino sólo implementaciones de `ObjectIdentityInterface`. Este paso adicional de indirección te permite trabajar con ACL, incluso cuando no tienes a mano ninguna instancia real del objeto dominio. Esto será muy útil si deseas comprobar los permisos de un gran número de objetos sin tener que hidratar estos objetos.

La otra parte interesante es la llamada a `->insertObjectAce()`. En nuestro ejemplo, estamos otorgando al usuario que ha iniciado sesión acceso de propietario al comentario. La `MaskBuilder::MASK_OWNER` es una máscara predefinida de bits enteros; no te preocupes que el constructor de la máscara debe abstraer la mayoría de los detalles técnicos, pero gracias a esta técnica puedes almacenar muchos permisos diferentes en la fila de la base de datos lo cual nos da un impulso considerable en cuanto a rendimiento.

Truco: El orden en que las ACE son revisadas es significativo. Como regla general, debes poner más entradas específicas al principio.

Comprobando el acceso

```
// BlogController.php
public function editCommentAction(Comment $comment)
{
    $securityContext = $this->get('security.context');

    // comprueba el acceso para edición
    if (false === $securityContext->isGranted('EDIT', $comment))
    {
        throw new AccessDeniedException();
    }

    // recupera el objeto comentario actual, y realiza tu edición aquí
    // ...
}
```

En este ejemplo, comprobamos si el usuario tiene el permiso de `EDICIÓN`. Internamente, *Symfony2* asigna el permiso a varias máscaras de bits enteros, y comprueba si el usuario tiene alguno de ellos.

Nota: Puedes definir hasta 32 permisos base (dependiendo de tu sistema operativo, *PHP* puede variar entre 30 a 32). Además, también puedes definir permisos acumulados.

3.38.3 Permisos acumulados

En nuestro primer ejemplo anterior, sólo concedemos al usuario el permiso `OWNER` base. Si bien este además permite efectivamente al usuario realizar cualquier operación, como ver, editar, etc., sobre el objeto dominio, hay casos en los que deseas conceder estos permisos de forma explícita.

El `MaskBuilder` se puede utilizar para crear máscaras de bits fácilmente combinando varios permisos base:

```
$generador = new MaskBuilder();
$generador
    ->add('view')
    ->add('edit')
    ->add('delete')
    ->add('undelete')
;
$mask = $generador->get(); // int(15)
```

Esta máscara de bits de enteros, entonces se puede utilizar para conceder a un usuario los permisos base que se añaden por encima:

```
$acl->insertObjectAce(new UserSecurityIdentity('johannes'), $mask);
```

El usuario ahora puede ver, editar, borrar, y recuperar objetos eliminados.

3.39 Conceptos ACL avanzados

El objetivo de este capítulo es dar una visión en mayor profundidad del sistema ACL, y también explicar algunas de las decisiones de diseño detrás de él.

3.39.1 Conceptos de diseño

La capacidad de la instancia del objeto seguridad de *Symfony2* está basada en el concepto de una Lista de Control de Acceso. Cada **instancia** del objeto dominio tiene su propia ACL. La instancia de ACL contiene una detallada lista de las entradas de control de acceso (ACE) utilizada para tomar decisiones de acceso. El sistema ACL de *Symfony2* se enfoca en dos objetivos principales:

- proporcionar una manera eficiente de recuperar una gran cantidad de ACL/ACE para tus objetos dominio, y para modificarlos;
- proporcionar una manera de facilitar las decisiones de si a una persona se le permite realizar una acción en un objeto dominio o no.

Según lo indicado por el primer punto, una de las principales capacidades del sistema ACL de *Symfony2* es una forma de alto rendimiento de recuperar las ACL/ACE. Esto es muy importante ya que cada ACL puede tener varias ACE, y heredar de otra ACL anterior en una forma de árbol. Por lo tanto, específicamente no aprovechamos cualquier *ORM*, pero la implementación predeterminada interactúa con tu conexión directamente usando *DBAL Doctrine*.

Identidades de objeto

El sistema ACL está disociado completamente de los objetos de tu dominio. Ni siquiera se tienen que almacenar en la misma base de datos, o en el mismo servidor. Para lograr esta disociación, en el sistema ACL los objetos son representados a través de objetos identidad objeto. Cada vez, que desees recuperar la ACL para un objeto dominio, el sistema ACL en primer lugar crea un objeto identidad de tu objeto dominio y, a continuación pasa esta identidad de objeto al proveedor de ACL para su posterior procesamiento.

Identidad de seguridad

Esto es análogo a la identidad de objeto, pero representa a un usuario o un rol en tu aplicación. Cada rol, o usuario tiene una identidad de seguridad propia.

3.39.2 Estructura de tabla en la base de datos

La implementación predeterminada usa cinco tablas de bases de datos enumeradas a continuación. Las tablas están ordenadas de menos filas a más filas en una aplicación típica:

- *acl_security_identities*: Esta tabla registra todas las identidades de seguridad (SID) de que dispone ACE. La implementación predeterminada viene con dos identidades de seguridad: `RoleSecurityIdentity` y `UserSecurityIdentity`

- *acl_classes*: Esta tabla asigna los nombres de clase a un identificador único el cual puede hacer referencia a otras tablas.
- *acl_object_identities*: Cada fila de esta tabla representa una única instancia del objeto dominio.
- *acl_object_identity_ancestors*: Esta tabla nos permite determinar todos los antepasadosde una ACL de una manera muy eficiente.
- *acl_entries*: Esta tabla contiene todas las ACE. Esta suele ser la tabla con más filas. Puede contener decenas de millones sin impactar significativamente en el rendimiento.

3.39.3 Alcance de las entradas de control de acceso

Las entradas del control de acceso pueden tener diferente ámbito en el cual se aplican. En *Symfony2*, básicamente tenemos dos diferentes ámbitos:

- Class-Scope: Estas entradas se aplican a todos los objetos con la misma clase.
- Object-Scope: Este fue el ámbito de aplicación utilizado únicamente en el capítulo anterior, y sólo se aplica a un objeto específico.

A veces, encontraras la necesidad de aplicar una ACE sólo a un campo específico del objeto. Digamos que deseas que el `ID` sólo sea visto por un gestor, pero no por tu servicio al cliente. Para resolver este problema común, hemos añadido dos subámbitos:

- Class-Field-Scope: Estas entradas se aplican a todos los objetos con la misma clase, pero sólo a un campo específico de los objetos.
- Object-Field-Scope: Estas entradas se aplican a un objeto específico, y sólo a un campo específico de ese objeto.

3.39.4 Decisiones de preautorización

Para las decisiones de preautorización, es decir, las decisiones antes de que el método o la acción de seguridad se invoque, confiamos en el servicio provisto `AccessDecisionManager` que también se utiliza para tomar decisiones de autorización basadas en roles. Al igual que los roles, el sistema ACL añade varios nuevos atributos que se pueden utilizar para comprobar diferentes permisos.

Mapa de permisos incorporados

Atributo	Significado previsto	Máscara de Bits
VIEW	Cuando le es permitido a alguien ver el objeto dominio.	VIEW, EDIT, OPERATOR, MASTER u OWNER
EDIT	Cuando le es permitido a alguien hacer cambios al objeto dominio.	EDIT, OPERATOR, MASTER, u OWNER
DELETE	Cuando le es permitido a alguien eliminar el objeto dominio.	DELETE, OPERATOR, MASTER u OWNER
UNDELETE	Cuando le es permitido a alguien restaurar un objeto dominio previamente eliminado.	UNDELETE, OPERATOR, MASTER u OWNER
OPERATOR	Cuando le es permitido a alguien realizar todas las acciones anteriores.	OPERATOR, MASTER u OWNER
MASTER	Cuando le es permitido a alguien realizar todas las acciones anteriores, y además tiene permitido conceder cualquiera de los permisos anteriores a otros.	MASTER u OWNER
OWNER	Cuando alguien es dueño del objeto dominio. un propietario puede realizar cualquiera de las acciones anteriores.	OWNER

Atributos de permisos frente a máscaras de bits de permisos

Los atributos los utiliza el `AccessDecisionManager`, al igual que los roles son los atributos utilizados por `AccessDecisionManager`. A menudo, estos atributos en realidad representan un conjunto de enteros como máscaras de bits. Las máscaras de bits de enteros en cambio, las utiliza el sistema ACL interno para almacenar de manera eficiente los permisos de los usuarios en la base de datos, y realizar comprobaciones de acceso mediante las operaciones muy rápidas de las máscaras de bits.

Extensibilidad

El mapa de permisos citado más arriba no es estático, y, teóricamente, lo podrías reemplazar a voluntad por completo. Sin embargo, debería abarcar la mayoría de los problemas que encuentres, y para interoperabilidad con otros paquetes, te animamos a que le adhieras el significado que tienes previsto para ellos.

3.39.5 Decisiones de postautorización

Las decisiones de postautorización se realizan después de haber invocado a un método seguro, y por lo general implican que el objeto dominio es devuelto por este método. Después de invocar a los proveedores también te permite modificar o filtrar el objeto dominio antes de devolverlo.

Debido a las limitaciones actuales del lenguaje PHP, no hay capacidad de postautorización integradas en el núcleo del componente seguridad. Sin embargo, hay un JMSSecurityExtraBundle experimental que añade estas capacidades. Consulta su documentación para más información sobre cómo se logra esto.

3.39.6 Proceso para conseguir decisiones de autorización

La clase ACL proporciona dos métodos para determinar si una identidad de seguridad tiene la máscara de bits necesaria, `isGranted` y `isFieldGranted`. Cuando la ACL recibe una petición de autorización a través de uno de

estos métodos, delega esta petición a una implementación de `PermissionGrantingStrategy`. Esto te permite reemplazar la forma en que se tomen decisiones de acceso sin tener que modificar la clase ACL misma.

`PermissionGrantingStrategy` primero verifica todo su ámbito de aplicación ACE a objetos si no es aplicable, comprobará el ámbito de la clase ACE, si no es aplicable, entonces el proceso se repetirá con las ACE de la ACL padre. Si no existe la ACL padre, será lanzada una excepción.

3.40 Cómo forzar *HTTPS* o *HTTP* a diferentes *URL*

Puedes forzar áreas de tu sitio para que utilicen el protocolo `HTTPS` en la configuración de seguridad. Esto se hace a través de las reglas `access_control` usando la opción `requires_channel`. Por ejemplo, si deseas forzar que todas las *URL* que empiecen con `/seguro` para que usen *HTTPS* podrías utilizar la siguiente configuración:

- *YAML*

```
access_control:
    - path: ^/secure
      roles: ROLE_ADMIN
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/seguro" role="ROLE_ADMIN" requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/secure',
          'role' => 'ROLE_ADMIN',
          'requires_channel' => 'https'
    ),
),
```

Debes permitir al formulario de acceso en sí acceso anónimo de lo contrario los usuarios no se podrán autenticar. Para forzarlo a usar *HTTPS* puedes utilizar reglas `access_control` usando el rol `IS_AUTHENTICATED_ANONYMOUSLY`:

- *YAML*

```
access_control:
    - path: ^/login
      roles: IS_AUTHENTICATED_ANONYMOUSLY
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/login"
          role="IS_AUTHENTICATED_ANONYMOUSLY"
          requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login',
          'role' => 'IS_AUTHENTICATED_ANONYMOUSLY',
```

```
            'requires_channel' => 'https'
        ),
    ),
```

También es posible especificar el uso de *HTTPS* en la configuración de enrutado consulta *Cómo forzar las rutas para utilizar siempre HTTPS* (Página 279) para más detalles.

3.41 Cómo personalizar el formulario de acceso

Usar un *formulario de acceso* (Página 189) para autenticación es un método común y flexible para gestionar la autenticación en *Symfony2*. Casi todos los aspectos del formulario de acceso se pueden personalizar. La configuración predeterminada completa se muestra en la siguiente sección.

3.41.1 Referencia de configuración del formulario de acceso

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # el usuario es redirigido aquí cuando él/ella necesita ingresar
                login_path:                     /login

                # si es ''true'', reenvía al usuario al formulario de acceso en lugar de rediriq
                use_forward:                    false

                # presenta el formulario de acceso aquí
                check_path:                     /login_check

                # por omisión, el formulario de acceso DEBE ser POST, no GET
                post_only:                      true

                # opciones para redirigir en ingreso satisfactorio (lee más adelante)
                always_use_default_target_path: false
                default_target_path:            /
                target_path_parameter:          _target_path
                use_referer:                    false

                # opciones para redirigir en ingreso fallido (lee más adelante)
                failure_path:                   null
                failure_forward:                false

                # nombres de campo para el nombre de usuario y contraseña
                username_parameter:             _username
                password_parameter:             _password

                # opciones del elemento csrf
                csrf_parameter:                 _csrf_token
                intention:                      authenticate
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            check_path="/login_check"
            login_path="/login"
            use_forward="false"
            always_use_default_target_path="false"
            default_target_path="/"
            target_path_parameter="_target_path"
            use_referer="false"
            failure_path="null"
            failure_forward="false"
            username_parameter="_username"
            password_parameter="_password"
            csrf_parameter="_csrf_token"
            intention="authenticate"
            post_only="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'check_path'                    => '/login_check',
            'login_path'                    => '/login',
            'user_forward'                  => false,
            'always_use_default_target_path' => false,
            'default_target_path'           => '/',
            'target_path_parameter'         => _target_path,
            'use_referer'                   => false,
            'failure_path'                  => null,
            'failure_forward'               => false,
            'username_parameter'            => '_username',
            'password_parameter'            => '_password',
            'csrf_parameter'                => '_csrf_token',
            'intention'                     => 'authenticate',
            'post_only'                     => true,
        )),
    ),
));
```

3.41.2 Redirigiendo después del ingreso

Puedes cambiar el lugar al cual el formulario de acceso redirige después de un inicio de sesión satisfactorio utilizando diferentes opciones de configuración. Por omisión, el formulario redirigirá a la *URL* que el usuario solicitó (por ejemplo, la *URL* que provocó la exhibición del formulario de acceso). Por ejemplo, si el usuario solicitó `http://www.ejemplo.com/admin/post/18/editar` entonces después de que él/ella haya superado el inicio de sesión, finalmente será devuelto a `http://www.ejemplo.com/admin/post/18/editar`. Esto se consigue almacenando la *URL* solicitada en la sesión. Si no hay presente una *URL* en la sesión (tal vez el usuario se dirigió directamente a la página de inicio de sesión), entonces el usuario es redirigido a la página predeterminada, la cual es `/` (es decir, la página predeterminada del formulario de acceso). Puedes cambiar este comportamiento de varias

maneras.

Cambiando la página predeterminada

En primer lugar, la página predeterminada se puede configurar (es decir, la página a la cual el usuario es redirigido si no se almacenó una página previa en la sesión). Para establecer esta a `/admin` usa la siguiente configuración:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                default_target_path: /admin
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            default_target_path="/admin"
        />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            // ...
            'default_target_path' => '/admin',
        )),
    ),
));
```

Ahora, cuando no se encuentre una *URL* en la sesión del usuario, será enviado a `/admin`.

Redirigiendo siempre a la página predeterminada

Puedes hacer que los usuarios siempre sean redirigidos a la página predeterminada, independientemente de la *URL* que hayan solicitado con anterioridad, estableciendo la opción `always_use_default_target_path` a `true`:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                always_use_default_target_path: true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            always_use_default_target_path="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            // ...
            'always_use_default_target_path' => true,
        )),
    ),
));
```

Utilizando la *URL* referente

En caso de no haber almacenado una *URL* anterior en la sesión, posiblemente desees intentar usar el `HTTP_REFERER` en su lugar, ya que a menudo será el mismo. Lo puedes hacer estableciendo `use_referer` a `true` (el valor predeterminado es `false`):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                use_referer:        true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            use_referer="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            // ...
            'use_referer' => true,
        )),
```

```
    ),
));
```

Controlando la *URL* a redirigir desde el interior del formulario

También puedes sustituir a dónde es redirigido el usuario a través del formulario en sí mismo incluyendo un campo oculto con el nombre `_target_path`. Por ejemplo, para redirigir a la *URL* definida por alguna ruta `cuenta`, utiliza lo siguiente:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="nombreusuario">Nombre:</label>
    <input type="text" id="nombreusuario" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="cuenta" />

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="nombreusuario">Nombre:</label>
    <input type="text" id="nombreusuario" name="_username" value="<?php echo $last_username ?>"

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="cuenta" />

    <input type="submit" name="login" />
</form>
```

Ahora, el usuario será redirigido al valor del campo oculto del formulario. El valor del atributo puede ser una ruta relativa, una *URL* absoluta, o un nombre de ruta. Incluso, puedes cambiar el nombre del campo oculto en el formulario cambiando la opción `target_path_parameter` a otro valor.

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
```

```
            form_login:
                target_path_parameter: redirect_url
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            target_path_parameter="redirect_url"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'target_path_parameter' => redirect_url,
        )),
    ),
));
```

Redirigiendo en ingreso fallido

Además de redirigir al usuario después de un inicio de sesión, también puedes definir la *URL* a que el usuario debe ser redirigido después de un ingreso fallido (por ejemplo, si presentó un nombre de usuario o contraseña no válidos). Por omisión, el usuario es redirigido de nuevo al formulario de acceso. Puedes establecer este a una *URL* diferente con la siguiente configuración:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                failure_path: /login_failure
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            failure_path="login_failure"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'firewalls' => array(
```

```
            'main' => array('form_login' => array(
                // ...
                'failure_path' => login_failure,
            )),
        ),
    ));
```

3.42 Cómo proteger cualquier servicio o método de tu aplicación

En el capítulo sobre seguridad, puedes ver cómo *proteger un controlador* (Página 196) requiriendo el servicio `security.context` desde el Contenedor de servicios y comprobando el rol del usuario actual:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException
// ...

public function holaAction($nombre)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

También puedes proteger *cualquier* servicio de manera similar, inyectándole el servicio `security.context`. Para una introducción general a la inyección de dependencias en servicios consulta el capítulo *Contenedor de servicios* (Página 237) del libro. Por ejemplo, supongamos que tienes una clase `BoletinGestor` que envía mensajes de correo electrónico y deseas restringir su uso a únicamente los usuarios que tienen algún rol `ROLE_BOLETIN_ADMIN`. Antes de agregar la protección, la clase se ve algo parecida a esta:

```
namespace Acme\HolaBundle\Boletin;

class BoletinGestor
{

    public function sendNewsletter()
    {
        // donde realmente haces el trabajo
    }

    // ...
}
```

Nuestro objetivo es comprobar el rol del usuario cuando se llama al método `sendNewsletter()`. El primer paso para esto es inyectar el servicio `security.context` en el objeto. Dado que no tiene sentido *no* realizar la comprobación de seguridad, este es un candidato ideal para el constructor de inyección, lo cual garantiza que el objeto del contexto de seguridad estará disponible dentro de la clase `BoletinGestor`:

```
namespace Acme\HolaBundle\Boletin;

use Symfony\Component\Security\Core\SecurityContextInterface;

class BoletinGestor
{
    protected $securityContext;
```

```
    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    // ...
}
```

Luego, en tu configuración del servicio, puedes inyectar el servicio:

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
parameters:
    boletin_gestor.class: Acme\HolaBundle\Boletin\BoletinGestor

services:
    boletin_gestor:
        class:    %boletin_gestor.class%
        arguments: [@security.context]
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="boletin_gestor.class">Acme\HolaBundle\Boletin\BoletinGestor</parameter>
</parameters>

<services>
    <service id="boletin_gestor" class="%boletin_gestor.class%">
        <argument type="service" id="security.context"/>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

$contenedor->setParameter('boletin_gestor.class', 'Acme\HolaBundle\Boletin\BoletinGestor');

$contenedor->setDefinition('boletin_gestor', new Definition(
    '%boletin_gestor.class%',
    array(new Reference('security.context'))
));
```

El servicio inyectado se puede utilizar para realizar la comprobación de seguridad cuando se llama al método `sendNewsletter()`:

```
namespace Acme\HolaBundle\Boletin;

use Symfony\Component\Security\Core\Exception\AccessDeniedException
use Symfony\Component\Security\Core\SecurityContextInterface;
// ...

class BoletinGestor
{
    protected $securityContext;
```

```
    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    public function sendNewsletter()
    {
        if (false === $this->securityContext->isGranted('ROLE_BOLETIN_ADMIN')) {
            throw new AccessDeniedException();
        }

        //--
    }

    // ...
}
```

Si el usuario actual no tiene el rol `ROLE_BOLETIN_ADMIN`, se le pedirá que inicie sesión.

3.42.1 Protegiendo métodos usando anotaciones

También puedes proteger las llamadas a métodos en cualquier servicio con anotaciones usando el paquete opcional JMSSecurityExtraBundle. Este paquete está incluido en la Edición estándar de *Symfony2*.

Para habilitar la funcionalidad de las anotaciones, *etiqueta* (Página 251) el servicio que deseas proteger con la etiqueta `security.secure_service` (también puedes habilitar esta funcionalidad automáticamente para todos los servicios, consulta la *barra lateral* (Página 392) más adelante):

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/services.yml
# ...

services:
    boletin_gestor:
        # ...
        tags:
            -  { name: security.secure_service }
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/services.xml -->
<!-- ... -->

<services>
    <service id="boletin_gestor" class="%boletin_gestor.class%">
        <!-- ... -->
        <tag name="security.secure_service" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/HolaBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

$definicion = new Definition(
```

```
        '%boletin_gestor.class%',
        array(new Reference('security.context'))
    ));
    $definicion->addTag('security.secure_service');
    $contenedor->setDefinition('boletin_gestor', $definicion);
```

Entonces puedes obtener los mismos resultados que el anterior usando una anotación:

```
namespace Acme\HolaBundle\Boletin;

use JMS\SecurityExtraBundle\Annotation\Secure;
// ...

class BoletinGestor
{

    /**
     * @Secure(roles="ROLE_BOLETIN_ADMIN")
     */
    public function sendNewsletter()
    {
        //--
    }

    // ...
}
```

Nota: Las anotaciones trabajan debido a que se crea una clase sustituta para la clase que realiza las comprobaciones de seguridad. Esto significa que, si bien puedes utilizar las anotaciones sobre métodos públicos y protegidos, no las puedes utilizar con los métodos privados o los métodos marcados como finales.

El `JMSSecurityExtraBundle` también te permite proteger los parámetros y valores devueltos de los métodos. Para más información, consulta la documentación de JMSSecurityExtraBundle.

Activando la funcionalidad de anotaciones para todos los servicios

Cuando proteges el método de un servicio (como se muestra arriba), puedes etiquetar cada servicio individualmente, o activar la funcionalidad para *todos* los servicios a la vez. Para ello, establece la opción de configuración `secure_all_services` a `true`:

- *YAML*

```
# app/config/config.yml
jms_security_extra:
    # ...
    secure_all_services: true
```

- *XML*

```
<!-- app/config/config.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/se

    <jms_security_extra secure_controllers="true" secure_all_services="true" />

</srv:container>
```

- *PHP*

```
// app/config/config.php
$contenedor->loadFromExtension('jms_security_extra', array(
    // ...
    'secure_all_services' => true,
));
```

La desventaja de este método es que, si está activada, la carga de la página inicial puede ser muy lenta dependiendo de cuántos servicios hayas definido.

3.43 Cómo cargar usuarios de la base de datos con seguridad (la entidad Proveedor)

Este artículo no se ha escrito todavía, pero será muy pronto. Si estás interesado en escribir esta entrada, consulta *Colaborando en la documentación* (Página 633).

Este tema está destinado a ser un ejemplo completo de cómo usar el proveedor de la `entidad` usuario con el componente de seguridad. Este debe mostrar cómo crear la clase `Usuario`, implementando métodos, asignando, etc. - todo lo que necesites para conseguir una entidad totalmente funcional de proveedor de usuario que trabaje.

3.44 Cómo crear un proveedor de usuario personalizado

Este artículo no se ha escrito todavía, pero será muy pronto. Si estás interesado en escribir esta entrada, consulta *Colaborando en la documentación* (Página 633).

Este tema pretende mostrar cómo se puede crear un proveedor de usuario personalizado. Un ejemplo de potencial - aunque un mejor ejemplo será bienvenido - es mostrar cómo se puede crear una clase personalizada `Usuario` y luego rellenarla por medio de un proveedor de usuario personalizado que carguen los usuarios, haciendo una llamada a la *API* del servicio desde algún sitio externo.

3.45 Cómo crear un proveedor de autenticación personalizado

Si has leído el capítulo sobre *Seguridad* (Página 181), entiendes la distinción que *Symfony2* hace entre autenticación y autorización en la implementación de la seguridad. Este capítulo cubre las clases del núcleo involucradas en el proceso de autenticación, y cómo implementar un proveedor de autenticación personalizado. Dado que la autenticación y autorización son conceptos independientes, esta extensión será un proveedor agnóstico de usuario, y funcionará con los proveedores de usuario de la aplicación, posiblemente basado en memoria, en una base de datos, o en cualquier otro lugar que elijas almacenarlos.

3.45.1 Conociendo WSSE

El siguiente capítulo demuestra cómo crear un proveedor de autenticación personalizado para la autenticación WSSE. El protocolo de seguridad para WSSE proporciona varias ventajas de seguridad:

1. Cifrado del Nombre de usuario/Contraseña
2. Salvaguardado contra ataques repetitivos
3. No requiere configuración del servidor web

WSSE es muy útil para proteger servicios web, pudiendo ser SOAP o REST.

Hay un montón de excelente documentación sobre WSSE, pero este artículo no se enfocará en el protocolo de seguridad, sino más bien en la manera en que puedes personalizar el protocolo para añadirlo a tu aplicación *Symfony2*. La base de WSSE es que un encabezado de la petición comprueba si las credenciales están cifradas, verificando una marca de tiempo y nonce, y autenticado por el usuario de la petición asimilando una contraseña.

Nota: WSSE también es compatible con aplicaciones de validación de clave, lo cual es útil para los servicios web, pero está fuera del alcance de este capítulo.

3.45.2 El testigo

El papel del testigo en el contexto de la seguridad en *Symfony2* es muy importante. Un testigo representa los datos de autenticación del usuario en la petición. Una vez se ha autenticado una petición, el testigo conserva los datos del usuario, y proporciona estos datos a través del contexto de seguridad. En primer lugar, vamos a crear nuestra clase testigo. Esta permitirá pasar toda la información pertinente a nuestro proveedor de autenticación.

```
// src/Acme/DemoBundle/Security/Authentication/Token/WsseUserToken.php
namespace Acme\DemoBundle\Security\Authentication\Token;

use Symfony\Component\Security\Core\Authentication\Token\AbstractToken;

class WsseUserToken extends AbstractToken
{
    public $created;
    public $digest;
    public $nonce;

    public function getCredentials()
    {
        return '';
    }
}
```

Nota: La clase `WsseUserToken` extiende la clase componente `seguridad Symfony\Component\Security\Core\Autenticación\Token\AbstractToken`, que proporciona una funcionalidad de testigo básica. Implementa la clase `Symfony\Component\Security\Core\Authentication\Token\TokenInterface` en cualquier clase que se utiliza como testigo.

3.45.3 El escucha

Después, necesitas un escucha para que esté atento al contexto de seguridad. El escucha es el responsable de capturar las peticiones de seguridad al servidor e invocar al proveedor de autenticación. Un escucha debe ser una instancia de `Symfony\Component\Security\Http\Firewall\ListenerInterface`. Un escucha de seguridad debería manejar el evento `Symfony\Component\HttpKernel\Event\GetResponseEvent`, y establecer el testigo autenticado en el contexto de seguridad en caso de éxito.

```
// src/Acme/DemoBundle/Security/Firewall/WsseListener.php
namespace Acme\DemoBundle\Security\Firewall;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\Security\Http\Firewall\ListenerInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\SecurityContextInterface;
use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseListener implements ListenerInterface
{
    protected $securityContext;
    protected $authenticationManager;

    public function __construct(SecurityContextInterface $securityContext, AuthenticationManagerInter
    {
        $this->securityContext = $securityContext;
        $this->authenticationManager = $authenticationManager;
    }

    public function handle(GetResponseEvent $evento)
    {
        $peticion = $evento->getRequest();

        if (!$peticion->headers->has('x-wsse')) {
            return;
        }

        $wsseRegex = '/UsernameToken Username="([^"]+)", PasswordDigest="([^"]+)", Nonce="([^"]+)", C

        if (preg_match($wsseRegex, $peticion->headers->get('x-wsse'), $matches)) {
            $muestra = new WsseUserToken();
            $muestra->setUser($matches[1]);

            $muestra->digest   = $matches[2];
            $muestra->nonce    = $matches[3];
            $muestra->creado  = $matches[4];
```

```
            try {
                $returnValue = $this->authenticationManager->authenticate($muestra);

                if ($returnValue instanceof TokenInterface) {
                    return $this->securityContext->setToken($returnValue);
                } else if ($returnValue instanceof Response) {
                    return $evento->setResponse($returnValue);
                }
            } catch (AuthenticationException $e) {
                // aquí puedes registrar algo
            }
        }

        $respuesta = new Response();
        $respuesta->setStatusCode(403);
        $evento->setResponse($respuesta);
    }
}
```

Este escucha comprueba que la petición tenga la cabecera `X-WSSE` esperada, empareja el valor devuelto con la información `WSSE` esperada, crea un testigo utilizando esa información, y pasa el testigo al gestor de autenticación. Si no proporcionas la información adecuada, o el gestor de autenticación lanza una `Symfony\Component\Security\Core\Exception\AuthenticationException`, devuelve una respuesta 403.

Nota: Una clase no usada arriba, `Symfony\Component\Security\Http\Firewall\AbstractAuthenticationListe` es una clase base muy útil que proporciona funcionalidad necesaria comúnmente por las extensiones de seguridad. Esto incluye mantener al testigo en la sesión, proporcionando manipuladores de éxito / fallo, url del formulario de acceso y mucho más. Puesto que WSSE no requiere mantener la autenticación entre sesiones o formularios de acceso, no la utilizaremos para este ejemplo.

3.45.4 Proveedor de autenticación

El proveedor de autenticación debe hacer la verificación del `WsseUserToken`. Es decir, el proveedor verificará si es válido el valor de la cabecera `Creado` dentro de los cinco minutos, el valor de la cabecera `Nonce` es único dentro de los cinco minutos, y el valor de la cabecera `PasswordDigest` coincide con la contraseña del usuario.

```
// src/Acme/DemoBundle/Security/Authentication/Provider/WsseProvider.php
namespace Acme\DemoBundle\Security\Authentication\Provider;

use Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\Exception\NonceExpiredException;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseProvider implements AuthenticationProviderInterface
{
    private $userProvider;
    private $cacheDir;

    public function __construct(UserProviderInterface $userProvider, $cacheDir)
    {
        $this->userProvider = $userProvider;
```

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

    public function authenticate(TokenInterface $muestra)
    {
        $user = $this->userProvider->loadUserByUsername($muestra->getUsername());

        if ($usuario && $this->validateDigest($token->digest, $token->nonce, $token->created, $user->
            $authenticatedToken = new WsseUserToken($usuario->getRoles());
            $authenticatedToken->setUsurario($usuario);

            return $authenticatedToken;
        }

        throw new AuthenticationException('The WSSE authentication failed.');
    }

    protected function validateDigest($digest, $nonce, $created, $secret)
    {
        // la fecha y hora caduca después de 5 minutos
        if (time() - strtotime($created) > 300) {
            return false;
        }

        // Valida si nonce es único dentro de 5 minutos
        if (file_exists($this->cacheDir.'/'.$nonce) && file_get_contents($this->cacheDir.'/'.$nonce)
            throw new NonceExpiredException('Previously used nonce detected');
        }
        file_put_contents($this->cacheDir.'/'.$nonce, time());

        // Valida secreto
        $expected = base64_encode(sha1(base64_decode($nonce).$created.$secret, true));

        return $digest === $expected;
    }

    public function supports(TokenInterface $muestra)
    {
        return $muestra instanceof WsseUserToken;
    }
}
```

Nota: La `Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterf` requiere un método `authenticate` en el testigo del usuario, y un método `supports`, el cual informa al gestor de autenticación cuando o no utilizar este proveedor para el testigo dado. En el caso de múltiples proveedores, el gestor de autenticación entonces pasa al siguiente proveedor en la lista.

3.45.5 La fábrica

Has creado un testigo personalizado, escucha personalizado y proveedor personalizado. Ahora necesitas mantener todo junto. ¿Cómo hacer disponible tu proveedor en la configuración de seguridad? La respuesta es usando una `fábrica`. Una fábrica es donde enganchas el componente de seguridad, diciéndole el nombre de tu proveedor y las opciones de configuración disponibles para ello. En primer lugar, debes crear una clase que implemente `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface`

```
// src/Acme/DemoBundle/DependencyInjection/Security/Factory/WsseFactory.php
namespace Acme\DemoBundle\DependencyInjection\Security\Factory;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\DefinitionDecorator;
use Symfony\Component\Config\Definition\Builder\NodeDefinition;
use Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface;

class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $contenedor, $id, $config, $userProvider, $defaultEntryPo
    {
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $contenedor
            ->setDefinition($providerId, new DefinitionDecorator('wsse.security.authentication.provid
            ->replaceArgument(0, new Reference($userProvider))
        ;

        $listenerId = 'security.authentication.listener.wsse.'.$id;
        $escucha = $contenedor->setDefinition($listenerId, new DefinitionDecorator('wsse.security.aut

        return array($providerId, $listenerId, $defaultEntryPoint);
    }

    public function getPosition()
    {
        return 'pre_auth';
    }

    public function getKey()
    {
        return 'wsse';
    }

    public function addConfiguration(NodeDefinition $nodo)
    {}
}
```

La `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterfa` requiere los siguientes métodos:

- el método `create`, el cual añade el escucha y proveedor de autenticación para el contenedor de ID en el contexto de seguridad adecuado;
- el método `getPosition`, el cual debe ser del tipo `pre_auth`, `form`, `http` y `remember_me` define la posición en la que se llama el proveedor;
- el método `getKey` el cual define la clave de configuración utilizada para hacer referencia al proveedor;
- el método `addConfiguration` el cual se utiliza para definir las opciones de configuración bajo la clave de configuración en tu configuración de seguridad. Cómo ajustar las opciones de configuración se explica más adelante en este capítulo.

Nota: Una clase no utilizada en este ejemplo, `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\F` es una clase base muy útil que proporciona una funcionalidad común necesaria para proteger la fábrica. Puede ser útil en la definición de un tipo proveedor de autenticación diferente.

Ahora que has creado una clase fábrica, puedes utilizar la clave `wsse` como un cortafuegos en tu configuración de seguridad.

Nota: Te estarás preguntando "¿por qué necesitamos una clase fábrica especial para añadir escuchas y proveedores en el contenedor de inyección de dependencias?". Esta es una muy buena pregunta. La razón es que puedes utilizar tu cortafuegos varias veces, para proteger varias partes de tu aplicación. Debido a esto, cada vez que utilizas tu cortafuegos, se crea un nuevo servicio en el contenedor de ID. La fábrica es la que crea estos nuevos servicios.

3.45.6 Configurando

Es hora de ver en acción tu proveedor de autenticación. Tendrás que hacer algunas cosas a fin de hacerlo funcionar. Lo primero es añadir los servicios mencionados al contenedor de ID. Tu clase fábrica anterior hace referencia a identificadores de servicio que aún no existen: `wsse.security.authentication.provider` y `wsse.security.authentication.listener`. Es hora de definir esos servicios.

- *YAML*

```
# src/Acme/DemoBundle/Resources/config/services.yml
services:
  wsse.security.authentication.provider:
    class:  Acme\DemoBundle\Security\Authentication\Provider\WsseProvider
    arguments: [’’, %kernel.cache_dir%/security/nonces]

  wsse.security.authentication.listener:
    class:  Acme\DemoBundle\Security\Firewall\WsseListener
    arguments: [@security.context, @security.authentication.manager]
```

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/config/services.xml -->
<services>
    <service id="wsse.security.authentication.provider"
      class="Acme\DemoBundle\Security\Authentication\Provider\WsseProvider" public="false">
        <argument /> <!-- User Provider -->
        <argument>%kernel.cache_dir%/security/nonces</argument>
    </service>

    <service id="wsse.security.authentication.listener"
      class="Acme\DemoBundle\Security\Firewall\WsseListener" public="false">
        <argument type="service" id="security.context"/>
        <argument type="service" id="security.authentication.manager" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/DemoBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

$contenedor->setDefinition(’wsse.security.authentication.provider’,
  new Definition(
    ’Acme\DemoBundle\Security\Authentication\Provider\WsseProvider’,
    array(’’, ’%kernel.cache_dir%/security/nonces’)
));

$contenedor->setDefinition(’wsse.security.authentication.listener’,
```

```
        new Definition(
          'Acme\DemoBundle\Security\Firewall\WsseListener', array(
            new Reference('security.context'),
            new Reference('security.authentication.manager'))
    ));
```

Ahora que tus servicios están definidos, informa de tu fábrica al contexto de seguridad. Las fábricas se deben incluir en un archivo de configuración individual, al momento de escribir este artículo. Necesitas crear un archivo que incluya el servicio fábrica, y luego usar la clave `factories` en tu configuración para importarla.

```
<!-- src/Acme/DemoBundle/Resources/config/security_factories.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services

    <services>
        <service id="security.authentication.factory.wsse"
          class="Acme\DemoBundle\DependencyInjection\Security\Factory\WsseFactory" public="false">
            <tag name="security.listener.factory" />
        </service>
    </services>
</container>
```

- *YAML*

```
# app/config/security.yml
security:
  factories:
    - "%kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.xml"
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <factories>
      "%kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.xml
    </factories>
</config>
```

- *PHP*

```
// app/config/security.php
$contenedor->loadFromExtension('security', array(
    'factories' => array(
      "%kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.xml"
    ),
));
```

¡Y está listo! Ahora puedes definir las partes de tu aplicación como bajo la protección del WSSE.

```
security:
    firewalls:
        wsse_secured:
            pattern:   /api/.*
            wsse:      true
```

¡Enhorabuena! ¡Has escrito tu propio proveedor de autenticación de seguridad!

3.45.7 Un poco más allá

¿Qué hay de hacer de tu proveedor de autenticación WSSE un poco más emocionante? Las posibilidades son infinitas. ¿Por qué no empezar agregando algo de brillo a la pasta?

Configurando

Puedes añadir opciones personalizadas bajo la clave `wsse` en tu configuración de seguridad. Por ejemplo, el tiempo permitido antes de expirar el elemento de encabezado Creado, por omisión, es de 5 minutos. Hazlo configurable, por lo tanto distintos cortafuegos pueden tener diferentes magnitudes del tiempo de espera.

En primer lugar, tendrás que editar `WsseFactory` y definir la nueva opción en el método `addConfiguration`.

```
class WsseFactory implements SecurityFactoryInterface
{
    # ...

    public function addConfiguration(NodeDefinition $nodo)
    {
      $nodo
        ->children()
          ->scalarNode('lifetime')->defaultValue(300)
        ->end()
      ;
    }
}
```

Ahora, en el método `create` de la fábrica, el argumento `$config` contendrá una clave "lifetime", establecida en 5 minutos (300 segundos) a menos que se establezca en la configuración. Pasa este argumento a tu proveedor de autenticación a fin de utilizarlo.

```
class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $contenedor, $id, $config, $userProvider, $defaultEntryPo
    {
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $contenedor
            ->setDefinition($providerId,
              new DefinitionDecorator('wsse.security.authentication.provider'))
            ->replaceArgument(0, new Reference($userProvider))
            ->replaceArgument(2, $config['lifetime'])
        ;
        // ...
    }
    // ...
}
```

Nota: También tendrás que añadir un tercer argumento a la configuración del servicio `wsse.security.authentication.provider`, el cual puede estar en blanco, pero se completará con la vida útil en la fábrica. La clase `WsseProvider` ahora también tiene que aceptar un tercer argumento constructor - la vida útil - el cual se debe utilizar en lugar de los rígidos 300 segundos. Estos dos pasos no se muestran aquí.

La vida útil de cada petición wsse ahora es configurable y se puede ajustar a cualquier valor deseado por el cortafuegos.

```
security:
    firewalls:
```

```
        wsse_secured:
            pattern:   /api/.*
            wsse:      { lifetime: 30 }
```

¡El resto depende de ti! Todos los elementos de configuración correspondientes se pueden definir en la fábrica y consumirse o pasarse a las otras clases en el contenedor.

3.46 Cómo utilizar *Varnish* para acelerar mi sitio *web*

Debido a que la caché de *Symfony2* utiliza las cabeceras de caché *HTTP* estándar, el *Delegado inverso de Symfony2* (Página 212) se puede sustituir fácilmente por cualquier otro delegado inverso. *Varnish* es un potente acelerador HTTP, de código abierto, capaz de servir contenido almacenado en caché de forma rápida y es compatible con *Inclusión del borde lateral* (Página 220).

3.46.1 Configurando

Como vimos anteriormente, *Symfony2* es lo suficientemente inteligente como para detectar si está hablando con un delegado inverso que entiende ESI o no. Funciona fuera de la caja cuando utilizas el delegado inverso de *Symfony2*, pero necesita una configuración especial para que funcione con *Varnish*. Afortunadamente, *Symfony2* se basa en otra norma escrita por Akamaï (Arquitectura límite), por lo que el consejo de configuración en este capítulo te puede ser útil incluso si no utilizas *Symfony2*.

Nota: *Varnish* sólo admite el atributo `src` para las etiquetas ESI (los atributos `onerror` y `alt` se omiten).

En primer lugar, configura *Varnish* para que anuncie su apoyo a ESI añadiendo una cabecera `Surrogate-Capability` a las peticiones remitidas a la interfaz administrativa de tu aplicación:

```
sub vcl_recv {
    set req.http.Surrogate-Capability = "abc=ESI/1.0";
}
```

Entonces, optimiza *Varnish* para que sólo analice el contenido de la respuesta cuando al menos hay una etiqueta ESI comprobando la cabecera `Surrogate-Control` que *Symfony2* añade automáticamente:

```
sub vcl_fetch {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;

        // para Varnish >= 3.0
        set beresp.do_esi = true;
        // para Varnish < 3.0
        // esi;
    }
}
```

Prudencia: La compresión con *ESI* no cuenta con el apoyo de *Varnish* hasta la versión 3.0 (lee GZIP y Varnish). Si no estás utilizando *Varnish 3.0*, coloca un servidor web frente a *Varnish* para llevar a cabo la compresión.

3.46.2 Invalidando la caché

Nunca debería ser necesario invalidar los datos almacenados en caché porque la invalidación ya se tiene en cuenta de forma nativa en los modelos de caché *HTTP* (consulta *Invalidando la caché* (Página 223)).

Sin embargo, *Varnish* se puede configurar para aceptar un método *HTTP* especial `PURGE` que invalida la caché para un determinado recurso:

```
sub vcl_hit {
    if (req.request == "PURGE") {
        set obj.ttl = 0s;
        error 200 "Purged";
    }
}

sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Not purged";
    }
}
```

Prudencia: De alguna manera, debes proteger el método `PURGE` HTTP para evitar que alguien aleatoriamente purgue los datos memorizados.

3.47 Cómo usar plantillas *PHP* en lugar de *Twig*

No obstante que *Symfony2* de manera predeterminada usa *Twig* como su motor de plantillas, puedes usar código *PHP* simple si lo deseas. Ambos motores de plantilla son igualmente compatibles en *Symfony2*. *Symfony2* añade algunas características interesantes en lo alto de *PHP* para hacer más poderosa la escritura de plantillas con *PHP*.

3.47.1 Reproduciendo plantillas *PHP*

Si deseas utilizar el motor de plantillas *PHP*, primero, asegúrate de activarlo en el archivo de configuración de tu aplicación:

- *YAML*

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

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ... >
    <!-- ... -->
    <framework:templating ... >
        <framework:engine id="twig" />
        <framework:engine id="php" />
    </framework:templating>
</framework:config>
```

- *PHP*

```
    $contenedor->loadFromExtension('framework', array(
        // ...
        'templating'      => array(
            'engines' => array('twig', 'php'),
        ),
    ));
```

Ahora puedes reproducir una plantilla PHP en lugar de una *Twig* simplemente usando la extensión `.php` en el nombre de la plantilla en lugar de `.twig`. El controlador a continuación reproduce la plantilla `index.html.php`:

```
// src/Acme/HolaBundle/Controller/HolaController.php

public function indexAction($nombre)
{
    return $this->render('HolaBundle:Hola:index.html.php', array('nombre' => $nombre));
}
```

3.47.2 Decorando plantillas

Muy a menudo, las plantillas en un proyecto comparten elementos comunes, como los bien conocidos encabezados y pies de página. En *Symfony2*, nos gusta pensar en este problema de forma diferente: una plantilla se puede decorar con otra.

La plantilla `index.html.php` está decorada por `base.html.php`, gracias a la llamada a `extend()`:

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/index.html.php -->
<?php $view->extend('AcmeHolaBundle::base.html.php') ?>

Hola <?php echo $nombre ?>!
```

La notación `HolaBundle::base.html.php` te suena familiar, ¿no? Es la misma notación utilizada para referir una plantilla. La parte `::` simplemente significa que el elemento controlador está vacío, por lo tanto el archivo correspondiente se almacena directamente bajo `views/`.

Ahora, echemos un vistazo al archivo `base.html.php`:

```
<!-- src/Acme/HolaBundle/Resources/views/base.html.php -->
<?php $view->extend('::base.html.php') ?>

<h1>Aplicación ``Hola``</h1>

<?php $view['slots']->salida('_content') ?>
```

El diseño en sí mismo está decorado por otra (`::base.html.php`). *Symfony2* admite varios niveles de decoración: un diseño en sí se puede decorar con otro. Cuando la parte nombre del paquete de la plantilla está vacía, se buscan las vistas en el directorio `app/Resources/views/`. Este directorio almacena vistas globales de tu proyecto completo:

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->salida('titulo', 'Aplicación ``Hola``') ?></title>
    </head>
    <body>
        <?php $view['slots']->salida('_content') ?>
    </body>
</html>
```

Para ambos diseños, la expresión `$view['slots']->salida('_content')` se sustituye por el contenido de la plantilla hija, `index.html.php` y `base.html.php`, respectivamente (más de ranuras en la siguiente sección).

Como puedes ver, *Symfony2* proporciona métodos en un misterioso objeto `$view`. En una plantilla, la variable `$view` siempre está disponible y se refiere a un objeto especial que proporciona una serie de métodos que hacen funcionar el motor de plantillas.

3.47.3 Trabajar con ranuras

Una ranura es un fragmento de código, definido en una plantilla, y reutilizable en cualquier diseño para decorar una plantilla. En la plantilla `index.html.php`, define una ranura `titulo`:

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/index.html.php -->
<?php $view->extend('AcmeHolaBundle::base.html.php') ?>

<?php $view['slots']->set('titulo', 'Aplicación Hola Mundo') ?>

Hola <?php echo $nombre ?>!
```

El diseño base ya tiene el código para reproducir el título en el encabezado:

```
<!-- app/Resources/views/base.html.php -->
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php $view['slots']->salida('titulo', 'Aplicación ``Hola``') ?></title>
</head>
```

El método `output()` inserta el contenido de una ranura y, opcionalmente, toma un valor predeterminado si la ranura no está definida. Y `_content` sólo es una ranura especial que contiene la plantilla hija reproducida.

Para las ranuras grandes, también hay una sintaxis extendida:

```
<?php $view['slots']->start('titulo') ?>
    Una gran cantidad de HTML
<?php $view['slots']->stop() ?>
```

3.47.4 Incluyendo otras plantillas

La mejor manera de compartir un fragmento de código de plantilla es definir una plantilla que se pueda incluir en otras plantillas.

Crea una plantilla `hola.html.php`:

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/hola.html.php -->
Hola <?php echo $nombre ?>!
```

Y cambia la plantilla `index.html.php` para incluirla:

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/index.html.php -->
<?php $view->extend('AcmeHolaBundle::base.html.php') ?>

<?php echo $view->render('AcmeHola:Hola:hola.html.php', array('nombre' => $nombre)) ?>
```

El método `render()` evalúa y devuelve el contenido de otra plantilla (este exactamente es el mismo método que utilizamos en el controlador).

3.47.5 Integrando otros controladores

¿Y si deseas incrustar el resultado de otro controlador en una plantilla? Eso es muy útil cuando se trabaja con Ajax, o cuando la plantilla incrustada necesita alguna variable que no está disponible en la plantilla principal.

Si creas una acción `maravillosa`, y quieres incluirla en la plantilla `index.html.php`, basta con utilizar el siguiente código:

```
<!-- src/Acme/HolaBundle/Resources/views/Hola/index.html.php -->
<?php echo $view['actions']->render('HolaBundle:Hola:maravillosa', array('nombre' => $nombre, 'color'
```

Aquí, la cadena `HolaBundle:Hola:maravillosa` se refiere a la acción `maravillosa` del controlador `Hola`:

```
// src/Acme/HolaBundle/Controller/HolaController.php

class HolaController extends Controller
{
    public function maravillosaAction($nombre, $color)
    {
        // crea algún objeto, basado en la variable $color
        $objeto = ...;

        return $this->render('HolaBundle:Hola:maravillosa.html.php', array('nombre' => $nombre, 'obje
    }

    // ...
}
```

Pero ¿dónde se define el elemento arreglo `$view['actions']` de la vista? Al igual que ``$view['slots']`, este invoca a un ayudante de plantilla, y la siguiente sección contiene más información sobre ellos.

3.47.6 Usando ayudantes de plantilla

El sistema de plantillas de *Symfony2* se puede extender fácilmente por medio de los ayudantes. Los ayudantes son objetos PHP que ofrecen funciones útiles en el contexto de la plantilla. `actions` y `slots` son dos de los ayudantes integrados en *Symfony2*.

Creando enlaces entre páginas

Hablando de aplicaciones web, crear enlaces entre páginas es una necesidad. En lugar de codificar las direcciones *URL* en las plantillas, el ayudante `router` sabe cómo generar direcciones *URL* basándose en la configuración de enrutado. De esta manera, todas tus direcciones *URL* se pueden actualizar fácilmente cambiando la configuración:

```
<a href="<?php echo $view['router']->generate('hola', array('nombre' => 'Thomas')) ?>">
    Greet Thomas!
</a>
```

El método `generate()` toma el nombre de la ruta y un arreglo de parámetros como argumentos. El nombre del route es la clave principal en la cual son referidas las rutas y los parámetros son los valores de los marcadores de posición definidos en el patrón route:

```
# src/Acme/HolaBundle/Resources/config/routing.yml
hola: # El nombre de ruta
    pattern:  /hola/{nombre}
    defaults: { _controller: AcmeHolaBundle:Hola:index }
```

Usando activos: imágenes, archivos *JavaScript* y hojas de estilo

¿Qué sería de Internet sin imágenes, JavaScript y hojas de estilo? *Symfony2* proporciona la etiqueta `assets` para hacer frente a los activos fácilmente:

```
<link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/css" /

<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" />
```

El principal objetivo del ayudante `assets` es hacer más portátil tu aplicación. Gracias a este ayudante, puedes mover el directorio raíz de tu aplicación a cualquier lugar bajo tu directorio raíz del servidor web sin cambiar nada en el código de tu plantilla.

3.47.7 Mecanismo de escape

Al utilizar plantillas *PHP*, escapa las variables cada vez que se muestren al usuario:

```
<?php echo $view->escape($var) ?>
```

De forma predeterminada, el método `escape()` supone que la variable se emite dentro de un contexto *HTML*. El segundo argumento te permite cambiar el contexto. Por ejemplo, para mostrar algo en un archivo de *JavaScript*, utiliza el contexto `js`:

```
<?php echo $view->escape($var, 'js') ?>
```

3.48 Cómo cargar clases automáticamente

Siempre que uses una clase no definida, *PHP* utiliza el mecanismo de carga automática para delegar la carga de un archivo de definición de clase. *Symfony2* proporciona un cargador automático "universal", que es capaz de cargar clases desde los archivos que implementan uno de los siguientes convenios:

- Los estándares de interoperabilidad técnica para los espacios de nombres y nombres de clases de *PHP 5.3*;
- La convención de nomenclatura de clases de PEAR.

Si tus clases y las bibliotecas de terceros que utilizas en tu proyecto siguen estas normas, el cargador automático de *Symfony2* es el único cargador automático que necesitarás siempre.

3.48.1 Uso

Registrar la clase del cargador automático `Symfony\Component\ClassLoader\UniversalClassLoader` es sencillo:

```
require_once '/ruta/a/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

use Symfony\Component\ClassLoader\UniversalClassLoader;

$cargador = new UniversalClassLoader();
$cargador->register();
```

Para una menor ganancia en rendimiento puedes memorizar en caché las rutas de las clases usando APC, con sólo registrar la clase `Symfony\Component\ClassLoader\ApcUniversalClassLoader`:

```
require_once '/ruta/a/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once '/ruta/a/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$cargador = new ApcUniversalClassLoader('apc.prefix.');
$cargador->register();
```

El cargador automático es útil sólo si agregas algunas bibliotecas al cargador automático.

Nota: El autocargador se registra automáticamente en una aplicación *Symfony2* (consulta el `app/autoload.php`).

Si las clases a cargar automáticamente utilizan espacios de nombres, utiliza cualquiera de los métodos **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNamespace'** o **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNamespaces'**:

```
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/src');

$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/src',
));
```

Para las clases que siguen la convención de nomenclatura de PEAR, utiliza cualquiera de los métodos **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerPrefix'** o **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerPrefixes'**:

```
$cargador->registerPrefix('Twig_', __DIR__.'/vendor/twig/lib');

$cargador->registerPrefixes(array(
    'Swift_' => __DIR__.'/vendor/swiftmailer/lib/classes',
    'Twig_'  => __DIR__.'/vendor/twig/lib',
));
```

Nota: Algunas bibliotecas también requieren que su ruta de acceso raíz esté registrada en la ruta de include *PHP* (`set_include_path()`).

Las clases de un subespacio de nombres o una subjerarquía de clases *PEAR* se puede buscar en una lista de ubicaciones para facilitar el "vendoring" de un subconjunto de clases para los grandes proyectos:

```
$cargador->registerNamespaces(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine-common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine-migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine-dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/lib',
));
```

En este ejemplo, si intentas utilizar una clase en el espacio de nombres `Doctrine\Common` o uno de sus hijos, el cargador automático buscará primero la clase en el directorio `doctrine\common`, y entonces vuelve a la reserva al directorio predeterminado `doctrine` (el último configurado) si no se encuentra, antes de darse por vencido. El orden de registro es importante en este caso.

3.49 Cómo localizar archivos

El componente **:namespace:‘Symfony\\Component\\Finder‘** te ayuda a buscar archivos y directorios rápida y fácilmente.

3.49.1 Uso

La clase `Symfony\Component\Finder\Finder` busca archivos y/o directorios:

```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    print $file->getRealpath()."\n";
}
```

`$file` es una instancia de **:phpclass:‘SplFileInfo‘**.

El código anterior imprime recursivamente los nombres de todos los archivos en el directorio actual. La clase Finder utiliza una interfaz fluida, por lo que todos los métodos devuelven la instancia del Finder.

Truco: Una instancia Finder es un iterador PHP. Por tanto, en lugar de iterar sobre el Finder con `foreach`, también lo puedes convertir en una matriz con el método **:phpfunction:‘iterator_to_array‘** u obtener el número de elementos con **:phpfunction:‘iterator_count‘**.

3.49.2 Criterios

Ubicación

La ubicación es el único criterio obligatorio. Este dice al buscador cual directorio utilizar para la búsqueda:

```
$finder->in(__DIR__);
```

Busca en varios lugares encadenando llamadas al método **:method:‘Symfony\\Component\\Finder\\Finder::in‘**:

```
$finder->files()->in(__DIR__)->in('/elsewhere');
```

Excluye directorios coincidentes con el método **:method:‘Symfony\\Component\\Finder\\Finder::exclude‘**:

```
$finder->in(__DIR__)->exclude('ruby');
```

Debido a que `Finder` utiliza iteradores *PHP*, puedes pasar cualquier *URL* compatible con protocolo:

```
$finder->in('ftp://ejemplo.com/pub/');
```

Y también trabaja con flujos definidos por el usuario:

```
use Symfony\Component\Finder\Finder;

$s3 = new \Zend_Service_Amazon_S3($key, $secret);
$s3->registerStreamWrapper("s3");

$finder = new Finder();
```

```
$finder->name('photos*')->size('< 100K')->date('since 1 hour ago');
foreach ($finder->in('s3://bucket-name') as $file) {
    // hace algo

    print $file->getFilename()."\n";
}
```

Nota: Lee la documentación de Flujos para aprender a crear tus propios flujos.

Archivos o directorios

Por omisión, Finder devuelve archivos y directorios, pero lo controlan los métodos **:method:'Symfony\\Component\\Finder\\Finder::files'** y **:method:'Symfony\\Component\\Finder\\Finder::directories'**:

```
$finder->files();

$finder->directories();
```

Si quieres seguir los enlaces, utiliza el método `followLinks()`:

```
$finder->files()->followLinks();
```

De forma predeterminada, el iterador ignora archivos VCS populares. Esto se puede cambiar con el método `ignoreVCS()`:

```
$finder->ignoreVCS(false);
```

Ordenación

Ordena el resultado por nombre o por tipo (primero directorios, luego archivos):

```
$finder->sortByName();

$finder->sortByType();
```

Nota: Ten en cuenta que los métodos `sort*` necesitan obtener todos los elementos para hacer su trabajo. Para iteradores grandes, es lento.

También puedes definir tu propio algoritmo de ordenación con el método `sort()`:

```
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

Nombre de archivo

Filtra archivos por nombre con el método **:method:'Symfony\\Component\\Finder\\Finder::name'**:

```
$finder->files()->name('*.php');
```

El método `name()` acepta globos, cadenas o expresiones regulares:

```
$finder->files()->name('/\.php$/');
```

El método `notNames()` excluye archivos coincidentes con un patrón:

```
$finder->files()->notName('*.rb');
```

Tamaño de archivo

Filtra archivos por tamaño con el método **:method:'Symfony\\Component\\Finder\\Finder::size'**:

```
$finder->files()->size('< 1.5K');
```

Filtra por rangos de tamaño encadenando llamadas a:

```
$finder->files()->size('>= 1K')->size('<= 2K');
```

El operador de comparación puede ser cualquiera de los siguientes: >, >=, <, <=, ==.

El valor destino puede utilizar magnitudes de kilobytes (`k`, `ki`), megabytes (`m`, `mi`), o gigabytes (`g`, `gi`). Los sufijos con una `i` usan la versión `2**n` adecuada de acuerdo al estándar IEC.

Fecha de archivo

Filtra archivos por fecha de última modificación con el método **:method:'Symfony\\Component\\Finder\\Finder::date'**:

```
$finder->date('since yesterday');
```

El operador de comparación puede ser cualquiera de los siguientes: >, >=, <, <=, ==. También puedes utilizar `since` o `after` como alias para >, y `until` o `before` como alias para <.

El valor destino puede ser cualquier fecha compatible con la función strtotime.

Profundidad de directorio

De manera predeterminada, Finder recorre directorios recursivamente. Filtra la profundidad del recorrido con el método **:method:'Symfony\\Component\\Finder\\Finder::depth'**:

```
$finder->depth('== 0');
$finder->depth('< 3');
```

Filtrado personalizado

Para restringir que el archivo coincida con su propia estrategia, utiliza el método **:method:'Symfony\\Component\\Finder\\Finder::filter'**:

```
$filtro = function (\SplFileInfo $file)
{
    if (strlen($file) > 10) {
        return false;
    }
```

```
};

$finder->files()->filter($filtro);
```

El método `filter()` toma un cierre como argumento. Para cada archivo coincidente, este es llamado con el archivo como una instancia de **:phpclass:'SplFileInfo'**. El archivo se excluye desde el conjunto de resultados si el cierre devuelve `false`.

3.50 Cómo crear la consola/línea de ordenes

Symfony2 viene con un componente Consola, que te permite crear ordenes para la línea de ordenes. Tus ordenes de consola se pueden utilizar para cualquier tarea repetitiva, como cronjobs, importaciones, u otros trabajos por lotes.

3.50.1 Creando una orden básica

Para hacer que las ordenes de la consola estén disponibles automáticamente con *Symfony2*, crea un directorio `Command` dentro de tu paquete y crea un archivo *PHP* con el sufijo `Command.php` para cada orden que deses proveer. Por ejemplo, si deseas ampliar el `AcmeDemoBundle` (disponible en la edición estándar de *Symfony*) para darnos la bienvenida desde la línea de ordenes, crea `GreetCommand.php` y añade lo siguiente al mismo:

```
// src/Acme/DemoBundle/Command/GreetCommand.php
namespace Acme\DemoBundle\Command;

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class GreetCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setNombre('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument('nombre', InputArgument::OPTIONAL, 'Who do you want to greet?')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'If set, the task will yell in upperca
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $salida)
    {
        $nombre = $input->getArgument('nombre');
        if ($nombre) {
            $text = 'Hola '.$nombre;
        } else {
            $text = 'Hola';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }
```

```
        $salida->writeln($text);
    }
}
```

Prueba la nueva consola de ordenes ejecutando lo siguiente

```
app/console demo:greet Fabien
```

Esto imprimirá lo siguiente en la línea de ordenes:

```
Hola Fabien
```

También puedes utilizar a `--yell` para convertir todo a mayúsculas:

```
app/console demo:greet Fabien --yell
```

Esto imprime:

```
HOLA FABIEN
```

Coloreando la salida

Cada vez que produces texto, puedes escribir el texto con etiquetas para colorear tu salida. Por ejemplo:

```
// texto verde
$salida->writeln('<info>foo</info>');

// texto amarillo
$salida->writeln('<comment>foo</comment>');

// texto negro sobre fondo cian
$salida->writeln('<question>foo</question>');

// texto blanco sobre fondo rojo
$salida->writeln('<error>foo</error>');
```

3.50.2 Utilizando argumentos de ordenes

La parte más interesante de las ordenes son los argumentos y opciones que puedes hacer disponibles. Los argumentos son cadenas - separadas por espacios - que vienen después del nombre de la orden misma. Ellos están ordenados, y pueden ser opcionales u obligatorios. Por ejemplo, añade un argumento `last_nombre` opcional a la orden y haz que el argumento `nombre` sea obligatorio:

```
$this
    // ...
    ->addArgument('nombre', InputArgument::REQUIRED, 'Who do you want to greet?')
    ->addArgument('apellido', InputArgument::OPTIONAL, 'Your last name?')
    // ...
```

Ahora tienes acceso a un argumento `apellido` en la orden:

```
if ($apellido = $input->getArgument('apellido')) {
    $text .= ' '.$apellido;
}
```

Ahora la orden se puede utilizar en cualquiera de las siguientes maneras:

```
app/console demo:greet Fabien
app/console demo:greet Fabien Potencier
```

3.50.3 Usando las opciones de la orden

A diferencia de los argumentos, las opciones no están ordenadas (lo cual significa que las puedes especificar en cualquier orden) y se especifican con dos guiones (por ejemplo, `--yell` también puedes declarar un atajo de una letra que puedes invocar con un único guión como `-y`). Las opciones son: *always* opcional, y se puede configurar para aceptar un valor (por ejemplo, `dir=src`) o simplemente como una variable lógica sin valor (por ejemplo, `yell`).

Truco: También es posible hacer que una opción *opcionalmente* acepte un valor (de modo que `--yell` o `yell=loud` funcione). Las opciones también se pueden configurar para aceptar una matriz de valores.

Por ejemplo, añadir una nueva opción a la orden que se puede usar para especificar cuántas veces se debe imprimir el mensaje en una fila:

```
$this
    // ...
    ->addOption('iterations', null, InputOption::VALUE_REQUIRED, 'How many times should the message b
```

A continuación, utilízalo en la orden para imprimir el mensaje varias veces:

```
for ($i = 0; $i < $input->getOption('iterations'); $i++) {
    $salida->writeln($text);
}
```

Ahora, al ejecutar la tarea, si lo deseas, puedes especificar un indicador `--iterations`:

```
app/console demo:greet Fabien

app/console demo:greet Fabien --iterations=5
```

El primer ejemplo sólo se imprimirá una vez, ya que `iterations` está vacía y el predeterminado es un `1` (el último argumento de `addOption`). El segundo ejemplo se imprimirá cinco veces.

Recordemos que a las opciones no les preocupa su orden. Por lo tanto, cualquiera de las siguientes trabajará:

```
app/console demo:greet Fabien --iterations=5 --yell
app/console demo:greet Fabien --yell --iterations=5
```

3.50.4 Pidiendo información al usuario

Al crear ordenes, tienes la capacidad de recopilar más información de los usuarios haciéndoles preguntas. Por ejemplo, supongamos que deseas confirmar una acción antes de llevarla a cabo realmente. Añade lo siguiente a tu orden:

```
$dialog = $this->getHelperSet()->get('dialog');
if (!$dialog->askConfirmation($salida, '<question>Continue with this action?</question>', false)) {
    return;
}
```

En este caso, el usuario tendrá que "Continuar con esta acción", y, a menos que responda con `y`, la tarea se detendrá. El tercer argumento de `askConfirmation` es el valor predeterminado que se devuelve si el usuario no introduce algo.

También puedes hacer preguntas con más que una simple respuesta sí/no. Por ejemplo, si necesitas saber el nombre de algo, puedes hacer lo siguiente:

```
$dialog = $this->getHelperSet()->get('dialog');
$nombre = $dialog->ask($salida, 'Por favor ingresa el nombre del elemento gráfico', 'foo');
```

3.50.5 Probando ordenes

Symfony2 proporciona varias herramientas para ayudarte a probar las ordenes. La más útil es la clase `Symfony\Component\Console\Tester\CommandTester`. Esta utiliza clases entrada y salida especiales para facilitar la prueba sin una consola real:

```
use Symfony\Component\Console\Tester\CommandTester;
use Symfony\Bundle\FrameworkBundle\Console\Application;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        // simula el Kernel o crea uno dependiendo de tus necesidades
        $application = new Application($kernel);

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getNombreCompleto()));

        $this->assertRegExp('/.../', $commandTester->getDisplay());

        // ...
    }
}
```

El método **:method:'Symfony\\Component\\Console\\Tester\\CommandTester::getDisplay'** devuelve lo que se ha exhibido durante una llamada normal de la consola.

Truco: También puedes probar toda una aplicación de consola utilizando `Symfony\Component\Console\Tester\ApplicationTester`.

3.50.6 Obteniendo servicios del contenedor de servicios

Al usar `SymfonyBundleFrameworkBundleCommandContainerAwareCommand` como la clase base para la orden (en lugar del más básico `SymfonyComponentConsoleCommandCommand`), tienes acceso al contenedor de servicios. En otras palabras, tienes acceso a cualquier servicio configurado. Por ejemplo, fácilmente podrías extender la tarea para que sea traducible:

```
protected function execute(InputInterface $input, OutputInterface $salida)
{
    $nombre = $input->getArgument('nombre');
    $translator = $this->getContainer()->get('translator');
    if ($nombre) {
        $salida->writeln($translator->trans('Hello %name%!', array('%name%' => $nombre)));
    } else {
        $salida->writeln($translator->trans('Hello!'));
    }
}
```

3.50.7 Llamando una orden existente

Si una orden depende de que se ejecute otra antes, en lugar de obligar al usuario a recordar el orden de ejecución, puedes llamarla directamente tú mismo. Esto también es útil si deseas crear una "metaorden" que ejecute un montón de otras ordenes (por ejemplo, todas las ordenes que se deben ejecutar cuando el código del proyecto ha cambiado en los servidores de producción: vaciar la caché, generar sustitutos Doctrine2, volcar activos `Assetic`, ...).

Llamar a una orden desde otra es sencillo:

```
protected function execute(InputInterface $input, OutputInterface $salida)
{
    $command = $this->getApplication()->find('demo:greet');

    $arguments = array(
        'nombre' => 'Fabien',
        '--yell' => true,
    );

    $input = new ArrayInput($arguments);
    $returnCode = $command->run($input, $salida);

    // ...
}
```

En primer lugar, **:method:'Symfony\\Component\\Console\\Command\\Command::find'** busca la orden que deseas ejecutar pasando el nombre de la orden.

Entonces, es necesario crear una nueva clase `Symfony\Component\Console\Input\ArrayInput` con los argumentos y opciones que desees pasar a la orden.

Eventualmente, llamar al método `run()` en realidad ejecuta la orden y regresa el código devuelto por la orden (`0` si todo va bien, cualquier otro número entero de otra manera).

Nota: La mayor parte del tiempo, llamar a una orden desde código que no se ejecuta en la línea de ordenes no es una buena idea por varias razones. En primer lugar, la salida de la orden se ha optimizado para la consola. Pero lo más importante, puedes pensar de una orden como si fuera un controlador; este debe utilizar el modelo para hacer algo y mostrar algún comentario al usuario. Así, en lugar de llamar una orden de la Web, reconstruye el código y mueve la lógica a una nueva clase.

3.51 Cómo optimizar tu entorno de desarrollo para depuración

Cuando trabajas en un proyecto de *Symfony* en tu equipo local, debes usar el entorno `dev` (con el controlador frontal `app_dev.php`). Esta configuración del entorno se ha optimizado para dos propósitos principales:

- Proporcionar retroalimentación de desarrollo precisa cada vez que algo sale mal (barra de herramientas de depuración web, páginas de excepción agradables, perfiles, ...);
- Ser lo más parecido posible al entorno de producción para evitar problemas al implantar el proyecto.

3.51.1 Desactivando el archivo de arranque y la caché de clase

Y para que el entorno de producción sea lo más rápido posible, *Symfony* crea grandes archivos *PHP* en la memoria caché que contienen la agregación de las clases *PHP* que tu proyecto necesita para cada petición. Sin embargo, este

comportamiento puede confundir a tu IDE o depurador. Esta fórmula muestra cómo puedes ajustar este mecanismo de memorización para que sea más amigable cuando necesitas depurar código que incluye clases de *Symfony*.

El controlador frontal `app_dev.php` por omisión lee lo siguiente:

```
// ...

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Para contentar a tu depurador, desactiva toda la caché de las clases *PHP* eliminando la llamada a `loadClassCache()` y sustituyendo las declaraciones *require* como la siguiente:

```
// ...

// require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once __DIR__.'/../app/autoload.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
// $kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Truco: Si desactivas la caché *PHP*, no olvides reactivarla después de tu sesión de depuración.

A algunos IDE no les gusta el hecho de que algunas clases se almacenen en lugares diferentes. Para evitar problemas, puedes decirle a tu *IDE* que omita los archivos de cache *PHP*, o puedes cambiar la extensión utilizada por *Symfony* para estos archivos:

```
$kernel->loadClassCache('classes', '.php.cache');
```

3.52 Cómo utilizar `Monolog` para escribir Registros

Monolog es una biblioteca de registro para *PHP* 5.3 utilizada por *Symfony2*. Inspirada en la biblioteca LogBook de Python.

3.52.1 Uso

En `Monolog` cada registrador define un canal de registro. Cada canal tiene una pila de controladores para escribir los registros (los controladores se pueden compartir).

Truco: Cuando inyectas el registrador en un servicio puedes *utilizar un canal personalizado* (Página 563) para ver fácilmente qué parte de la aplicación registró el mensaje.

El encargado básico es el `StreamHandler` el cual escribe los registros en un flujo (por omisión en `app/logs/prod.log` en el entorno de producción y `app/logs/dev.log` en el entorno de desarrollo).

Monólogo también viene con una potente capacidad integrada para encargarse del registro en el entorno de producción: `FingersCrossedHandler`. Esta te permite almacenar los mensajes en la memoria intermedia y registrarla sólo si un mensaje llega al nivel de acción (ERROR en la configuración prevista en la edición estándar) transmitiendo los mensajes a otro controlador.

Para registrar un mensaje, simplemente obtén el servicio registrador del contenedor en tu controlador:

```
$registrador = $this->get('logger');
$registrador->info('acabamos de recibir el registrador');
$registrador->err('Ha ocurrido un error');
```

Truco: Usar sólo los métodos de la interfaz `SymfonyComponentHttpKernelLogLoggerInterface` te permite cambiar la implementación del anotador sin cambiar el código.

Utilizando varios controladores

El registrador utiliza una pila de controladores que se llaman sucesivamente. Esto te permite registrar los mensajes de varias formas fácilmente.

- *YAML*

```
monolog:
    handlers:
        syslog:
            type: stream
            path: /var/log/symfony.log
            level: error
        main:
            type: fingerscrossed
            action_level: warning
            handler: file
        file:
            type: stream
            level: debug
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="syslog"
            type="stream"
            path="/var/log/symfony.log"
            level="error"
        />
        <monolog:handler
            name="main"
            type="fingerscrossed"
            action-level="warning"
```

```
            handler="file"
        />
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
        />
    </monolog:config>
</container>
```

La configuración anterior define una pila de controladores que se llamarán en el orden en el cual se definen.

Truco: El controlador denominado "file" no se incluirá en la misma pila que se utiliza como un controlador anidado del controlador *fingerscrossed*.

Nota: Si deseas cambiar la configuración de *MonologBundle* en otro archivo de configuración necesitas redefinir toda la pila. No se pueden combinar, ya que el orden es importante y una combinación no te permite controlar el orden.

Cambiando el formateador

El controlador utiliza un `formateador` para dar formato al registro antes de ingresarlo. Todos los manipuladores `Monolog` utilizan una instancia de `Monolog\Formatter\LineFormatter` por omisión, pero la puedes reemplazar fácilmente. Tu formateador debe implementar `Monolog\Formatter\FormatterInterface`.

- *YAML*

```
services:
    my_formatter:
        class: Monolog\Formatter\JsonFormatter
monolog:
    handlers:
        file:
            type: stream
            level: debug
            formatter: my_formatter
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <services>
        <service id="my_formatter" class="Monolog\Formatter\JsonFormatter" />
    </services>
    <monolog:config>
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
            formatter="my_formatter"
        />
```

```
        </monolog:config>
    </container>
```

3.52.2 Agregando algunos datos adicionales en los mensajes de registro

Monolog te permite procesar el registro antes de ingresarlo añadiendo algunos datos adicionales. Un procesador se puede aplicar al manipulador de toda la pila o sólo para un manipulador específico.

Un procesador simplemente es un ejecutable que recibe el registro como primer argumento.

Los procesadores se configuran usando la etiqueta DIC `monolog.processor`. Consulta la *referencia sobre esto* (Página 563).

Agregando un segmento Sesión/Petición

A veces es difícil saber cuál de las entradas en el registro pertenece a cada sesión y/o petición. En el siguiente ejemplo agregaremos una ficha única para cada petición usando un procesador.

```
namespace Acme\MiBundle;

use Symfony\Component\HttpFoundation\Session;

class SessionRequestProcessor
{
    private $sesion;
    private $muestra;

    public function __construct(Session $sesion)
    {
        $this->sesion = $sesion;
    }

    public function processRecord(array $record)
    {
        if (null === $this->muestra) {
            try {
                $this->muestra = substr($this->session->getId(), 0, 8);
            } catch (\RuntimeException $e) {
                $this->muestra = '????????';
            }
            $this->muestra .= '-' . substr(uniqid(), -8);
        }
        $record['extra']['token'] = $this->muestra;

        return $record;
    }
}
```

- *YAML*

```
services:
    monolog.formatter.session_request:
        class: Monolog\Formatter\LineFormatter
        arguments:
            - "[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%\n"

    monolog.processor.session_request:
```

```
            class: Acme\MiBundle\SessionRequestProcessor
            arguments:  [ @session ]
            tags:
                - { name: monolog.processor, method: processRecord }

    monolog:
        handlers:
            main:
                type: stream
                path: %kernel.logs_dir%/%kernel.environment%.log
                level: debug
                formatter: monolog.formatter.session_request
```

Nota: Si utilizas varios manipuladores, también puedes registrar el procesador a nivel del manipulador en lugar de globalmente.

3.53 Cómo extender una clase sin necesidad de utilizar herencia

Para permitir que varias clases agreguen métodos a otra, puedes definir el método mágico `__call()` de la clase que deseas ampliar de esta manera:

```
class Foo
{
    // ...

    public function __call($method, $arguments)
    {
        // crea un evento llamado 'foo.method_is_not_found'
        $evento = new HandleUndefinedMethodEvent($this, $method, $arguments);
        $this->despachador->dispatch($this, 'foo.method_is_not_found', $evento);

        // ¿ningún escucha es capaz de procesar el evento? El método no existe
        if (!$evento->isProcessed()) {
            throw new \Exception(sprintf('Call to undefined method %s::%s.', get_class($this), $metho
        }

        // devuelve el escucha del valor de retorno
        return $evento->getReturnValue();
    }
}
```

Este utiliza un `HandleUndefinedMethodEvent` especial que también se debe crear. Esta es una clase genérica que se podría reutilizar cada vez que necesites usar este modelo de extensión de la clase:

```
use Symfony\Component\EventDispatcher\Event;

class HandleUndefinedMethodEvent extends Event
{
    protected $subject;
    protected $method;
    protected $arguments;
    protected $returnValue;
    protected $isProcessed = false;

    public function __construct($subject, $method, $arguments)
```

```
    {
        $this->subject = $subject;
        $this->method = $method;
        $this->arguments = $arguments;
    }

    public function getSubject()
    {
        return $this->subject;
    }

    public function getMethod()
    {
        return $this->method;
    }

    public function getArguments()
    {
        return $this->arguments;
    }

    /**
     * Fija el valor a devolver y detiene la notificación a otros escuchas
     */
    public function setReturnValue($val)
    {
        $this->returnValue = $val;
        $this->isProcessed = true;
        $this->stopPropagation();
    }

    public function getReturnValue($val)
    {
        return $this->returnValue;
    }

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

A continuación, crea una clase que debe escuchar el evento `foo.method_is_not_found` y *añade* el método `bar()`:

```
class Bar
{
    public function onFooMethodIsNotFound(HandleUndefinedMethodEvent $evento)
    {
        // únicamente deseamos responder a las llamadas al método 'bar'
        if ('bar' != $evento->getMethod()) {
            // permite que otro escucha se preocupe del método desconocido devuelto
            return;
        }

        // el objeto subject (la instancia foo)
        $foo = $evento->getSubject();
```

```
        // los argumentos del método bar
        $arguments = $evento->getArguments();

        // Hacer alguna cosa
        // ...

        // fija el valor de retorno
        $evento->setReturnValue($someValue);
    }
}
```

Por último, agrega el nuevo método `bar` a la clase `Foo` registrando una instancia de `Bar` con el evento `foo.method_is_not_found`:

```
$bar = new Bar();
$despachador->addListener('foo.method_is_not_found', $bar);
```

3.54 Cómo personalizar el comportamiento de un método sin utilizar herencia

3.54.1 Haciendo algo antes o después de llamar a un método

Si quieres hacer algo justo antes o justo después de invocar a un método, puedes enviar un evento, al principio o al final del método, respectivamente:

```
class Foo
{
    // ...

    public function send($foo, $bar)
    {
        // hace algo antes del método
        $evento = new FilterBeforeSendEvent($foo, $bar);
        $this->despachador->dispatch('foo.pre_send', $evento);

        // obtiene $foo y $bar desde el evento, esto se puede modificar
        $foo = $evento->getFoo();
        $bar = $evento->getBar();
        // aquí va la implementación real
        // $ret = ...;

        // hace algo después del método
        $evento = new FilterSendReturnValue($ret);
        $this->despachador->dispatch('foo.post_send', $evento);

        return $evento->getReturnValue();
    }
}
```

En este ejemplo, se lanzan dos eventos: `foo.pre_send`, antes de ejecutar el método, y `foo.post_send` después de ejecutar el método. Cada uno utiliza una clase Evento personalizada para comunicar información a los escuchas de los dos eventos. Estas clases de evento se tendrían que crear por ti y deben permitir que, en este ejemplo, las variables `$foo`, `$bar` y `$ret` sean recuperadas y establecidas por los escuchas.

Por ejemplo, suponiendo que el `FilterSendReturnValue` tiene un método `setReturnValue`, un escucha puede tener este aspecto:

```
public function onFooPostSend(FilterSendReturnValue $evento)
{
    $ret = $evento->getReturnValue();
    // modifica el valor original ``$ret``

    $evento->setReturnValue($ret);
}
```

3.55 Cómo registrar un nuevo formato de petición y tipo MIME

Cada `Petición` tiene un "formato" (por ejemplo, `html`, `json`), el cual se utiliza para determinar qué tipo de contenido regresar en la `Respuesta`. De hecho, el formato de petición, accesible a través del método **:method:'Symfony\\Component\\HttpFoundation\\Request::getRequestFormat'**, se utiliza para establecer el tipo MIME de la cabecera `Content-Type` en el objeto `Respuesta`. Internamente, *Symfony* incluye un mapa de los formatos más comunes (por ejemplo, `html`, `json`) y sus correspondientes tipos MIME (por ejemplo, `text/html`, `application/json`). Por supuesto, se pueden agregar entradas de formato tipo MIME adicionales fácilmente. Este documento te mostrará cómo puedes agregar el formato `jsonp` y el tipo MIME correspondiente.

3.55.1 Crea un escucha `kernel.request`

La clave para definir un nuevo tipo MIME es crear una clase que debe "escuchar" el evento `kernel.request` enviado por el núcleo de *Symfony*. El evento `kernel.request` se difunde temprano en *Symfony*, el manipulador de la petición lo procesa y te permite modificar el objeto `Petición`.

Crea la siguiente clase, sustituyendo la ruta con una ruta a un paquete en tu proyecto:

```
// src/Acme/DemoBundle/RequestListener.php
namespace Acme\DemoBundle;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $evento)
    {
        $evento->getRequest()->setFormat('jsonp', 'application/javascript');
    }
}
```

3.55.2 Registrando tu escucha

Como para cualquier otro escucha, tienes que añadirlo en uno de tus archivos de configuración y registrarlo como un escucha añadiendo la etiqueta `kernel.event_listener`:

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

        <service id="acme.demobundle.listener.request" class="Acme\DemoBundle\RequestListener">
            <tag name="kernel.event_listener" event="kernel.request" method="onKernelRequest" />
        </service>

    </container>
```

- *YAML*

```
# app/config/config.yml
services:
    acme.demobundle.listener.request:
        class: Acme\DemoBundle\RequestListener
        tags:
            - { name: kernel.event_listener, event: kernel.request, method: onKernelRequest }
```

- *PHP*

```
# app/config/config.php
$definicion = new Definition('Acme\DemoBundle\RequestListener');
$definicion->addTag('kernel.event_listener', array('event' => 'kernel.request', 'method' => 'onK
$contenedor->setDefinition('acme.demobundle.listener.request', $definicion);
```

En este punto, el servicio `acme.demobundle.listener.request` se ha configurado y será notificado cuando el núcleo de *Symfony* difunda el evento `kernel.request`.

Truco: También puedes registrar el escucha en una clase extensión de configuración (consulta *Importando configuración vía extensiones del contenedor* (Página 242) para más información).

3.56 Cómo crear un colector de datos personalizado

El `Generador de perfiles` de *Symfony2* delega la recolección de datos a los colectores de datos. *Symfony2* incluye algunos colectores, pero puedes crear tu propio diseño fácilmente.

3.56.1 Creando un colector de datos personalizado

Crear un colector de datos personalizado es tan simple como implementar la clase `Symfony\Component\HttpKernel\DataCollector\DataCollectorInterface`:

```
interface DataCollectorInterface
{
    /**
     * Recoge los datos de las instancias de Request y Response.
     *
     * @param Request    $peticion   Una instancia de Request
     * @param Response   $respuesta  Una instancia de Response
     * @param \Exception $exception Una instancia de Exception
     */
    function collect(Request $peticion, Response $respuesta, \Exception $exception = null);

    /**
     * Devuelve el nombre del colector.
```

```
     *
     * @return string El nombre del colector
     */
    function getNombre();
}
```

El método `getNombre()` debe devolver un nombre único. Este se utiliza para acceder a la información más adelante en (consulta *Cómo utilizar el generador de perfiles en una prueba funcional* (Página 366) para ver un ejemplo).

El método `collect()` se encarga de almacenar los datos de las propiedades locales a las que quieres dar acceso.

Prudencia: Puesto que el generador de perfiles serializa instancias del colector de datos, no debes almacenar objetos que no se puedan serializar (como objetos PDO), o tendrás que proporcionar tu propio método `serialize()`.

La mayoría de las veces, es conveniente extender `Symfony\Component\HttpKernel\DataCollector\DataCollector` y rellenar los datos de la propiedad `$this->datos` (que se encarga de serializar la propiedad `$this->datos`):

```
class MemoryDataCollector extends DataCollector
{
    public function collect(Request $peticion, Response $respuesta, \Exception $exception = null)
    {
        $this->datos = array(
            'memory' => memory_get_peak_usage(true),
        );
    }

    public function getMemory()
    {
        return $this->datos['memory'];
    }

    public function getNombre()
    {
        return 'memory';
    }
}
```

3.56.2 Habilitando colectores de datos personalizados

Para habilitar un colector de datos, lo tienes que añadir como un servicio regular a tu configuración, y etiquetarlo con `data_collector`:

- *YAML*

```
services:
    data_collector.el_nombre_de_tu_colector:
        class: Fully\Qualified\Collector\Class\Name
        tags:
            - { name: data_collector }
```

- *XML*

```
<service id="data_collector.el_nombre_de_tu_colector" class="Fully\Qualified\Collector\Class\Nam
    <tag name="data_collector" />
</service>
```

- *PHP*

```
    $contenedor
        ->register('data_collector.el_nombre_de_tu_colector', 'Fully\Qualified\Collector\Class\Name'
        ->addTag('data_collector')
    ;
```

3.56.3 Añadiendo el generador de perfiles web en plantillas

Cuando desees mostrar los datos recogidos por el colector de datos en la barra de depuración o el generador de perfiles web, crea una plantilla *Twig* siguiendo este esqueleto:

```
{% extends 'WebProfilerBundle:Profiler:base.html.twig' %}

{% block toolbar %}
    {# contenido de la barra de herramientas de depuración web #}
{% endblock %}

{% block head %}
    {# si el panel del generador de perfiles necesita algunos archivos JS o CSS específicos #}
{% endblock %}

{% block menu %}
    {# el contenido del menú #}
{% endblock %}

{% block panel %}
    {# el contenido del panel #}
{% endblock %}
```

Cada bloque es opcional. El bloque `toolbar` se utiliza para la barra de herramientas de depuración web `menu` y `panel` se utilizan para agregar un grupo especial al generador de perfiles web.

Todos los bloques tienen acceso al objeto `collector`.

Truco: Las plantillas incorporadas utilizan una imagen codificada en base64 para la barra de herramientas (`<img src="src="..."`). Puedes calcular el valor base64 para una imagen con este pequeño guión: `echo base64_encode(file_get_contents($_SERVER['argv'][1]));`.

Para habilitar la plantilla, agrega un atributo `template` a la etiqueta `data_collector` en tu configuración. Por ejemplo, asumiendo que tu plantilla está en algún `AcmeDebugBundle`:

- *YAML*

```
services:
    data_collector.el_nombre_de_tu_colector:
        class: Acme\DebugBundle\Collector\Class\Name
        tags:
            - { name: data_collector, template: "AcmeDebug:Collector:templatename", id: "el_nomb
```

- *XML*

```
<service id="data_collector.el_nombre_de_tu_colector" class="Acme\DebugBundle\Collector\Class\Na
    <tag name="data_collector" template="AcmeDebug:Collector:templatename" id="el_nombre_de_tu_c
</service>
```

- *PHP*

```
    $contenedor
        ->register('data_collector.el_nombre_de_tu_colector', 'Acme\DebugBundle\Collector\Class\Name
        ->addTag('data_collector', array('template' => 'AcmeDebugBundle:Collector:templatename', 'id
    ;
```

3.57 Cómo crear un servicio Web *SOAP* en un controlador de *Symfony2*

Configurar un controlador para que actúe como un servidor SOAP es muy sencillo con un par de herramientas. Debes, por supuesto, tener instalada la extensión SOAP de *PHP*. Debido a que la extensión SOAP de *PHP*, actualmente no puede generar un WSDL, debes crear uno desde cero o utilizar un generador de un tercero.

Nota: Hay varias implementaciones de servidor *SOAP* disponibles para usarlas con *PHP*. Zend SOAP y NuSOAP son dos ejemplos. A pesar de que usamos la extensión *SOAP* de *PHP* en nuestros ejemplos, la idea general debería seguir siendo aplicable a otras implementaciones.

SOAP trabaja explotando los métodos de un objeto *PHP* a una entidad externa (es decir, la persona que utiliza el servicio *SOAP*). Para empezar, crea una clase - `HolaServicio` - que representa la funcionalidad que vas a exponer en tu servicio *SOAP*. En este caso, el servicio *SOAP* debe permitir al cliente invocar a un método llamado `hola`, el cual sucede que envía una dirección de correo electrónico:

```
namespace Acme\SoapBundle;

class HolaServicio
{
    private $cartero;

    public function __construct(\Swift_Mailer $cartero)
    {
        $this->cartero = $cartero;
    }

    public function hola($nombre)
    {

        $mensaje = \Swift_Message::newInstance()
                                ->setTo('yo@ejemplo.com')
                                ->setSubject('Hola servicio')
                                ->setBody($nombre . ' ¡di hola!');

        $this->cartero->send($mensaje);


        return 'Hola, ' . $nombre;
    }

}
```

A continuación, puedes entrenar a *Symfony* para que sea capaz de crear una instancia de esta clase. Puesto que la clase envía un correo electrónico, se ha diseñado para aceptar una instancia de `Swift_Mailer`. Usando el contenedor de servicios, podemos configurar a *Symfony* para construir un objeto *HolaServicio'* correctamente:

- *YAML*

```yaml
# app/config/config.yml
services:
    hola_servicio:
        class: Acme\DemoBundle\Services\HolaServicio
        arguments: [cartero]
```

- *XML*

```xml
<!-- app/config/config.xml -->
<services>
 <service id="hola_servicio" class="Acme\DemoBundle\Services\HolaServicio">
  <argument>cartero</argument>
 </service>
</services>
```

A continuación mostramos un ejemplo de un controlador que es capaz de manejar una petición *SOAP*. Si `indexAction()` es accesible a través de la ruta `/soap`, se puede recuperar el documento WSDL a través de `/soap?wsdl`.

```
namespace Acme\SoapBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HolaServicioController extends Controller
{
    public function indexAction()
    {
        $servidor = new \SoapServer('/ruta/a/hola.wsdl');
        $servidor->setObject($this->get('hola_servicio'));

        $respuesta = new Response();
        $respuesta->headers->set('Content-Type', 'text/xml; charset=ISO-8859-1');

        ob_start();
        $servidor->handle();
        $respuesta->setContent(ob_get_clean());

        return $respuesta;
    }
}
```

Toma nota de las llamadas a `ob_start()` y `ob_get_clean()`. Estos métodos controlan la salida almacenada temporalmente que te permite "atrapar" la salida difundida por el `$servidor->handle()`. Esto es necesario porque *Symfony* espera que el controlador devuelva un objeto `Respuesta` con la salida como "contenido". También debes recordar establecer la cabecera `Content-Type` a `text/xml`, ya que esto es lo que espera el cliente. Por lo tanto, utiliza `ob_start()` para empezar a amortiguar la `STDOUT` y usa `ob_get_clean()` para volcar la salida difundida al contenido de la `Respuesta` y limpiar la salida. Por último, estás listo para devolver la `Respuesta`.

A continuación hay un ejemplo de llamada al servicio usando el cliente NuSOAP. Este ejemplo asume que el `indexAction` del controlador de arriba es accesible a través de la ruta `/soap`:

```
$cliente = new \soapclient('http://ejemplo.com/app.php/soap?wsdl', true);

$result = $cliente->call('hola', array('nombre' => 'Scott'));
```

Abajo está un ejemplo de `WSDL`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns="urn:arnleadservicewsdl"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="urn:helloservicewsdl">
  <types>
   <xsd:schema targetNamespace="urn:holawsdl">
    <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" />
   </xsd:schema>
  </types>
  <message name="holaPeticion">
   <part name="name" type="xsd:string" />
  </message>
  <message name="holaRespuesta">
   <part name="return" type="xsd:string" />
  </message>
  <portType name="holawsdlPortType">
   <operation name="hola">
    <documentation>Hola Mundo</documentation>
    <input message="tns:holaPeticion"/>
    <output message="tns:holaRespuesta"/>
   </operation>
  </portType>
  <binding name="holawsdlBinding" type="tns:holawsdlPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="hola">
   <soap:operation soapAction="urn:arnleadservicewsdl#hello" style="rpc"/>
   <input>
    <soap:body use="encoded" namespace="urn:holawsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </input>
   <output>
    <soap:body use="encoded" namespace="urn:holawsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </output>
  </operation>
 </binding>
 <service name="holawsdl">
  <port name="holawsdlPort" binding="tns:holawsdlBinding">
   <soap:address location="http://ejemplo.com/app.php/soap" />
  </port>
 </service>
</definitions>
```

3.58 En qué difiere *Symfony2* de *symfony1*

La plataforma *Symfony2* representa una evolución significativa en comparación con la primera versión. Afortunadamente, con la arquitectura *MVC* en su núcleo, las habilidades para dominar un proyecto *Symfony1* siguen siendo muy relevantes para el desarrollo de *Symfony2*. Claro, `app.yml` se ha ido, pero el enrutado, los controladores y las plantillas permanecen.

En este capítulo, vamos a recorrer las diferencias entre *Symfony1* y *Symfony2*. Como verás, se abordan muchas tareas

de una manera ligeramente diferente. Llegarás a apreciar estas diferencias menores ya que promueven código estable, predecible, verificable y disociado de tus aplicaciones *Symfony2*.

Por lo tanto, siéntate y relájate mientras te llevamos de "entonces" a "ahora".

3.58.1 Estructura del directorio

Al examinar un proyecto *Symfony2* - por ejemplo, la Edición estándar de Symfony2 - te darás cuenta que la estructura de directorios es muy diferente a la de *Symfony1*. Las diferencias, sin embargo, son un tanto superficiales.

El directorio `app/`

En *symfony1*, tu proyecto tiene una o más aplicaciones, y cada una vive dentro del directorio `apps/` (por ejemplo, `apps/frontend`). De forma predeterminada en *Symfony2*, tienes una sola aplicación representada por el directorio `app/`. Al igual que en *Symfony1*, el directorio `app/` contiene configuración específica a esa aplicación. Este también contiene directorios caché, registro y plantillas específicas de tu aplicación, así como una clase `Kernel` (`AppKernel`), la cual es el objeto base que representa la aplicación.

A diferencia de *Symfony1*, casi no vive código *PHP* en el directorio `app/`. Este directorio no está destinado a ser el hogar de módulos o archivos de biblioteca como lo hizo en *Symfony1*. En cambio, simplemente es el hogar de la configuración y otros recursos (plantillas, archivos de traducción).

El directorio `src/`

En pocas palabras, tu verdadero código va aquí. En *Symfony2*, todo el código real de tu aplicación, vive dentro de un paquete (aproximadamente equivalente a un complemento de *Symfony1*) y, por omisión, cada paquete vive dentro del directorio `src`. De esta manera, el directorio `src` es un poco como el directorio `plugins` en *Symfony1*, pero mucho más flexible. Además, mientras que *tus* paquetes deben vivir en el directorio `src/`, los paquetes de otros fabricantes pueden vivir en el directorio `vendor/bundles`.

Para obtener una mejor imagen del directorio `src/`, primero vamos a pensar en una aplicación *Symfony1*. En primer lugar, parte de tu código probablemente viva dentro de una o más aplicaciones. Comúnmente son módulos, pero también podrían incluir otras clases *PHP* que pones en tu aplicación. Es posible que también crees un archivo `schema.yml` en el directorio `config` de tu proyecto y construyas varios archivos de modelo. Por último, para ayudar con alguna funcionalidad común, estarás usando varios complementos de terceros que viven en el directorio `plugins/`. En otras palabras, el código que impulsa tu aplicación vive en muchos lugares diferentes.

En *Symfony2*, la vida es mucho más simple porque *todo* el código *Symfony2* debe vivir en un paquete. En nuestro pretendido proyecto *Symfony1*, todo el código se *podría* trasladar a uno o más complementos (lo cual es, de hecho, una muy buena práctica). Suponiendo que todos los módulos, clases *PHP*, esquema, configuración de enrutado, etc. fueran trasladados a un complemento, el directorio `plugins/` de *Symfony1* sería muy similar al `src/` de *Symfony2*.

En pocas palabras de nuevo, el directorio `src/` es donde vive el código, activos, plantillas y la mayoría de cualquier otra cosa específica a tu proyecto.

El directorio `vendor/`

El directorio `vendor/` básicamente es el equivalente al directorio `lib/vendor/` en *Symfony1*, que fue el directorio convencional para todas las bibliotecas y paquetes de los proveedores. De manera predeterminada, encontrarás los archivos de la biblioteca *Symfony2* en este directorio, junto con varias otras bibliotecas dependientes, como Doctrine2, *Twig* y SwiftMailer. La 3ª parte de los paquetes de *Symfony2* generalmente vive en `vendor/bundles/`.

El Directorio `web/`

No ha cambiado mucho el directorio `web/`. La diferencia más notable es la ausencia de los directorios `css/`, `js/` e `images/`. Esto es intencional. Al igual que con tu código *PHP*, todos los activos también deben vivir dentro de un paquete. Con la ayuda de una consola de ordenes, el directorio `Resources/public/` de cada paquete se copia o enlaza simbólicamente al directorio `web/bundles/`. Esto nos permite mantener los activos organizados dentro de tu paquete, pero estando disponibles al público. Para asegurarte que todos los paquetes están disponibles, ejecuta la siguiente orden:

```
php app/console assets:install web
```

Nota: Esta orden es el equivalente *Symfony2* a la orden `plugin:publish-assets` de *Symfony1*.

3.58.2 Carga automática

Una de las ventajas de las plataformas modernas es nunca tener que preocuparte de requerir archivos manualmente. Al utilizar un cargador automático, puedes referirte a cualquier clase en tu proyecto y confiar en que esté disponible. La carga automática de clases ha cambiado en *Symfony2* para ser más universal, más rápida e independiente de la necesidad de vaciar la caché.

En *Symfony1*, la carga automática de clases se llevó a cabo mediante la búsqueda en todo el proyecto de la presencia de archivos de clases *PHP* y almacenamiento en caché de esta información en una matriz gigante. Esa matriz decía a *Symfony1* exactamente qué archivo contenía cada clase. En el entorno de producción, esto causó que necesitaras borrar la memoria caché cuando añadías o movías clases.

En *Symfony2*, una nueva clase - `UniversalClassLoader` - se encarga de este proceso. La idea detrás del cargador automático es simple: el nombre de tu clase (incluyendo el espacio de nombres) debe coincidir con la ruta al archivo que contiene esa clase. Tomemos como ejemplo el `FrameworkExtraBundle` de la Edición estándar de *Symfony2*:

```
namespace Sensio\Bundle\FrameworkExtraBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
// ...

class SensioFrameworkExtraBundle extends Bundle
{
    // ...
```

El archivo en sí mismo vive en `vendor/bundle/Sensio/Bundle/FrameworkExtraBundle/SensioFrameworkExtraB`
Como puedes ver, la ubicación del archivo sigue el espacio de nombres de la clase. Específicamente, el espacio de nombres, `Sensio\Bundle\FrameworkExtraBundle`, explica el directorio en que el archivo debe vivir (`vendor/bundle/Sensio/Bundle/FrameworkExtraBundle`). Esto es así porque, en el archivo `app/autoload.php`, debes configurar a *Symfony* para buscar el espacio de nombres `Sensio` en el directorio `vendor/bundle`:

```
// app/autoload.php

// ...
$cargador->registerNamespaces(array(
    // ...
    'Sensio'           => __DIR__.'/../vendor/bundles',
));
```

Si el archivo *no* vive en ese lugar exacto, recibirás un error `La clase "Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle" no existe.`

En *Symfony2*, una "clase no existe" significa que el espacio de nombres de la clase sospechosa y la ubicación física no coinciden. Básicamente, *Symfony2* está buscando en una ubicación exacta dicha clase, pero ese lugar no existe (o contiene una clase diferente). Para que una clase se cargue automáticamente, en *Symfony2* **nunca necesitas vaciar la caché**.

Como se mencionó anteriormente, para que trabaje el cargador automático, es necesario saber que el espacio de nombres `Sensio` vive en el directorio `vendor/bundles` y que, por ejemplo, el espacio de nombres `Doctrine` vive en el directorio `vendor/doctrine/lib/`. Esta asignación es controlada totalmente por ti a través del archivo `app/autoload.php`.

Si nos fijamos en el `HolaController` de la edición estándar de *Symfony2* puedes ver que este vive en el espacio de nombres `Acme\DemoBundle\Controller`. Sin embargo, el espacio de nombres `Acme` no está definido en el `app/autoload.php`. Por omisión, no es necesario configurar explícitamente la ubicación de los paquetes que viven en el directorio `src/`. El `UniversalClassLoader` está configurado para retroceder al directorio `src/` usando el método `registerNamespaceFallbacks`:

```
// app/autoload.php

// ...
$cargador->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
```

3.58.3 Usando la consola

En *symfony1*, la consola se encuentra en el directorio raíz de tu proyecto y se llama `symfony`:

```
php symfony
```

En *Symfony2*, la consola se encuentra ahora en el subdirectorio `app` y se llama `console`:

```
php app/console
```

3.58.4 Aplicaciones

En un proyecto *Symfony1*, es común tener varias aplicaciones: una para la interfaz de usuario y otra para la interfaz de administración, por ejemplo.

En un proyecto *Symfony2*, sólo tienes que crear una aplicación (una aplicación de blog, una aplicación de intranet, ...). La mayoría de las veces, si deseas crear una segunda aplicación, en su lugar podrías crear otro proyecto y compartir algunos paquetes entre ellos.

Y si tienes que separar la interfaz y las funciones de interfaz de administración de algunos paquetes, puedes crear subespacios de nombres para los controladores, subdirectorios de plantillas, diferentes configuraciones semánticas, configuración de enrutado separada, y así sucesivamente.

Por supuesto, no hay nada malo en tener varias aplicaciones en el proyecto, lo cual es una elección totalmente tuya. Una segunda aplicación significaría un nuevo directorio, por ejemplo, `mi_aplic/`, con la misma configuración básica que el directorio `app/`.

Truco: Lee la definición de un *Proyecto*, una *Aplicación*, y un *Paquete* en el glosario.

3.58.5 Paquetes y complementos

En un proyecto *Symfony1*, un complemento puede contener configuración, módulos, bibliotecas *PHP*, activos y cualquier otra cosa relacionada con tu proyecto. En *Symfony2*, la idea de un complemento es reemplazada por el "paquete". Un paquete es aún más poderoso que un complemento porque el núcleo de la plataforma *Symfony2* consta de una serie de paquetes. En *Symfony2*, los paquetes son ciudadanos de primera clase y son tan flexibles que incluso el código del núcleo en sí es un paquete.

En *Symfony1*, un complemento se debe activar dentro de la clase `ProjectConfiguration`:

```
// config/ProjectConfiguration.class.php
public function setup()
{
    $this->enableAllPluginsExcept(array(/* algunos complementos aquí */));
}
```

En *Symfony2*, los paquetes se activan en el interior del núcleo de la aplicación:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        // ...
        new Acme\DemoBundle\AcmeDemoBundle(),
    );

    return $bundles;
}
```

Enrutando (`routing.yml`) y configurando (`config.yml`)

En *Symfony1*, los archivos de configuración `routing.yml` y `app.yml` cargan cualquier complemento automáticamente. En *Symfony2*, la configuración de enrutado y de la aplicación dentro de un paquete se debe incluir manualmente. Por ejemplo, para incluir la ruta a un recurso de un paquete, puedes hacer lo siguiente:

```
# app/config/routing.yml
_hola:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

Esto cargará las rutas que se encuentren en el archivo `Resources/config/routing.yml` del `AcmeDemoBundle`. La sintaxis especial `@AcmeDemoBundle` es un atajo que, internamente, resuelve la ruta al directorio del paquete.

Puedes utilizar esta misma estrategia en la configuración de un paquete:

```
# app/config/config.yml
imports:
    - { resource: "@AcmeDemoBundle/Resources/config/config.yml" }
```

En *Symfony2*, la configuración es un poco como `app.yml` en *Symfony1*, salvo que mucho más sistemática. Con `app.yml`, simplemente puedes crear las claves que quieras. De forma predeterminada, estas entradas no tenían sentido y dependían totalmente de cómo se utilizaban en tu aplicación:

```
# algún archivo app.yml de symfony1
all:
```

```
  correo:
    from_domicilio:  foo.bar@ejemplo.com
```

En *Symfony2*, también puedes crear entradas arbitrarias bajo la clave `parameters` de tu configuración:

```
parameters:
    email.from_domicilio: foo.bar@ejemplo.com
```

Ahora puedes acceder a ella desde un controlador, por ejemplo:

```
public function holaAction($nombre)
{
    $fromAddress = $this->contenedor->getParameter('email.from_domicilio');
}
```

En realidad, la configuración de *Symfony2* es mucho más poderosa y se utiliza principalmente para configurar los objetos que puedes utilizar. Para más información, consulta el capítulo titulado "*Contenedor de servicios* (Página 237)".

- **Flujo de trabajo**
  - *Cómo crear y guardar un proyecto Symfony2 en git* (Página 275)
- **Controladores**
  - *Cómo personalizar páginas de error* (Página 277)
  - *Cómo definir controladores como servicios* (Página 278)
- **Enrutando**
  - *Cómo forzar las rutas para utilizar siempre HTTPS* (Página 279)
  - *Cómo permitir un carácter "/" en un parámetro de ruta* (Página 280)
- **Manejando activos JavaScript y CSS**
  - *Cómo utilizar Assetic para gestionar activos* (Página 281)
  - *Cómo minimizar JavaScript y hojas de estilo con YUI Compressor* (Página 285)
  - *Cómo utilizar Assetic para optimizar imágenes con funciones Twig* (Página 287)
  - *Cómo aplicar un filtro Assetic a una extensión de archivo especifica* (Página 291)
- **Interacción con la base de datos (Doctrine)**
  - *Cómo manejar archivos subidos con Doctrine* (Página 294)
  - *Extensiones Doctrine: Timestampable: Sluggable, Translatable, etc.* (Página 300)
  - *Registrando escuchas y suscriptores de eventos* (Página 300)
  - *Cómo utiliza Doctrine la capa DBAL* (Página 304)
  - *Cómo generar entidades de una base de datos existente* (Página 302)
  - *Cómo trabajar con varios gestores de entidad* (Página 307)
  - *Registrando funciones DQL personalizadas* (Página 308)
- **Formularios y Validación**
  - *Cómo personalizar la reproducción de un formulario* (Página 309)
  - *Cómo crear un tipo de campo de formulario personalizado* (Página 322)
  - *Cómo crear una restricción de validación personalizada* (Página 323)
  - (doctrine) *Cómo manejar archivos subidos con Doctrine* (Página 294)

- **Configuración y el contenedor de servicios**
  - *Cómo dominar y crear nuevos entornos* (Página 324)
  - *Cómo configurar parámetros externos en el contenedor de servicios* (Página 329)
  - *Cómo utilizar el patrón fábrica para crear servicios* (Página 331)
  - *Cómo gestionar dependencias comunes con servicios padre* (Página 335)
  - *Cómo utilizar PdoSessionStorage para almacenar sesiones en la base de datos* (Página 344)
- **Paquetes**
  - *Estructura de un paquete y buenas prácticas* (Página 346)
  - *Cómo utilizar la herencia de paquetes para redefinir partes de un paquete* (Página 350)
  - *Cómo exponer la configuración semántica de un paquete* (Página 350)
- **Correo electrónico**
  - *Cómo enviar correo electrónico* (Página 358)
  - *Cómo utilizar Gmail para enviar mensajes de correo electrónico* (Página 361)
  - *Cómo trabajar con correos electrónicos durante el desarrollo* (Página 361)
  - *Cómo organizar el envío de correo electrónico* (Página 363)
- **Probando**
  - *Cómo simular autenticación HTTP en una prueba funcional* (Página 365)
  - *Cómo probar la interacción de varios clientes* (Página 365)
  - *Cómo utilizar el generador de perfiles en una prueba funcional* (Página 366)
  - *Cómo probar repositorios Doctrine* (Página 367)
- **Seguridad**
  - *Cómo agregar la funcionalidad "recuérdame" al inicio de sesión* (Página 370)
  - *Cómo implementar tu propio votante para agregar direcciones IP a la lista negra* (Página 373)
  - *Listas de control de acceso (ACL)* (Página 376)
  - *Conceptos ACL avanzados* (Página 379)
  - *Cómo forzar HTTPS o HTTP a diferentes URL* (Página 382)
  - *Cómo personalizar el formulario de acceso* (Página 383)
  - *Cómo proteger cualquier servicio o método de tu aplicación* (Página 389)
  - *Cómo cargar usuarios de la base de datos con seguridad (la entidad Proveedor)* (Página 393)
  - *Cómo crear un proveedor de usuario personalizado* (Página 393)
  - *Cómo crear un proveedor de autenticación personalizado* (Página 394)
- **Memoria caché**
  - *Cómo utilizar Varnish para acelerar mi sitio web* (Página 402)
- **Plantillas**
  - *Cómo usar plantillas PHP en lugar de Twig* (Página 403)
- **Herramientas, registro y funcionamiento interno**

- *Cómo cargar clases automáticamente* (Página 407)
- *Cómo localizar archivos* (Página 409)
- *Cómo crear la consola/línea de ordenes* (Página 412)
- *Cómo optimizar tu entorno de desarrollo para depuración* (Página 416)
- *Cómo utilizar Monolog para escribir Registros* (Página 417)

- **Servicios Web**
  - *Cómo crear un servicio Web SOAP en un controlador de Symfony2* (Página 428)
- **Extendiendo Symfony**
  - *Cómo extender una clase sin necesidad de utilizar herencia* (Página 421)
  - *Cómo personalizar el comportamiento de un método sin utilizar herencia* (Página 423)
  - *Cómo registrar un nuevo formato de petición y tipo MIME* (Página 424)
  - *Cómo crear un colector de datos personalizado* (Página 425)
- **Symfony2 para usuarios de symfony1**
  - *En qué difiere Symfony2 de symfony1* (Página 430)

Lee el *Recetario* (Página 275).

Parte IV

Documentos de referencia

Consigue respuestas rápidamente con los documentos de referencia:

Documentos de referencia

4.1 Configurando el `FrameworkBundle` ("framework")

Este documento de referencia es un trabajo en progreso. Este debe ser preciso, pero todavía no están totalmente cubiertas todas las opciones.

El `FrameworkBundle` contiene la mayor parte de la funcionalidad "base" de la plataforma y se puede configurar bajo la clave `framework` en la configuración de tu aplicación. Esto incluye ajustes relacionados con sesiones, traducción, formularios, validación, enrutado y mucho más.

4.1.1 Configurando

charset

tipo: `string` **predeterminado**: `UTF-8`

El conjunto de caracteres que se utiliza en toda la plataforma. Se convierte en el parámetro del contenedor de servicios llamado `kernel.charset`.

secret

tipo: `string` **requerido**

Esta es una cadena que debe ser única para tu aplicación. En la práctica, se utiliza para generar las señales CSRF, pero esta se podría utilizar en cualquier otro contexto donde una cadena única sea útil. Se convierte en el parámetro llamado `kernel.secret` del contenedor de servicios.

ide

tipo: `string` **predeterminado**: `null`

Si estás utilizando un IDE como TextMate o Vim Mac, *Symfony* puede convertir todas las rutas de archivo en un mensaje de excepción en un enlace, el cual abrirá el archivo en el IDE.

Si usas TextMate o Vim Mac, simplemente puedes utilizar uno de los siguientes valores integrados:

- `textmate`
- `macvim`

También puedes especificar una cadena personalizada como enlace al archivo. Si lo haces, debes duplicar todos los signos de porcentaje (`%`) para escapar ese carácter. Por ejemplo, la cadena completa de TextMate se vería así:

```
framework:
    ide:  "txmt://open?url=file://%%f&line=%%l"
```

Por supuesto, ya que cada desarrollador utiliza un IDE diferente, es mejor poner esto a nivel del sistema. Esto se puede hacer estableciendo en php.ini el valor de `xdebug.file_link_format` a la cadena de enlace al archivo. Si estableces este valor de configuración, entonces no es necesario determinar la opción `ide`.

test

tipo: `Boolean`

Si este parámetro de configuración está presente, se cargan los servicios relacionados con las pruebas de tu aplicación. Este valor debe estar presente en tu entorno `test` (por lo general a través de `app/config/config_test.yml`). Para más información, consulta *Probando* (Página 137).

form

csrf_protection

4.1.2 Configuración predeterminada completa

- *YAML*

```
framework:

    # Configuración general
    charset:              ~
    secret:               ~ # Required
    ide:                  ~
    test:                 ~

    # configuración de formulario
    form:
```

```
    enabled:              true
csrf_protection:
    enabled:              true
    field_name:           _token

# configuración esi
esi:
    enabled:              true

# configuración del generador de perfiles
profiler:
    only_exceptions:      false
    only_master_requests:  false
    dsn:                  sqlite:%kernel.cache_dir%/profiler.db
    username:
    password:
    lifetime:             86400
    matcher:
        ip:                   ~
        path:                 ~
        service:              ~

# configuración de enrutado
router:
    resource:             ~ # Required
    type:                 ~
    http_port:            80
    https_port:           443

# configuración de sesión
session:
    auto_start:           ~
    default_locale:       en
    storage_id:           session.storage.native
    name:                 ~
    lifetime:             ~
    path:                 ~
    domain:               ~
    secure:               ~
    httponly:             ~

# configuración de plantillas
templating:
    assets_version:       ~
    assets_version_format:  ~
    assets_base_urls:
        http:                 []
        ssl:                  []
    cache:                ~
    engines:              # Required
    form:
        resources:        [FrameworkBundle:Form]

        # Ejemplo:
        - twig
    loaders:              []
    packages:
```

```
            # Prototipo
            name:
                version:              ~
                version_format:       ~
                base_urls:
                    http:                 []
                    ssl:                  []

    # configuración de traducción
    translator:
        enabled:              true
        fallback:             en

    # configuración de validación
    validation:
        enabled:              true
        cache:                ~
        enable_annotations:   false

    # configuración de anotaciones
    annotations:
        cache:                file
        file_cache_dir:       %kernel.cache_dir%/annotations
        debug:                true
```

4.2 Referencia de configuración de `AsseticBundle`

4.2.1 Configuración predeterminada completa

- *YAML*

```
assetic:
    debug:                true
    use_controller:       true
    read_from:            %kernel.root_dir%/../web
    write_to:             %assetic.read_from%
    java:                 /usr/bin/java
    node:                 /usr/bin/node
    sass:                 /usr/bin/sass
    bundles:

        # Predeterminados (todos los paquetes actualmente registrados):
        - FrameworkBundle
        - SecurityBundle
        - TwigBundle
        - MonologBundle
        - SwiftmailerBundle
        - DoctrineBundle
        - AsseticBundle
        - ...

    assets:

        # Prototipo
        name:
```

```
            inputs:               []
            filters:              []
            options:

                # Prototipo
                name:                 []
    filters:

        # Prototipo
        name:                 []
    twig:
        functions:

            # Prototipo
            name:                 []
```

4.3 Referencia de configuración

- *YAML*

```
doctrine:
    dbal:
        default_connection:   default
        connections:
            default:
                dbname:               database
                host:                 localhost
                port:                 1234
                user:                 user
                password:             secret
                driver:               pdo_mysql
                driver_class:         MyNamespace\MyDriverImpl
                options:
                    foo: bar
                path:                 %kernel.data_dir%/data.sqlite
                memory:               true
                unix_socket:          /tmp/mysql.sock
                wrapper_class:        MyDoctrineDbalConnectionWrapper
                charset:              UTF8
                logging:              %kernel.debug%
                platform_service:     MyOwnDatabasePlatformService
                mapping_types:
                    enum: string
            conn1:
                # ...
        types:
            custom: Acme\HelloBundle\MyCustomType
    orm:
        auto_generate_proxy_classes:    false
        proxy_namespace:                Proxies
        proxy_dir:                      %kernel.cache_dir%/doctrine/orm/Proxies
        default_entity_manager:         default # The first defined is used if not set
        entity_managers:
            default:
                # The name of a DBAL connection (the one marked as default is used if not set)
                connection:                     conn1
```

```
                mappings: # Required
                    AcmeHelloBundle: ~
                class_metadata_factory_name:    Doctrine\ORM\Mapping\ClassMetadataFactory
                # All cache drivers have to be array, apc, xcache or memcache
                metadata_cache_driver:          array
                query_cache_driver:             array
                result_cache_driver:
                    type:           memcache
                    host:           localhost
                    port:           11211
                    instance_class: Memcache
                    class:          Doctrine\Common\Cache\MemcacheCache
                dql:
                    string_functions:
                        test_string: Acme\HelloBundle\DQL\StringFunction
                    numeric_functions:
                        test_numeric: Acme\HelloBundle\DQL\NumericFunction
                    datetime_functions:
                        test_datetime: Acme\HelloBundle\DQL\DatetimeFunction
            em2:
                # ...
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:dbal default-connection="default">
            <doctrine:connection
                name="default"
                dbname="database"
                host="localhost"
                port="1234"
                user="user"
                password="secret"
                driver="pdo_mysql"
                driver-class="MyNamespace\MyDriverImpl"
                path="%kernel.data_dir%/data.sqlite"
                memory="true"
                unix-socket="/tmp/mysql.sock"
                wrapper-class="MyDoctrineDbalConnectionWrapper"
                charset="UTF8"
                logging="%kernel.debug%"
                platform-service="MyOwnDatabasePlatformService"
            >
                <doctrine:option key="foo">bar</doctrine:option>
                <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
            </doctrine:connection>
            <doctrine:connection name="conn1" />
            <doctrine:type name="custom">Acme\HolaBundle\MyCustomType</doctrine:type>
        </doctrine:dbal>

        <doctrine:orm default-entity-manager="default" auto-generate-proxy-classes="false" proxy
            <doctrine:entity-manager name="default" query-cache-driver="array" result-cache-driv
                <doctrine:metadata-cache-driver type="memcache" host="localhost" port="11211" in
```

```
                    <doctrine:mapping name="AcmeHelloBundle" />
                    <doctrine:dql>
                        <doctrine:string-function name="test_string">Acme\HelloBundle\DQL\StringFunct
                        <doctrine:numeric-function name="test_numeric">Acme\HelloBundle\DQL\NumericFu
                        <doctrine:datetime-function name="test_datetime">Acme\HelloBundle\DQL\Datetim
                    </doctrine:dql>
                </doctrine:entity-manager>
                <doctrine:entity-manager name="em2" connection="conn2" metadata-cache-driver="apc">
                    <doctrine:mapping
                        name="DoctrineExtensions"
                        type="xml"
                        dir="%kernel.root_dir%/../src/vendor/DoctrineExtensions/lib/DoctrineExtensi
                        prefix="DoctrineExtensions\Entity"
                        alias="DExt"
                    />
                </doctrine:entity-manager>
            </doctrine:orm>
        </doctrine:config>
    </container>
```

4.3.1 Describiendo la configuración

El siguiente ejemplo de configuración muestra todos los valores de configuración predeterminados que resuelve *ORM*:

```
doctrine:
    orm:
        auto_mapping: true
        # the standard distribution overrides this to be true in debug, false otherwise
        auto_generate_proxy_classes: false
        proxy_namespace: Proxies
        proxy_dir: %kernel.cache_dir%/doctrine/orm/Proxies
        default_entity_manager: default
        metadata_cache_driver: array
        query_cache_driver: array
        result_cache_driver: array
```

Hay un montón de opciones de configuración que puedes utilizar para redefinir ciertas clases, pero solamente son para casos de uso muy avanzado.

Controladores de caché

Para los controladores de memoria caché puedes especificar los valores “array”, “apc”, “memcache” o “xcache”.

El siguiente ejemplo muestra una descripción de los ajustes de la memoria caché:

```
doctrine:
    orm:
        auto_mapping: true
        metadata_cache_driver: apc
        query_cache_driver: xcache
        result_cache_driver:
            type: memcache
            host: localhost
            port: 11211
            instance_class: Memcache
```

Configurando la asignación

La definición explícita de todas las entidades asignadas es la única configuración necesaria para el *ORM* y hay varias opciones de configuración que puedes controlar. Existen las siguientes opciones de configuración para una asignación:

- `type` Uno de `annotation`, `xml`, `yml`, `php` o `staticphp`. Esto especifica cual tipo de metadatos usa el tipo de tu asignación.
- `dir` Ruta a la asignación o archivos de entidad (dependiendo del controlador). Si esta ruta es relativa, se supone que es relativa a la raíz del paquete. Esto sólo funciona si el nombre de tu asignación es un nombre de paquete. Si deseas utilizar esta opción para especificar rutas absolutas debes prefijar la ruta con los parámetros del núcleo existentes en el DIC (por ejemplo %kernel.root_dir %).
- `prefix` Un prefijo común del espacio de nombres que comparten todas las entidades de esta asignación. Este prefijo nunca debe entrar en conflicto con otros prefijos de asignación definidos, de otra manera *Doctrine* no podrá encontrar algunas de tus entidades. Esta opción tiene predeterminado el espacio de nombres paquete + `Entidad`, por ejemplo, para un paquete llamado `AcmeHelloBundle` el prefijo sería `Acme\HolaBundle\Entidad`.
- `alias` *Doctrine* ofrece una forma de simplificar el espacio de nombres de la entidad, para utilizar nombres más cortos en las consultas `DQL` o para acceder al Repositorio. Cuando utilices un paquete el alias predeterminado es el nombre del paquete.
- `is_bundle` Esta opción es un valor derivado de `dir` y por omisión se establece en `true` si dir es relativo provisto por un `file_exists()` comprueba que devuelve `false`. Este es `false` si al comprobar la existencia devuelve `true`. En este caso se ha especificado una ruta absoluta y es más probable que los archivos de metadatos estén en un directorio fuera del paquete.

4.3.2 Configurando DBAL *Doctrine*

Nota: `DoctrineBundle` apoya todos los parámetros por omisión que los controladores de *Doctrine* aceptan, convertidos a la nomenclatura estándar de *XML* o *YAML* que *Symfony* hace cumplir. Consulta la Documentación DBAL de *Doctrine* para más información.

Además de las opciones por omisión de *Doctrine*, hay algunas relacionadas con *Symfony* que se pueden configurar. El siguiente bloque muestra todas las posibles claves de configuración:

- *YAML*

```
doctrine:
    dbal:
        dbname:               database
        host:                 localhost
        port:                 1234
        user:                 user
        password:             secret
        driver:               pdo_mysql
        driver_class:         MyNamespace\MyDriverImpl
        options:
            foo: bar
        path:                 %kernel.data_dir%/data.sqlite
        memory:               true
        unix_socket:          /tmp/mysql.sock
        wrapper_class:        MyDoctrineDbalConnectionWrapper
        charset:              UTF8
        logging:              %kernel.debug%
        platform_service:     MyOwnDatabasePlatformService
```

```
            mapping_types:
                enum: string
            types:
                custom: Acme\HolaBundle\MyCustomType
```

- *XML*

```
<!-- xmlns:doctrine="http://symfony.com/schema/dic/doctrine" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/do

<doctrine:config>
    <doctrine:dbal
        name="default"
        dbname="database"
        host="localhost"
        port="1234"
        user="user"
        password="secret"
        driver="pdo_mysql"
        driver-class="MyNamespace\MyDriverImpl"
        path="%kernel.data_dir%/data.sqlite"
        memory="true"
        unix-socket="/tmp/mysql.sock"
        wrapper-class="MyDoctrineDbalConnectionWrapper"
        charset="UTF8"
        logging="%kernel.debug%"
        platform-service="MyOwnDatabasePlatformService"
    >
        <doctrine:option key="foo">bar</doctrine:option>
        <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
        <doctrine:type name="custom">Acme\HolaBundle\MyCustomType</doctrine:type>
    </doctrine:dbal>
</doctrine:config>
```

Si deseas configurar varias conexiones en YAML, ponlas bajo la clave `connections` y dales un nombre único:

```
doctrine:
    dbal:
        default_connection:       default
        connections:
            default:
                dbname:           Symfony2
                user:             root
                password:         null
                host:             localhost
            customer:
                dbname:           customer
                user:             root
                password:         null
                host:             localhost
```

El servicio `database_connection` siempre se refiere a la conexión *default*, misma que es la primera definida o la configurada a través del parámetro `default_connection`.

Cada conexión también es accesible a través del servicio `doctrine.dbal.[name]_connection` donde `[name]` es el nombre de la conexión.

4.4 Referencia en configurando `Security`

El sistema de seguridad es una de las piezas más poderosas de *Symfony2*, y en gran medida se puede controlar por medio de su configuración.

4.4.1 Configuración predeterminada completa

La siguiente es la configuración predeterminada para el sistema de seguridad completo. Cada parte se explica en la siguiente sección.

- *YAML*

```
# app/config/security.yml
security:
    access_denied_url: /foo/error403

    always_authenticate_before_granting: false

    # la estrategia puede ser: none, migrate, invalidate
    session_fixation_strategy: migrate

    access_decision_manager:
        strategy: affirmative
        allow_if_all_abstain: false
        allow_if_equal_granted_denied: true

    acl:
        connection: default # algún nombre configurado en la sección doctrine.dbal
        tables:
            class: acl_classes
            entry: acl_entries
            object_identity: acl_object_identities
            object_identity_ancestors: acl_object_identity_ancestors
            security_identity: acl_security_identities
        cache:
            id: service_id
            prefix: sf2_acl_
        voter:
            allow_if_object_identity_unavailable: true

    encoders:
        somename:
            class: Acme\DemoBundle\Entity\User
        Acme\DemoBundle\Entity\User: sha512
        Acme\DemoBundle\Entity\User: plaintext
        Acme\DemoBundle\Entity\User:
            algorithm: sha512
            encode_as_base64: true
            iterations: 5000
        Acme\DemoBundle\Entity\User:
            id: my.custom.encoder.service.id

    providers:
        memory:
            name: memory
            users:
                foo: { password: foo, roles: ROLE_USER }
```

```
            bar: { password: bar, roles: [ROLE_USER, ROLE_ADMIN] }
        entity:
            entity: { class: SecurityBundle:User, property: nombreusuario }

    factories:
        MyFactory: %kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.

    firewalls:
        somename:
            pattern: .*
            request_matcher: some.service.id
            access_denied_url: /foo/error403
            access_denied_handler: some.service.id
            entry_point: some.service.id
            provider: name
            context: name
            stateless: false
            x509:
                provider: name
            http_basic:
                provider: name
            http_digest:
                provider: name
            form_login:
                check_path: /login_check
                login_path: /login
                use_forward: false
                always_use_default_target_path: false
                default_target_path: /
                target_path_parameter: _target_path
                use_referer: false
                failure_path: /foo
                failure_forward: false
                failure_handler: some.service.id
                success_handler: some.service.id
                username_parameter: _username
                password_parameter: _password
                csrf_parameter: _csrf_token
                csrf_page_id: form_login
                csrf_provider: my.csrf_provider.id
                post_only: true
                remember_me: false
            remember_me:
                token_provider: name
                key: someS3cretKey
                name: NameOfTheCookie
                lifetime: 3600 # in seconds
                path: /foo
                domain: somedomain.foo
                secure: true
                httponly: true
                always_remember_me: false
                remember_me_parameter: _remember_me
            logout:
                path:   /logout
                target: /
                invalidate_session: false
                delete_cookies:
```

```
                a: { path: null, domain: null }
                b: { path: null, domain: null }
            handlers: [some.service.id, another.service.id]
            success_handler: some.service.id
        anonymous: ~

access_control:
    -
        path: ^/foo
        host: mydomain.foo
        ip: 192.0.0.0/8
        roles: [ROLE_A, ROLE_B]
        requires_channel: https

role_hierarchy:
    ROLE_SUPERADMIN: ROLE_ADMIN
    ROLE_SUPERADMIN: 'ROLE_ADMIN, ROLE_USER'
    ROLE_SUPERADMIN: [ROLE_ADMIN, ROLE_USER]
    anything: { id: ROLE_SUPERADMIN, value: 'ROLE_USER, ROLE_ADMIN' }
    anything: { id: ROLE_SUPERADMIN, value: [ROLE_USER, ROLE_ADMIN] }
```

4.4.2 Configurando el formulario de acceso

Cuando usas el escucha de autenticación `form_login` bajo un cortafuegos, hay varias opciones comunes para experimentar en la configuración del "formulario de acceso":

Procesando el formulario de acceso

- `login_path` (tipo: `string`, predeterminado: `/login`) Esta es la *URL* a que el usuario será redirigido (a menos que `use_forward` se haya fijado en `true`) cuando él/ella intente acceder a un recurso protegido, pero no está totalmente autenticado.

  Esta **URL** debe ser accesible por un usuario normal, no autenticado, de lo contrario puede crear un bucle de redireccionamiento. Para más información, consulta "*Evitando errores comunes* (Página 193)".

- `check_path` (tipo: `string`, predeterminado: `/login_check`) Esta es la *URL* en la cual se debe presentar el formulario de acceso. El cortafuegos intercepta cualquier petición (sólo las peticiones `POST`, por omisión) a esta *URL* y procesa las credenciales presentadas.

  Asegúrate de que esta dirección está cubierta por el cortafuegos principal (es decir, no crees un servidor de seguridad independiente sólo para la *URL* `check_path`).

- `use_forward` (tipo: `Boolean`, predeterminado: `false`) Si deseas que el usuario sea remitido al formulario de acceso en vez de ser redirigido, fija esta opción a `true`.

- `username_parameter` (tipo: `string`, predeterminado: `_username`) Este es el nombre del campo que debes dar al campo *nombre de usuario* de tu formulario de acceso. Cuando se presenta el formulario a `check_path`, el sistema de seguridad buscará un parámetro POST con este nombre.

- `password_parameter` (tipo: `string`, predeterminado: `_password`) Este es el nombre del campo que debes dar al campo de la *contraseña* de tu formulario de acceso. Cuando se presenta el formulario a `check_path`, el sistema de seguridad buscará un parámetro POST con este nombre.

- `post_only` (tipo: `Boolean`, predeterminado: `true`) De forma predeterminada, debes enviar tu formulario de acceso a la *URL* `check_path` como una petición *POST*. Al establecer esta opción a `true`, puedes enviar una petición GET a la *URL* `check_path`.

Redirigiendo después del inicio de sesión

- `always_use_default_target_path` (tipo: `Boolean`, predeterminado: `false`)
- `default_target_path` (tipo: `string`, predeterminado: `/`)
- `target_path_parameter` (tipo: `string`, predeterminado: `_target_path`)
- `use_referer` (tipo: `Boolean`, predeterminado: `false`)

4.5 Configurando SwiftmailerBundle

4.5.1 Configuración predeterminada completa

- *YAML*

```
swiftmailer:
    transport:            smtp
    username:             ~
    password:             ~
    host:                 localhost
    port:                 false
    encryption:           ~
    auth_mode:            ~
    spool:
        type:                 file
        path:                 %kernel.cache_dir%/swiftmailer/spool
    sender_domicilio:       ~
    antiflood:
        threshold:            99
        sleep:                0
    delivery_domicilio:     ~
    disable_delivery:     ~
    logging:              true
```

4.6 Referencia de configuración de `TwigBundle`

- *YAML*

```
twig:
    form:
        resources:

            # predeterminado:
            - div_base.html.twig

            # Ejemplo:
            - MiBundle::formulario.html.twig
    globals:

        # Ejemplos:
        foo:                  "@bar"
        pi:                   3.14

        # Prototipo
```

```
        key:
            id:                   ~
            type:                 ~
            value:                ~
    autoescape:           ~
    base_template_class:  ~ # Ejemplo: Twig_Template
    cache:                %kernel.cache_dir%/twig
    charset:              %kernel.charset%
    debug:                %kernel.debug%
    strict_variables:     ~
    auto_reload:          ~
    exception_controller:  Symfony\Bundle\TwigBundle\Controller\ExceptionController::showAction
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:twig="http://symfony.com/schema/dic/twig"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/twig http://symfony.com/schema/dic/doctrin

    <twig:config auto-reload="%kernel.debug%" autoescape="true" base-template-class="Twig_Templa
        <twig:form>
            <twig:resource>MiBundle::formulario.html.twig</twig:resource>
        </twig:form>
        <twig:global key="foo" id="bar" type="service" />
        <twig:global key="pi">3.14</twig:global>
    </twig:config>
</container>
```

- *PHP*

```
$contenedor->loadFromExtension('twig', array(
    'form' => array(
        'resources' => array(
            'MiBundle::formulario.html.twig',
        )
     ),
     'globals' => array(
         'foo' => '@bar',
         'pi'  => 3.14,
     ),
     'auto_reload'         => '%kernel.debug%',
     'autoescape'          => true,
     'base_template_class' => 'Twig_Template',
     'cache'               => '%kernel.cache_dir%/twig',
     'charset'             => '%kernel.charset%',
     'debug'               => '%kernel.debug%',
     'strict_variables'    => false,
));
```

4.6.1 Configurando

exception_controller

tipo: `string` **predeterminado**: `Symfony\\Bundle\\TwigBundle\\Controller\\ExceptionController::showAct`

Este es el controlador que se activa después de una excepción en cualquier lugar de tu aplicación. El controlador predeterminado (`Symfony\Bundle\TwigBundle\Controller\ExceptionController`) es el responsable de reproducir plantillas específicas en diferentes condiciones de error (consulta *Cómo personalizar páginas de error* (Página 277)). La modificación de esta opción es un tema avanzado. Si necesitas personalizar una página de error debes utilizar el enlace anterior. Si necesitas realizar algún comportamiento en una excepción, debes añadir un escucha para el evento `kernel.exception` (consulta *Habilitando escuchas personalizados* (Página 561)).

4.7 Referencia de configuración

- *YAML*

```
monolog:
    handlers:

        # Ejemplos:
        syslog:
            type:                 stream
            path:                 /var/log/symfony.log
            level:                ERROR
            bubble:               false
            formatter:            my_formatter
        main:
            type:                 fingerscrossed
            action_level:         WARNING
            buffer_size:          30
            handler:              custom
        custom:
            type:                 service
            id:                   my_handler

        # Prototipo
        name:
            type:                  ~ # Required
            id:                    ~
            priority:              0
            level:                 DEBUG
            bubble:                true
            path:                 %kernel.logs_dir%/%kernel.environment%.log
            ident:                 false
            facility:              user
            max_files:             0
            action_level:          WARNING
            stop_buffering:        true
            buffer_size:           0
            handler:               ~
            members:               []
            from_correo:            ~
            to_correo:              ~
            subject:               ~
            correo_prototype:
                id:      ~ # Requerido (cuando se utiliza el correo_prototype)
                method: ~
            formatter:             ~
```

- *XML*

```xml
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="syslog"
            type="stream"
            path="/var/log/symfony.log"
            level="error"
            bubble="false"
            formatter="my_formatter"
        />
        <monolog:handler
            name="main"
            type="fingerscrossed"
            action-level="warning"
            handler="custom"
        />
        <monolog:handler
            name="custom"
            type="service"
            id="my_handler"
        />
    </monolog:config>
</container>
```

Nota: Cuando está habilitado el generador de perfiles, se agrega un controlador para almacenar los mensajes del registro en el generador de perfiles. El generador de perfiles utiliza el nombre "debug" por lo tanto está reservado y no se puede utilizar en la configuración.

4.8 Configurando WebProfiler

4.8.1 Configuración predeterminada completa

- *YAML*

```yaml
web_profiler:

    # muestra información secundaria para abreviar la barra de herramientas
    verbose:             true

    # Muestra la barra de depuración web en la parte inferior de la páginas con un resumen
    # de información del generador de perfiles
    toolbar:             false

    # te da la oportunidad de analizar los datos recogidos antes de la siguiente redirección
    intercept_redirects: false
```

4.9 Referencia de tipos para formulario

4.9.1 Tipo de campo `birthday`

Un campo *date* (Página 469) que se especializa en el manejo de fechas de cumpleaños.

Se puede reproducir como un cuadro de texto, tres cuadros de texto (mes, día y año), o tres cuadros de selección.

Este tipo esencialmente es el mismo que el tipo *date* (Página 469), pero con un predeterminado más apropiado para la opción years (Página 459). La opción predeterminada de years (Página 459) es 120 años atrás del año en curso.

Tipo de dato subyacente	puede ser `DateTime`, `string`, `timestamp`, o `array` (ve la *input option* (Página 471))
Reproducido como	pueden ser tres cajas de selección o 1 o 3 cajas de texto, basándose en la opción widget (Página 459)
Opciones	▪ years (Página 459)
Opciones heredadas	▪ widget (Página 459) ▪ input (Página 460) ▪ months (Página 460) ▪ days (Página 460) ▪ format (Página 460) ▪ pattern (Página 460) ▪ data_timezone (Página 461) ▪ user_timezone (Página 461)
Tipo del padre	*date* (Página 469)
Clase	`Symfony\Component\Form\Extension\Core\Type\Birthda`

Opciones del campo

`years`

tipo: `array` **predeterminado**: 120 años atrás de la fecha actual

Lista de años disponibles para el tipo de campo `year`. Esta opción sólo es relevante cuando la opción widget (Página 459) está establecida en `choice`.

Opciones heredadas

Estas opciones las hereda del tipo *date* (Página 469):

`widget`

tipo: `string` **predeterminado**: `choice`

La forma básica en que se debe reproducir este campo. Puede ser una de las siguientes:

- `choice`: reproduce tres entradas de selección. El orden de los selectores se define en la opción `pattern`.
- `text`: reproduce tres campos de entrada tipo texto (mes, día, año).
- `single_text`: reproduce un sólo cuadro de tipo texto. La entrada del usuario se valida en base a la opción `format`.

input

tipo: `string` **predeterminado**: `datetime`

El formato del dato *input* - es decir, el formato de la fecha en que se almacena en el objeto subyacente. Los valores válidos son los siguientes:

- `string` (por ejemplo `2011-06-05`)
- `datetime` (un objeto `DateTime`)
- `array` (por ejemplo `array('anio' => 2011, 'mes' => 06, 'dia' => 05)`)
- `timestamp` (por ejemplo `1307232000`)

El valor devuelto por el formulario también se normaliza de nuevo a este formato.

months

tipo: `array` **predeterminado**: entre 1 y 12

Lista de los meses disponibles para el tipo de campo `month`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

days

tipo: `array` **predeterminado**: entre 1 y 31

Lista de los días disponibles para el tipo de campo `day`. Esta opción sólo es relevante cuando la opción widget (Página 459) está establecida en `choice`:

```
'days' => range(1,31)
```

format

tipo: `integer` o `string` **predeterminado**: `IntlDateFormatter::MEDIUM`

Opción pasada a la clase `IntlDateFormatter`, utilizada para transformar la entrada del usuario al formato adecuado. Esto es crítico cuando la opción widget (Página 459) está establecida en `single_text`, y definirá la forma de transformar la entrada. De manera predeterminada, el formato es determinado basándose en la configuración regional del usuario actual; la puedes redefinir pasando el formato como una cadena.

pattern

tipo: `string`

Esta opción sólo es relevante cuando el widget (Página 459) se ajusta a `choice`. El patrón predeterminado está basado en la opción format (Página 460), y trata de coincidir con los caracteres `M`, `d`, e `y` en el patrón del formato. Si no hay coincidencia, el valor predeterminado es la cadena `{{ year }}-{{ month }}-{{ day }}`. Los elementos para esta opción son:

- `{{ year }}`: Reemplazado con el elemento gráfico `year`
- `{{ month }}`: Reemplazado con el elemento gráfico `month`
- `{{ day }}`: Reemplazado por el elemento gráfico `day`

data_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria en que se almacenan los datos entrantes. Esta debe ser una de las zonas horarias compatibles con PHP

user_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria para mostrar los datos al usuario (y por lo tanto también los datos que el usuario envía). Esta debe ser una de las zonas horarias compatibles con PHP

4.9.2 Tipo de campo checkbox

Crea una única casilla de entrada. Esta siempre se debe utilizar para un campo que tiene un valor booleano: si la casilla está marcada, el campo se establece en `true`, si la casilla está sin marcar, el valor se establece en `false`.

Reproducido como	campo `input text`
Opciones	▪ value (Página 461)
Opciones heredadas	▪ required (Página 462) ▪ label (Página 462) ▪ read_only (Página 462) ▪ error_bubbling (Página 462)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\Checkbo`

Ejemplo de uso

```
$generador->add('public', 'checkbox', array(
    'label'     => '¿Mostrar esta entrada al público?',
    'required'  => false,
));
```

Opciones del campo

value

tipo: `mixed` **predeterminado**: `1`

El valor utilizado realmente como valor de la casilla de verificación. Esto no afecta al valor establecido en tu objeto.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.3 Tipo de campo `choice`

Un campo multipropósito para que el usuario pueda "elegir" una o más opciones. Este se puede representar como una etiqueta `select`, botones de radio o casillas de verificación.

Para utilizar este campo, debes especificar *algún* elemento de la `choice_list` o `choices`.

Reproducido como	pueden ser varias etiquetas (ve más adelante)
Opciones	■ choices (Página 464) ■ choice_list (Página 464) ■ multiple (Página 464) ■ expanded (Página 464) ■ preferred_choices (Página 464) ■ empty_value (Página 465)
Opciones heredadas	■ required (Página 465) ■ label (Página 465) ■ read_only (Página 466) ■ error_bubbling (Página 466)
Tipo del padre	*form* (Página 483) (si `expanded`), de lo contrario `field`
Clase	`Symfony\Component\Form\Extension\Core\Type\ChoiceT`

Ejemplo de uso

La forma fácil de utilizar este campo es especificando las opciones directamente a través de la opción `choices`. La clave de la matriz se convierte en el valor que en realidad estableces en el objeto subyacente (por ejemplo, `m`), mientras que el valor es lo que el usuario ve en el formulario (por ejemplo, `Hombre`).

```
$generador->add('genero', 'choice', array(
    'choices'   => array('m' => 'Masculino', 'f' => 'Femenino'),
    'required'  => false,
));
```

Al establecer `multiple` a `true`, puedes permitir al usuario elegir varios valores. El elemento gráfico se reproduce como una etiqueta `select` múltiple o una serie de casillas de verificación dependiendo de la opción `expanded`:

```
$generador->add('availability', 'choice', array(
    'choices'   => array(
        'morning'   => 'Mañana',
        'afternoon' => 'Tarde',
        'evening'   => 'Noche',
    ),
    'multiple'  => true,
));
```

También puedes utilizar la opción `choice_list`, la cual toma un objeto que puede especificar las opciones para el elemento gráfico.

Etiqueta select, casillas de verificación o botones de radio

Este campo se puede reproducir como uno de varios campos *HTML*, dependiendo de las opciones `expanded` y `multiple`:

tipo elemento	expandido	múltiple
etiqueta de selección	false	false
etiqueta de selección (con atributo `multiple`)	false	true
botones de radio	true	false
caja de verificación (checkboxes)	true	true

Opciones del campo

choices

tipo: `array` **predeterminado**: `array()`

Esta es la forma más sencilla de especificar las opciones que debe utilizar por este campo. La opción `choices` es una matriz, donde la clave del arreglo es el valor del elemento y el valor del arreglo es la etiqueta del elemento:

```
$generador->add('genero', 'choice', array(
    'choices' => array('m' => 'Masculino', 'f' => 'Femenino')
));
```

choice_list

tipo: `Symfony\Component\Form\Extension\Core\ChoiceList\ChoiceListInterface`

Esta es una manera de especificar las opciones que se utilizan para este campo. La opción `choice_list` debe ser una instancia de `ChoiceListInterface`. Para casos más avanzados, puedes crear una clase personalizada que implemente la interfaz para suplir las opciones.

multiple

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `-------------------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

```
$generador->add('estados', 'choice', array(
    'empty_value' => 'Elige una opción',
));
```

- Garantiza que ninguna opción con valor `vacío` se muestre:

```
$generador->add('estados', 'choice', array(
    'empty_value' => false,
));
```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.4 Recolectando tipos de campo

Consulta la `Symfony\Component\Form\Extension\Core\Type\CollectionType`.

4.9.5 Tipo de campo country

El tipo `country` es un subconjunto de `ChoiceType` que muestra los países del mundo. Como bono adicional, los nombres de los países se muestran en el idioma del usuario.

El "valor" para cada país es el código de país de dos letras.

Nota: La configuración regional del usuario se adivina usando Locale::getDefault()

A diferencia del tipo `choice`, no es necesario especificar una opción `choices` o `choice_list` como el tipo de campo utilizando automáticamente todos los países del mundo. *Puedes* especificar cualquiera de estas opciones manualmente, pero entonces sólo debes utilizar el tipo `choice` directamente.

Reproducido como	pueden ser varias etiquetas (consulta *Etiqueta select, casillas de verificación o botones de radio* (Página 463))
Opciones heredadas	▪ multiple (Página 467) ▪ expanded (Página 467) ▪ preferred_choices (Página 467) ▪ empty_value (Página 467) ▪ error_bubbling (Página 468) ▪ required (Página 468) ▪ label (Página 468) ▪ read_only (Página 468)
Tipo del padre	*choice* (Página 462)
Clase	`Symfony\Component\Form\Extension\Core\Type\Country`

Opciones heredadas

Estas opciones las hereda del tipo *choice* (Página 462):

multiple

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `-------------------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

  ```
  {{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
  ```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

  ```
  $generador->add('estados', 'choice', array(
      'empty_value' => 'Elige una opción',
  ));
  ```

- Garantiza que ninguna opción con valor `vacío` se muestre:

```
    $generador->add('estados', 'choice', array(
        'empty_value' => false,
    ));
```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

4.9.6 Tipo de campo `csrf`

El tipo `csrf` es un campo de entrada oculto que contiene un fragmento CSRF.

Reproducido como	campo `input hidden`
Opciones	▪ `csrf_provider` ▪ `page_id` ▪ `property_path`
Tipo del padre	`hidden`
Clase	`Symfony\Component\Form\Extension\Csrf\Type\CsrfTyp`

Opciones del campo

`csrf_provider`

tipo: `Symfony\Component\Form\CsrfProvider\CsrfProviderInterface`

El objeto `CsrfProviderInterface` que debe generar la ficha CSRF. Si no se establece, el valor predeterminado es el proveedor predeterminado.

intención

tipo: `string`

Un opcional identificador único que se utiliza para generar la ficha CSRF.

- `property_path` [tipo: cualquiera, predeterminado: el valor del campo] Los campos muestran un valor de la propiedad del objeto dominio por omisión. Cuando se envía el formulario, el valor presentado se escribe de nuevo en el objeto.

  Si deseas reemplazar la propiedad que un campo lee y escribe, puedes establecer la opción `property_path`. Su valor predeterminado es el nombre del campo.

4.9.7 Tipo de campo `date`

Un campo que permite al usuario modificar información de fecha a través de una variedad de diferentes elementos *HTML*.

Los datos subyacentes utilizados para este tipo de campo pueden ser un objeto `DateTime`, una cadena, una marca de tiempo o una matriz. Siempre y cuando la opción input (Página 471) se configure correctamente, el campo se hará cargo de todos los detalles.

El campo se puede reproducir como un cuadro de texto, tres cuadros de texto (mes, día y año) o tres cuadros de selección (ve la opción widget (Página 470)).

Tipo de dato subyacente	puede ser `DateTime`, string, timestamp, o array (ve la opción `input`)
Reproducido cómo	un sólo campo de texto o tres campos de selección
Opciones	■ widget (Página 470) ■ input (Página 471) ■ empty_value (Página 471) ■ years (Página 471) ■ months (Página 471) ■ days (Página 471) ■ format (Página 472) ■ pattern (Página 472) ■ data_timezone (Página 472) ■ user_timezone (Página 472)
Tipo del padre	`field` (su es texto), `form` en cualquier otro caso
Clase	`Symfony\Component\Form\Extension\Core\Type\DateTyp`

Uso básico

Este tipo de campo es altamente configurable, pero fácil de usar. Las opciones más importantes son `input` y `widget`.

Supongamos que tienes un campo `publishedAt` cuya fecha subyacente es un objeto `DateTime`. El siguiente código configura el tipo `date` para ese campo como tres campos de opciones diferentes:

```
$generador->add('publishedAt', 'date', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

La opción `input` se *debe* cambiar para que coincida con el tipo de dato de la fecha subyacente. Por ejemplo, si los datos del campo `publishedAt` eran una marca de tiempo Unix, habría la necesidad de establecer `input` a `timestamp`:

```
$generador->add('publishedAt', 'date', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

El campo también es compatible con `array` y `string` como valores válidos de la opción `input`.

Opciones del campo

widget

tipo: `string` **predeterminado**: `choice`

La forma básica en que se debe reproducir este campo. Puede ser una de las siguientes:

- `choice`: reproduce tres entradas de selección. El orden de los selectores se define en la opción `pattern`.
- `text`: reproduce tres campos de entrada tipo texto (mes, día, año).
- `single_text`: reproduce un sólo cuadro de tipo texto. La entrada del usuario se valida en base a la opción `format`.

input

tipo: `string` **predeterminado**: `datetime`

El formato del dato *input* - es decir, el formato de la fecha en que se almacena en el objeto subyacente. Los valores válidos son los siguientes:

- `string` (por ejemplo `2011-06-05`)
- `datetime` (un objeto `DateTime`)
- `array` (por ejemplo `array('anio' => 2011, 'mes' => 06, 'dia' => 05)`)
- `timestamp` (por ejemplo `1307232000`)

El valor devuelto por el formulario también se normaliza de nuevo a este formato.

empty_value

tipo: `string``|``array`

Si la opción de elemento gráfico se ajusta a `choice`, entonces este campo se reproduce como una serie de cajas de `selección`. La opción `empty_value` se puede utilizar para agregar una entrada "en blanco" en la parte superior de cada caja de selección:

```
$generador->add('fechaVencimiento', 'date', array(
    'empty_value' => '',
));
```

Alternativamente, puedes especificar una cadena que se mostrará en lugar del valor "en blanco":

```
$generador->add('fechaVencimiento', 'date', array(
    'empty_value' => array('year' => 'Año', 'month' => 'Mes', 'day' => 'Día')
));
```

years

tipo: `array` **predeterminado**: de cinco años antes a cinco años después del año actual

Lista de años disponibles para el tipo de campo `year`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

months

tipo: `array` **predeterminado**: entre 1 y 12

Lista de los meses disponibles para el tipo de campo `month`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

days

tipo: `array` **predeterminado**: entre 1 y 31

Lista de los días disponibles para el tipo de campo `day`. Esta opción sólo es relevante cuando la opción widget (Página 470) está establecida en `choice`:

```
'days' => range(1,31)
```

format

tipo: `integer` o `string` **predeterminado**: `IntlDateFormatter::MEDIUM`

Opción pasada a la clase `IntlDateFormatter`, utilizada para transformar la entrada del usuario al formato adecuado. Esto es crítico cuando la opción widget (Página 470) está establecida en `single_text`, y definirá la forma de transformar la entrada. De manera predeterminada, el formato es determinado basándose en la configuración regional del usuario actual; la puedes redefinir pasando el formato como una cadena.

pattern

tipo: `string`

Esta opción sólo es relevante cuando el widget (Página 470) se ajusta a `choice`. El patrón predeterminado está basado en la opción format (Página 472), y trata de coincidir con los caracteres `M`, `d`, e `y` en el patrón del formato. Si no hay coincidencia, el valor predeterminado es la cadena `{{ year }}-{{ month }}-{{ day }}`. Los elementos para esta opción son:

- `{{ year }}`: Reemplazado con el elemento gráfico `year`
- `{{ month }}`: Reemplazado con el elemento gráfico `month`
- `{{ day }}`: Reemplazado por el elemento gráfico `day`

data_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria en que se almacenan los datos entrantes. Esta debe ser una de las zonas horarias compatibles con PHP

user_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria para mostrar los datos al usuario (y por lo tanto también los datos que el usuario envía). Esta debe ser una de las zonas horarias compatibles con PHP

4.9.8 Tipo de campo `datetime`

Este tipo de campo permite al usuario modificar los datos que representan una fecha y hora específica (por ejemplo, `05/06/2011 12:15:30`).

Se pueden reproducir como una entrada de texto o etiquetas de selección. El formato subyacente de los datos puede ser un objeto `DateTime`, una cadena, una marca de tiempo o una matriz.

Tipo de dato subyacente	puede ser `DateTime`, string, timestamp, o array (ve la opción `input`)
Reproducido como	un solo campo de texto o tres cuadros de selección
Opciones	■ date_widget (Página 473) ■ time_widget (Página 473) ■ input (Página 473) ■ date_format (Página 473) ■ years (Página 474) ■ months (Página 474) ■ days (Página 474)
Tipo del padre	*form* (Página 483)
Clase	`Symfony\Component\Form\Extension\Core\Type\DateTim`

Opciones del campo

date_widget

tipo: `string` **predeterminado**: `choice`

Define la opción `widget` para el tipo *date* (Página 469)

time_widget

tipo: `string` **predeterminado**: `choice`

Define la opción `widget` para el tipo *time* (Página 505)

input

tipo: `string` **predeterminado**: `datetime`

El formato del dato *input* - es decir, el formato de la fecha en que se almacena en el objeto subyacente. Los valores válidos son los siguientes:

- `string` (p. ej. `2011-06-05 12:15:00`)
- `datetime` (un objeto `DateTime`)
- `array` (p. ej. `array(2011, 06, 05, 12, 15, 0)`)
- `timestamp` (p. ej. `1307276100`)

El valor devuelto por el formulario también se normaliza de nuevo a este formato.

date_format

tipo: `integer` o `string` **predeterminado**: `IntlDateFormatter::MEDIUM`

Define la opción `format` que se transmite al campo de fecha.

hours

tipo: `integer` **predeterminado**: entre 1 y 23

Lista de las horas disponibles para el tipo de campo `hours`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

minutes

tipo: `integer` **predeterminado**: entre 1 y 59

Lista de los minutos disponibles para el tipo de campo `minutes`. Esta opción sólo es relevante cuando la opción **'widget'_** está establecida en `choice`.

seconds

tipo: `integer` **predeterminado**: entre 1 y 59

Lista de los segundos disponibles para el tipo de campo segundos. Esta opción sólo es relevante cuando la opción **'widget'_** está establecida en `choice`.

years

tipo: `array` **predeterminado**: de cinco años antes a cinco años después del año actual

Lista de años disponibles para el tipo de campo `year`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

months

tipo: `array` **predeterminado**: entre 1 y 12

Lista de los meses disponibles para el tipo de campo `month`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

days

tipo: `array` **predeterminado**: entre 1 y 31

Lista de los días disponibles para el tipo de campo `day`. Esta opción sólo es relevante cuando la opción **'widget'_** está establecida en `choice`:

```
'days' => range(1,31)
```

with_seconds

tipo: `Boolean` **predeterminado**: `false`

Si debe o no incluir los segundos en la entrada. Esto resultará en una entrada adicional para capturar los segundos.

data_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria en que se almacenan los datos entrantes. Esta debe ser una de las zonas horarias compatibles con PHP

user_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria para mostrar los datos al usuario (y por lo tanto también los datos que el usuario envía). Esta debe ser una de las zonas horarias compatibles con PHP

4.9.9 Tipo de campo `email`

El campo `email` es un campo de texto que se reproduce usando etiquetas `HTML5 <input type=\"email\" />`.

Reproducido como	campo `input email` (un cuadro de texto)
Opciones heredadas	▪ max_length (Página 475) ▪ required (Página 475) ▪ label (Página 476) ▪ trim (Página 476) ▪ read_only (Página 476) ▪ error_bubbling (Página 476)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\EmailTy`

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.10 Tipo de campo `entity`

Un campo `choice` especial que está diseñado para cargar las opciones de una entidad *Doctrine*. Por ejemplo, si tiene una entidad `Categoria`, puedes utilizar este campo para mostrar un campo `select` todos o algunos de los objetos `Categoria` de la base de datos.

Reproducido como	pueden ser varias etiquetas (consulta *Etiqueta select, casillas de verificación o botones de radio* (Página 463))
Opciones	■ class (Página 478) ■ property (Página 478) ■ query_builder (Página 478) ■ em (Página 478)
Opciones heredadas	■ required (Página 480) ■ label (Página 480) ■ multiple (Página 478) ■ expanded (Página 479) ■ preferred_choices (Página 479) ■ empty_value (Página 479) ■ read_only (Página 480) ■ error_bubbling (Página 480)
Tipo del padre	*choice* (Página 462)
Clase	`Symfony\Bridge\Doctrine\Form\Type\EntityType`

Uso básico

El tipo `entidad` tiene una sola opción obligatoria: la entidad que debe aparecer dentro del campo de elección:

```
$generador->add('users', 'entity', array(
    'class' => 'Acme\\HolaBundle\\Entity\\User',
));
```

En este caso, todos los objetos `Usuario` serán cargados desde la base de datos y representados como etiquetas `select`, botones de radio o una serie de casillas de verificación (esto depende del valor `multiple` y `expanded`).

Usando una consulta personalizada para las Entidades

Si es necesario especificar una consulta personalizada a utilizar al recuperar las entidades (por ejemplo, sólo deseas devolver algunas entidades, o necesitas ordenarlas), utiliza la opción `query_builder`. La forma más fácil para utilizar la opción es la siguiente:

```
use Doctrine\ORM\EntityRepository;
// ...

$generador->add('users', 'entity', array(
    'class' => 'Acme\\HolaBundle\\Entity\\User',
    'query_builder' => function(EntityRepository $er) {
        return $er->createQueryBuilder('u')
            ->orderBy('u.nombreusuario', 'ASC');
    },
));
```

Etiqueta select, casillas de verificación o botones de radio

Este campo se puede reproducir como uno de varios campos *HTML*, dependiendo de las opciones `expanded` y `multiple`:

tipo elemento	expandido	múltiple
etiqueta de selección	false	false
etiqueta de selección (con atributo `multiple`)	false	true
botones de radio	true	false
caja de verificación (checkboxes)	true	true

Opciones del campo

`class`

tipo: `string` **requerido**

La clase de tu entidad (por ejemplo, `Acme\GuardaBundle\Entity\Categoria`).

`property`

tipo: `string`

Esta es la propiedad que se debe utilizar para visualizar las entidades como texto en el elemento *HTML*. Si la dejas en blanco, el objeto entidad será convertido en una cadena y por lo tanto debe tener un método `__toString()`.

`query_builder`

tipo: `Doctrine\ORM\QueryBuilder` o un Cierre

Si lo especificas, se utiliza para consultar el subconjunto de opciones (y su orden) que se debe utilizar para el campo. El valor de esta opción puede ser un objeto `QueryBuilder` o un Cierre. Si utilizas un Cierre, este debe tener un sólo argumento, el cual es el `EntityRepository` de la entidad.

`em`

tipo: `string` **predeterminado**: el gestor de la entidad

Si lo especificas, el gestor de la entidad especificada se utiliza para cargar las opciones en lugar de las predeterminadas del gestor de la entidad.

Opciones heredadas

Estas opciones las hereda del tipo *choice* (Página 462):

`multiple`

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `-------------------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

  ```
  {{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
  ```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

  ```
  $generador->add('estados', 'choice', array(
      'empty_value' => 'Elige una opción',
  ));
  ```

- Garantiza que ninguna opción con valor `vacío` se muestre:

  ```
  $generador->add('estados', 'choice', array(
      'empty_value' => false,
  ));
  ```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.11 Tipo de campo `file`

El tipo `file` representa una entrada de archivo en tu formulario.

Reproducido como	campo `input file`
Opciones heredadas	▪ required (Página 482) ▪ label (Página 482) ▪ read_only (Página 482) ▪ error_bubbling (Página 482)
Typo del padre	*form* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\FileTyp`

Uso básico

Digamos que tienes esta definición de formulario:

```
$generador->add('attachment', 'file');
```

Prudencia: No olvides añadir el atributo `enctype` en la etiqueta del formulario: `<form action="#" method="post" {{ form_enctype(form) }}>`.

Cuando se envía el formulario, el campo de datos adjuntos (`attchment`) será una instancia de `Symfony\Component\HttpFoundation\File\UploadedFile`. Este lo puedes utilizar para mover el archivo `adjunto` a una ubicación permanente:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

public function uploadAction()
{
    // ...

    if ($formulario->isValid()) {
        $algunNuevoNombreDeArchivo = ...

        $formulario['attachment']->getData()->move($dir, $algunNuevoNombreDeArchivo);

        // ...
    }

    // ...
}
```

El método `move()` toma un directorio y un nombre de archivo como argumentos. Puedes calcular el nombre de archivo en una de las siguientes formas:

```
// usa el nombre de archivo original
$file->move($dir, $file->getClientOriginalName());

// calcula un nombre aleatorio e intenta adivinar la extensión (más seguro)
$extension = $file->guessExtension();
if (!$extension) {
    // no puede adivinar la extensión
    $extension = 'bin';
}
$file->move($dir, rand(1, 99999).'.'.$extension);
```

Usar el nombre original a través de `getClientOriginalName()` no es seguro, ya que el usuario final lo podría haber manipulado. Además, puede contener caracteres que no están permitidos en nombres de archivo. Antes de usar el nombre directamente, lo debes desinfectar.

Lee en el recetario el *ejemplo* (Página 294) de cómo manejar la carga de archivos adjuntos con una entidad *Doctrine*.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.12 El tipo abstracto `field`

El tipo `field` de formulario no es un tipo de campo real que utilices, sino que funciona como el tipo de campo padre de muchos otros campos.

El tipo `field` predefine un par de opciones:

data

tipo: mixed **predeterminado**: De manera predeterminada al campo del objeto subyacente (si existe)

Cuando creas un formulario, cada campo inicialmente muestra el valor de la propiedad correspondiente al objeto del dominio del formulario (si está ligado un objeto al formulario). Si sólo deseas cambiar el valor inicial para el formulario y campo individual, lo puedes establecer en la opción de `data`:

```
$generador->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

- `disabled` [tipo: Boolean, predeterminado: false] Si no deseas que el usuario pueda modificar el valor de un campo, puedes establecer la opción `disabled` en `true`. Cualquier valor presentado será omitido.

```
use Symfony\Component\Form\TextField

$field = new TextField('status', array(
    'data' => 'Old data',
    'disabled' => true,
));
$field->submit('New data');

// imprime el "dato antiguo"
echo $field->getData();
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

- `property_path` [tipo: cualquiera, predeterminado: el valor del campo] Los campos muestran un valor de la propiedad del objeto dominio por omisión. Cuando se envía el formulario, el valor presentado se escribe de nuevo en el objeto.

  Si deseas reemplazar la propiedad que un campo lee y escribe, puedes establecer la opción `property_path`. Su valor predeterminado es el nombre del campo.

4.9.13 Tipo de campo `Form`

Consulta `Symfony\Component\Form\Extension\Core\Type\FormType`.

4.9.14 Tipo de campo `hidden`

El tipo `hidden` representa un campo de entrada oculto.

Reproducido como	campo `input hidden`
Tipo del padre	`field`
Clase	`Symfony\Component\Form\Extension\Core\Type\HiddenType`

4.9.15 Tipo de campo `integer`

Reproduce un campo de entrada para "número". Básicamente, se trata de un campo de texto que es bueno manejando datos enteros en un formulario. El campo de entrada `number` se parece a un cuadro de texto, salvo que - si el navegador

del usuario es compatible con HTML5 - tendrá algunas funciones de interfaz adicionales.

Este campo tiene diferentes opciones sobre cómo manejar los valores de entrada que no son enteros. Por omisión, todos los valores no enteros (por ejemplo 6.78) se redondearán hacia abajo (por ejemplo, 6).

Reproducido como	campo `input text`
Opciones	■ rounding_mode (Página 484) ■ grouping (Página 484)
Opciones heredadas	■ required (Página 484) ■ label (Página 485) ■ read_only (Página 485) ■ error_bubbling (Página 485)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\Integer`

Opciones del campo

rounding_mode

tipo: `integer` **predeterminado**: `IntegerToLocalizedStringTransformer::ROUND_DOWN`

Por omisión, si el usuario introduce un número no entero, se redondeará hacia abajo. Hay varios métodos de redondeo, y cada uno es una constante en la en la clase `Symfony\Component\Form\Extension\Core\DataTransformer\IntegerToLocalizedStringTransformer`:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` modo de redondeo para redondear hacia cero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` modo de redondeo para redondear hacia el infinito negativo.
- `IntegerToLocalizedStringTransformer::ROUND_UP` modo de redondeo para redondear alejándose del cero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` modo de redondeo para redondear hacia el infinito positivo.

grouping

tipo: `integer` **predeterminado**: `false`

El valor se establece como el atributo `NumberFormatter::GROUPING_USED` cuando se utiliza la clase *PHP* `NumberFormatter`.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.16 Tipo de campo language

El tipo `lenguaje` es un subconjunto de `ChoiceType` que permite al usuario seleccionar entre una larga lista de idiomas. Como bono adicional, los nombres de idioma se muestran en el idioma del usuario.

El "valor" de cada localidad es o bien el código de dos letras ISO639-1 *idioma* (por ejemplo, `es`).

Nota: La configuración regional del usuario se adivina usando Locale::getDefault()

A diferencia del tipo `choice`, no es necesario especificar una opción `choice` o `choice_list` ya que el tipo de campo automáticamente utiliza una larga lista de idiomas. *Puedes* especificar cualquiera de estas opciones manualmente, pero entonces sólo debes utilizar el tipo `choice` directamente.

Reproducido como	pueden ser varias etiquetas (consulta *Etiqueta select, casillas de verificación o botones de radio* (Página 463))
Opciones heredadas	■ multiple (Página 486) ■ expanded (Página 486) ■ preferred_choices (Página 486) ■ empty_value (Página 487) ■ error_bubbling (Página 487) ■ required (Página 487) ■ label (Página 488) ■ read_only (Página 488)
Tipo del padre	*choice* (Página 462)
Clase	`Symfony\Component\Form\Extension\Core\Type\Languag`

Opciones heredadas

Estas opciones las hereda del tipo *choice* (Página 462):

multiple

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `----------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

```
$generador->add('estados', 'choice', array(
    'empty_value' => 'Elige una opción',
));
```

- Garantiza que ninguna opción con valor `vacío` se muestre:

```
$generador->add('estados', 'choice', array(
    'empty_value' => false,
));
```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

4.9.17 Tipo de campo locale

El tipo `locale` es un subconjunto de `ChoiceType` que permite al usuario seleccionar entre una larga lista de regiones (idioma + país). Como bono adicional, los nombres regionales se muestran en el idioma del usuario.

El "valor" de cada región es o bien el del código de *idioma* ISO639-1 de dos letras (por ejemplo, `es`), o el código de idioma seguido de un guión bajo (_), luego el código de *país* ISO3166 (por ejemplo, `es_ES` para el Español/España).

Nota: La configuración regional del usuario se adivina usando Locale::getDefault()

A diferencia del tipo `choice`, no es necesario especificar una opción `choices` o `choice_list`, ya que el tipo de campo utiliza automáticamente una larga lista de regiones. *Puedes* especificar cualquiera de estas opciones manualmente, pero entonces sólo debes utilizar el tipo `choice` directamente.

Reproducido como	pueden ser varias etiquetas (consulta *Etiqueta select, casillas de verificación o botones de radio* (Página 463))
Opciones heredadas	▪ multiple (Página 488) ▪ expanded (Página 489) ▪ preferred_choices (Página 489) ▪ empty_value (Página 489) ▪ error_bubbling (Página 490) ▪ required (Página 490) ▪ label (Página 490) ▪ read_only (Página 490)
Tipo del padre	*choice* (Página 462)
Clase	`Symfony\Component\Form\Extension\Core\Type\Languag`

Opciones heredadas

Estas opciones las hereda del tipo *choice* (Página 462):

multiple

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `----------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

  ```
  {{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
  ```

- *PHP*

  ```
  <?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
  ```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

  ```
  $generador->add('estados', 'choice', array(
      'empty_value' => 'Elige una opción',
  ));
  ```

- Garantiza que ninguna opción con valor `vacío` se muestre:

  ```
  $generador->add('estados', 'choice', array(
      'empty_value' => false,
  ));
  ```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

4.9.18 Tipo de campo `money`

Reproduce un campo de entrada de texto especializado en el manejo de la presentación de datos tipo "moneda".

Este tipo de campo te permite especificar una moneda, cuyo símbolo se representa al lado del campo de texto. También hay otras opciones para personalizar la forma de la entrada y salida de los datos manipulados.

Reproducido como	campo `input text`
Opciones	■ currency (Página 491) ■ divisor (Página 491) ■ precision (Página 491) ■ grouping (Página 491)
Opciones heredadas	■ required (Página 492) ■ label (Página 492) ■ read_only (Página 492) ■ error_bubbling (Página 492)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\MoneyTy`

Opciones del campo

currency

tipo: `string` **predeterminado**: `EUR`

Especifica la moneda en la cual se especifica el dinero. Esta determina el símbolo de moneda que se debe mostrar en el cuadro de texto. Dependiendo de la moneda - el símbolo de moneda se puede mostrar antes o después del campo de entrada de texto.

También lo puedes establecer a `false` para ocultar el símbolo de moneda.

divisor

tipo: `integer` **predeterminado**: `1`

Si, por alguna razón, tienes que dividir tu valor inicial por un número antes de reproducirlo para el usuario, puedes utilizar la opción `divisor`. Por ejemplo:

```
$generador->add('precio', 'money', array(
    'divisor' => 100,
));
```

En este caso, si el campo `precio` está establecido en `9900`, entonces en realidad al usuario se le presentará el valor `99`. Cuando el usuario envía el valor `99`, este se multiplicará por `100` y finalmente se devolverá `9900` a tu objeto.

precision

tipo: `integer` **predeterminado**: `2`

Por alguna razón, si necesitas alguna precisión que no sean dos decimales, puedes modificar este valor. Probablemente no necesitarás hacer esto a menos que, por ejemplo, desees redondear al dólar más cercano (ajustando la precisión a `0`).

grouping

tipo: `integer` **predeterminado**: `false`

El valor se establece como el atributo `NumberFormatter::GROUPING_USED` cuando se utiliza la clase *PHP* `NumberFormatter`.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.19 Tipo de campo number

Reproduce un campo de entrada de texto y se especializa en el manejo de entradas numéricas. Este tipo ofrece diferentes opciones para precisión, redondeo y agrupamiento que desees utilizar para tu número.

Reproducido como	campo `input text`
Opciones	■ rounding_mode (Página 493) ■ precision (Página 493) ■ grouping (Página 494)
Opciones heredadas	■ required (Página 494) ■ label (Página 494) ■ read_only (Página 494) ■ error_bubbling (Página 494)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\NumberT`

Opciones del campo

precision

tipo: `integer` **predeterminado**: Específico a la región (usualmente alrededor de `3`)

Este especifica cuantos decimales se permitirán para redondear el campo al valor presentado (a través de `rounding_mode`). Por ejemplo, si `precision` se establece en `2`, un valor presentado de `20.123` se redondeará a, por ejemplo, `20.12` (dependiendo de tu `rounding_mode`).

rounding_mode

tipo: `integer` **predeterminado**: `IntegerToLocalizedStringTransformer::ROUND_DOWN`

Si es necesario redondear un número presentado (basándonos en la opción `precision`), tienes varias opciones configurables para el redondeo. Cada opción es una constante en la clase `Symfony\Component\Form\Extension\Core\DataTransformer\IntegerToLocalizedStringTransformer`:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` modo de redondeo para redondear hacia cero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` modo de redondeo para redondear hacia el infinito negativo.
- `IntegerToLocalizedStringTransformer::ROUND_UP` modo de redondeo para redondear alejándose del cero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` modo de redondeo para redondear hacia el infinito positivo.
- `IntegerToLocalizedStringTransformer::ROUND_HALFDOWN` El modo de redondeo para redondear hacia "el vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso se redondea hacia abajo.
- `IntegerToLocalizedStringTransformer::ROUND_HALFEVEN` El modo de redondeo para redondear hacia "el vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso, se redondea hacia el vecino par.
- `IntegerToLocalizedStringTransformer::ROUND_HALFUP` El modo de redondeo para redondear hacia "el vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso se redondea hacia arriba.

grouping

tipo: `integer` **predeterminado**: `false`

El valor se establece como el atributo `NumberFormatter::GROUPING_USED` cuando se utiliza la clase *PHP* `NumberFormatter`.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.20 Tipo de campo password

El tipo de campo `password` reproduce un campo de texto para entrada de contraseñas.

Reproducido como	campo `input password`
Opciones	▪ always_empty (Página 495)
Opciones heredadas	▪ max_length (Página 495) ▪ required (Página 495) ▪ label (Página 495) ▪ trim (Página 496) ▪ read_only (Página 496) ▪ error_bubbling (Página 496)
Tipo del padre	*text* (Página 502)
Clase	`Symfony\Component\Form\Extension\Core\Type\Passwor`

Opciones del campo

always_empty

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el campo *siempre* se reproduce en blanco, incluso si el campo correspondiente tiene un valor. Cuando se establece en `false`, el campo de la contraseña se reproduce con el atributo `value` fijado a su valor real.

En pocas palabras, si por alguna razón deseas reproducir tu campo de contraseña *con* el valor de contraseña ingresado anteriormente en el cuadro, ponlo a `false`.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.21 Tipo de campo percent

El tipo de campo `percent` reproduce un campo de entrada de texto, especializado en el manejo de datos porcentuales. Si los datos porcentuales se almacenan como un decimal (por ejemplo, `0.95`), puedes utilizar este campo fuera de la caja. Si almacenas tus datos como un número (por ejemplo, `95`), debes establecer la opción `type` a `integer`.

Este campo añade un signo de porcentaje "`%`" después del cuadro de entrada.

Reproducido como	campo `input text`
Opciones	■ type (Página 496) ■ precision (Página 497)
Opciones heredadas	■ required (Página 497) ■ label (Página 497) ■ read_only (Página 497) ■ error_bubbling (Página 497)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\Percent`

Opciones

type

tipo: `string` **predeterminado**: `fractional`

Esto controla la forma en que tus datos están almacenados en el objeto. Por ejemplo, un porcentaje correspondiente al "55 %", lo puedes almacenar en el objeto como `0.55` o `55`. Los dos "tipos" manejan estos dos casos:

- `fractional` Si los datos se almacenan como un decimal (por ejemplo, `0.55`), usa este tipo. Los datos se multiplicarán por `100` antes de mostrarlos al usuario (por ejemplo, `55`). Los datos presentados se dividirán por `100` al presentar el formulario para almacenar el valor decimal (`0.55`);
- `integer` Si almacenas tus datos como un entero (por ejemplo, 55), entonces, utiliza esta opción. El valor crudo (`55`) se muestra al usuario y se almacena en tu objeto. Ten en cuenta que esto sólo funciona para valores enteros.

precision

tipo: `integer` **predeterminado**: `0`

Por omisión, los números ingresado se redondean. Para tener en cuenta más cifras decimales, utiliza esta opción.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.22 Tipo de campo radio

Crea un solo botón de radio. Este se debe utilizar siempre para un campo que tiene un valor booleano: si el botón de radio es seleccionado, el campo se establece en `true`, si el botón no está seleccionado, el valor se establece en `false`.

El tipo `radio` no suele usarse directamente. Comúnmente se utiliza internamente por otros tipos, tales como *choice* (Página 462). Si quieres tener un campo booleano, utiliza una *casilla de verificación* (Página 461).

Reproducido como	campo `input text`
Opciones	▪ value (Página 498)
Opciones heredadas	▪ required (Página 498) ▪ label (Página 498) ▪ read_only (Página 499) ▪ error_bubbling (Página 499)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\RadioTy`

Opciones del campo

value

tipo: `mixed` **predeterminado**: `1`

El valor utilizado realmente como valor para el botón de radio. Esto no afecta al valor establecido en tu objeto.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.23 Tipo de campo `repeated`

Este es un campo "grupo" especial, el cual crea dos campos idénticos, cuyos valores deben coincidir (o lanza un error de validación). Se utiliza comúnmente cuando necesitas que el usuario repita su contraseña o correo electrónico para verificar su exactitud.

Reproducido cómo	campo `input text` por omisión, pero ve la opción type (Página 500)
Opciones	▪ type (Página 500) ▪ options (Página 500) ▪ first_name (Página 500) ▪ second_name (Página 500)
Opciones heredadas	▪ invalid_message (Página 500) ▪ invalid_message_parameters (Página 501) ▪ error_bubbling (Página 501)
Tipo del padre	*field* (Página 483)
Clase	`Symfony\Component\Form\Extension\Core\Type\Repeate`

Ejemplo de uso

```
$generador->add('password', 'repeated', array(
    'type' => 'password',
    'invalid_message' => 'Los campos de contraseña deben coincidir.',
    'options' => array('label' => 'Contraseña'),
));
```

Al presentar satisfactoriamente un formulario, el valor ingresado en ambos campos "contraseña" se convierte en los datos de la clave `password`. En otras palabras, a pesar de que ambos campos efectivamente son reproducidos, el dato final del formulario sólo es el único valor que necesitas (generalmente una cadena).

La opción más importante es `type`, el cual puede ser cualquier tipo de campo y determina el tipo real de los dos campos subyacentes. La opción `options` se pasa a cada uno de los campos individuales, lo cual significa - en este ejemplo - que cualquier opción compatible con el tipo `password` la puedes pasar en esta matriz.

Validando

Una de las características clave del campo `repeated` es la validación interna (sin necesidad de hacer nada para configurar esto) el cual obliga a que los dos campos tengan un valor coincidente. Si los dos campos no coinciden, se mostrará un error al usuario.

El `invalid_message` se utiliza para personalizar el error que se mostrará cuando los dos campos no coinciden entre sí.

Opciones del campo

type

tipo: `string` **predeterminado**: `text`

Los dos campos subyacentes serán de este tipo de campo. Por ejemplo, pasando un tipo de `password` reproducirá dos campos de contraseña.

options

tipo: `array` **predeterminado**: `array()`

Esta matriz de opciones se pasará a cada uno de los dos campos subyacentes. En otras palabras, estas son las opciones que personalizan los tipos de campo individualmente. Por ejemplo, si la opción `typo` se establece en `password`, esta matriz puede contener las opciones `always_empty` o `required` - ambas opciones son compatibles con el tipo de campo `password`.

first_name

tipo: `string` **predeterminado**: `first`

Este es el nombre real del campo que se utilizará para el primer campo. Esto sobre todo no tiene sentido, sin embargo, puesto que los datos reales especificados en ambos campos disponibles bajo la clave asignada al campo `repeated` en sí mismo (por ejemplo, `password`). Sin embargo, si no se especificas una etiqueta, el nombre de este campo se utiliza para "adivinar" la etiqueta por ti.

second_name

tipo: `string` **predeterminado**: `second`

Al igual que `nombre_de_pila`, pero para el segundo campo.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

invalid_message

tipo: `string` **predeterminado**: `This value is not valid` (`Este valor no es válido`)

Este es el mensaje de error de validación utilizado cuando se determinan los datos ingresados mediante la validación interna de un tipo de campo. Esto puede suceder, por ejemplo, si el usuario ingresa una cadena en un campo *time* (Página 505) el cual no la puede convertir en una hora real. Para los mensajes de validación normales (como cuando se establece una longitud mínima en un campo), fija los mensajes de validación con tus reglas de validación (*referencia* (Página 153)).

invalid_message_parameters

tipo: `array` **predeterminado**: `array()`

Al establecer la opción `invalid_message`, posiblemente sea necesario que incluyas algunas variables en la cadena. Esto se puede lograr agregando marcadores de posición y variables en esa opción:

```
$generador->add('algún_campo', 'algún_tip', array(
    // ...
    'invalid_message'            => 'Introdujiste un valor no válido - este debe incluir %num% letras
    'invalid_message_parameters' => array('%num%' => 6),
));
```

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.24 Tipo de campo search

Esto reproduce un campo `<input type="search" />`, el cual es un cuadro de texto con funcionalidad especial apoyada por algunos navegadores.

Lee sobre el campo de de entrada búsqueda en DiveIntoHTML5.org

Reproducido como	campo `input search`
Opciones heredadas	▪ max_length (Página 501) ▪ required (Página 502) ▪ label (Página 502) ▪ trim (Página 502) ▪ read_only (Página 502) ▪ error_bubbling (Página 502)
Tipo del padre	*text* (Página 502)
Clase	`Symfony\Component\Form\Extension\Core\Type\SearchT`

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.25 Tipo de campo text

El campo de texto reproduce el campo de entrada de texto más básico.

Reproducido como	campo `input text`
Opciones heredadas	■ max_length (Página 503) ■ required (Página 503) ■ label (Página 503) ■ trim (Página 503) ■ read_only (Página 504) ■ error_bubbling (Página 504)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\TextTyp`

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.26 Tipo de campo `textarea`

Reproduce un elemento `textarea` *HTML*.

Reproducido como	etiqueta `textarea`
Opciones heredadas	▪ max_length (Página 504) ▪ required (Página 504) ▪ label (Página 505) ▪ trim (Página 505) ▪ read_only (Página 505) ▪ error_bubbling (Página 505)
Tipo del padre	*field* (Página 482)
Clase	`Symfony\Component\Form\Extension\Core\Type\Textare`

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.27 Tipo de campo `time`

Un campo para capturar entradas de tiempo.

Este se puede reproducir como un campo de texto, una serie de campos de texto (por ejemplo, horas, minutos, segundos) o una serie de campos de selección. Los datos subyacentes se pueden almacenar como un objeto `DateTime`, una cadena, una marca de tiempo (`timestamp`) o una matriz.

Tipo de dato subyacente	puede ser `DateTime`, `string`, `timestamp`, o `array` (consulta la opción `input`)
Reproducido como	pueden ser varias etiquetas (ve abajo)
Opciones	■ widget (Página 506) ■ input (Página 506) ■ with_seconds (Página 507) ■ hours (Página 507) ■ minutes (Página 507) ■ seconds (Página 507) ■ data_timezone (Página 507) ■ user_timezone (Página 507)
Tipo del padre	`form`
Clase	`Symfony\Component\Form\Extension\Core\Type\TimeTyp`

Uso básico

Este tipo de campo es altamente configurable, pero fácil de usar. Las opciones más importantes son `input` y `widget`.

Supongamos que tienes un campo `startTime` cuyo dato de hora subyacente es un objeto `DateTime`. Lo siguiente configura el tipo `time` para ese campo como tres campos de opciones diferentes:

```
$generador->add('startTime', 'time', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

La opción `input` se *debe* cambiar para que coincida con el tipo de dato de la fecha subyacente. Por ejemplo, si los datos del campo `startTime` fueran una marca de tiempo Unix, habría necesidad de establecer la `entrada` a `timestamp`:

```
$generador->add('startTime', 'time', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

El campo también es compatible con `array` y `string` como valores válidos de la opción `input`.

Opciones del campo

widget

tipo: `string` **predeterminado**: `choice`

La forma básica en que se debe reproducir este campo. Puede ser una de las siguientes:

- `choice`: reproduce dos (o tres di with_seconds (Página 507) es `true`) cuadros de selección.
- `text`: reproduce dos o tres cuadros de texto (hora, minuto, segundo).
- `single_text`: reproduce un sólo cuadro de tipo texto. La entrada del usuario se validará contra la forma `hh:mm` (o `hh:mm:ss` si se utilizan segundos).

input

tipo: `string` **predeterminado**: `datetime`

El formato del dato *input* - es decir, el formato de la fecha en que se almacena en el objeto subyacente. Los valores válidos son los siguientes:

- `string` (por ejemplo `12:17:26`)
- `datetime` (un objeto `DateTime`)
- `array` (por ejemplo `array('hora' => 12, 'minuto' => 17, 'segundo' => 26)`)
- `timestamp` (por ejemplo `1307232000`)

El valor devuelto por el formulario también se normaliza de nuevo a este formato.

with_seconds

tipo: `Boolean` **predeterminado**: `false`

Si debe o no incluir los segundos en la entrada. Esto resultará en una entrada adicional para capturar los segundos.

hours

tipo: `integer` **predeterminado**: entre 1 y 23

Lista de las horas disponibles para el tipo de campo `hours`. Esta opción sólo es relevante cuando la opción `widget` está establecida en `choice`.

minutes

tipo: `integer` **predeterminado**: entre 1 y 59

Lista de los minutos disponibles para el tipo de campo `minutes`. Esta opción sólo es relevante cuando la opción widget (Página 506) está establecida en `choice`.

seconds

tipo: `integer` **predeterminado**: entre 1 y 59

Lista de los segundos disponibles para el tipo de campo segundos. Esta opción sólo es relevante cuando la opción widget (Página 506) está establecida en `choice`.

data_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria en que se almacenan los datos entrantes. Esta debe ser una de las zonas horarias compatibles con PHP

user_timezone

tipo: `string` **predeterminado**: la zona horaria del sistema

La zona horaria para mostrar los datos al usuario (y por lo tanto también los datos que el usuario envía). Esta debe ser una de las zonas horarias compatibles con PHP

4.9.28 Tipo de campo timezone

El tipo `timezone` es un subconjunto de `ChoiceType` que permite al usuario seleccionar entre todas las posibles zonas horarias.

El "value" para cada zona horaria es el nombre completo de la zona horaria, por ejemplo `América/Chicago` o `Europa/Estambul`.

A diferencia del tipo `choice`, no es necesario especificar una opción `choices` o `choice_list`, ya que el tipo de campo utiliza automáticamente una larga lista de regiones. *Puedes* especificar cualquiera de estas opciones manualmente, pero entonces sólo debes utilizar el tipo `choice` directamente.

Reproducido como	pueden ser varias etiquetas (consulta *Etiqueta select, casillas de verificación o botones de radio* (Página 463))
Opciones heredadas	■ multiple (Página 508) ■ expanded (Página 508) ■ preferred_choices (Página 508) ■ empty_value (Página 509) ■ error_bubbling (Página 510) ■ required (Página 509) ■ label (Página 509) ■ read_only (Página 510)
Tipo del padre	*choice* (Página 462)
Clase	`Symfony\Component\Form\Extension\Core\Type\Timezon`

Opciones heredadas

Estas opciones las hereda del tipo *choice* (Página 462):

multiple

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el usuario podrá seleccionar varias opciones (en contraposición a elegir sólo una opción). Dependiendo del valor de la opción `expanded`, esto reproducirá una etiqueta de selección o casillas de verificación si es `true` y una etiqueta de selección o botones de radio si es `false`. El valor devuelto será una matriz.

expanded

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los botones de radio o casillas de verificación se reproducirán (en función del valor de `multiple`). Si es `false`, se reproducirá un elemento de selección.

preferred_choices

tipo: `array` **predeterminado**: `array()`

Si se especifica esta opción, entonces un subconjunto de todas las opciones se trasladará a la parte superior del menú de selección. Lo siguiente debe mover la opción "Baz" a la parte superior, con una separación visual entre esta y el resto de las opciones:

```
$generador->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Ten en cuenta que las opciones preferidas sólo son útiles cuando se reproducen como un elemento `select` (es decir, `expanded` es `false`). Las opciones preferidas y las opciones normales están separadas visualmente por un conjunto de líneas punteadas (es decir, `----------`). Esto se puede personalizar cuando reproduzcas el campo:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($formulario['foo_choices'], array('separator' => '=====')) ?>
```

empty_value

tipo: `string` o `Boolean`

Esta opción determina si o no una opción especial `empty` (por ejemplo, "Elige una opción") aparecerá en la parte superior de un elemento gráfico de selección. Esta opción sólo se aplica si ambas opciones `expanded` y `multiple` se establecen en `false`.

- Añade un valor vacío con "Elige una opción", como el texto:

```
$generador->add('estados', 'choice', array(
    'empty_value' => 'Elige una opción',
));
```

- Garantiza que ninguna opción con valor `vacío` se muestre:

```
$generador->add('estados', 'choice', array(
    'empty_value' => false,
));
```

Si dejas sin establecer la opción `empty_value`, entonces automáticamente se añadirá una opción con espacio en blanco (sin texto) si y sólo si la opción `requiered` es `false`:

```
// añadirá una opción de espacio en blanco (sin texto)
$generador->add('estados', 'choice', array(
    'required' => false,
));
```

Estas opciones las hereda del tipo *field* (Página 482):

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

4.9.29 Tipo de campo url

El campo `url` es un campo de texto que prefija el valor presentado con un determinado protocolo (por ejemplo, `http://`) si el valor presentado no tiene ya un protocolo.

Reproducido como	campo `input url`
Opciones	▪ `default_protocol`
Opciones heredadas	▪ max_length (Página 510) ▪ required (Página 511) ▪ label (Página 511) ▪ trim (Página 511) ▪ read_only (Página 511) ▪ error_bubbling (Página 511)
Tipo del padre	*text* (Página 502)
Clase	`Symfony\Component\Form\Extension\Core\Type\UrlType`

Opciones del campo

default_protocol

tipo: `string` **predeterminado**: `http`

Si un valor es presentado que no comience con un protocolo (por ejemplo, `http://`, `ftp://`, etc.), se prefija la cadena con este protocolo al vincular los datos al formulario.

Opciones heredadas

Estas opciones las hereda del tipo *field* (Página 482):

max_length

tipo: `integer`

Esta opción se utiliza para añadir un atributo `max_length`, que algunos navegadores utilizan para limitar la cantidad de texto en un campo.

required

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, reproducirá un atributo required de HTML5. La `label` correspondiente será reproducida con una clase `required`.

Esto es superficial e independiente de la validación. A lo sumo, si dejas que *Symfony* adivine el tipo de campo, entonces el valor de esta opción, se puede adivinar a partir de tu información de validación.

label

tipo: `string` **predeterminado**: La etiqueta es "adivinada" a partir del nombre del campo

Establece la etiqueta que se utilizará al reproducir el campo. La etiqueta también se puede fijar directamente dentro de la plantilla:

```
{{ render_label(form.nombre, 'Tu nombre') }}
```

trim

tipo: `Boolean` **predeterminado**: `true`

Si es `true`, el espacio en blanco de la cadena presentada será eliminado a través de la función `trim()` cuando se vinculan los datos. Esto garantiza que si un valor es presentado con espacios en blanco excedentes, estos serán removidos antes de fusionar de nuevo el valor con el objeto subyacente.

read_only

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, el campo se debe reproducir con el atributo `disabled` para que el campo no sea editable.

error_bubbling

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, los errores de este campo serán pasados al campo padre o al formulario. Por ejemplo, si estableces en `true` un campo normal, cualquier error de ese campo se adjuntará al formulario principal, no al campo específico.

Un formulario se compone de *campos*, cada uno de los cuales se construye con la ayuda de un *tipo* de campo (por ejemplo, un tipo `text`, tipo `choices`, etc.) *Symfony2* viene con una larga lista de tipos de campo que puedes utilizar en tu aplicación.

4.9.30 Tipos de campo admitidos

Los siguientes tipos de campo están disponibles de forma nativa en *Symfony2*:

Campos de texto

- *text* (Página 502)
- *textarea* (Página 504)

- *email* (Página 475)
- *integer* (Página 483)
- *money* (Página 490)
- *number* (Página 492)
- *password* (Página 494)
- *percent* (Página 496)
- *search* (Página 501)
- *url* (Página 510)

Campos de elección

- *choice* (Página 462)
- *entity* (Página 476)
- *country* (Página 466)
- *language* (Página 485)
- *locale* (Página 488)
- *timezone* (Página 507)

Campos de fecha y hora

- *date* (Página 469)
- *datetime* (Página 472)
- *time* (Página 505)
- *birthday* (Página 459)

Otros campos

- *checkbox* (Página 461)
- *file* (Página 480)
- *radio* (Página 498)

Campos de grupos

- *collection* (Página 466)
- *repeated* (Página 499)

Campos ocultos

- *hidden* (Página 483)
- *csrf* (Página 468)

Campos base

- *field* (Página 482)
- *form* (Página 483)

4.10 Referencia de funciones de formulario en plantillas *Twig*

Este manual de referencia cubre todas las posibles funciones *Twig* disponibles para reproducir formularios. Hay varias funciones diferentes disponibles, y cada una es responsable de representar una parte diferente de un formulario (por ejemplo, etiquetas, errores, elementos gráficos, etc.).

4.10.1 form_label(form.nombre, label, variables)

Reproduce la etiqueta para el campo dado. Si lo deseas, puedes pasar como segundo argumento la etiqueta específica que deseas mostrar.

```
{{ form_label(form.nombre) }}

{# Las dos siguientes sintaxis son equivalentes #}
{{ form_label(form.nombre, 'Tu nombre', { 'attr': {'class': 'foo'} }) }}
{{ form_label(form.nombre, null, { 'label': 'Tu nombre', 'attr': {'class': 'foo'} }) }}
```

4.10.2 form_errors(form.nombre)

Representa los errores para el campo dado.

```
{{ form_errors(form.nombre) }}

{# render any "global" errors #}
{{ form_errors(form) }}
```

4.10.3 form_widget(form.nombre, variables)

Representa el elemento gráfico *HTML* de un campo determinado. Si aplicas este a todo el formulario o a la colección de campos, reproducirá cada fila subyacente del formulario.

```
{# reproduce un elemento gráfico, pero le agrega una clase "foo" #}
{{ form_widget(form.nombre, { 'attr': {'class': 'foo'} }) }}
```

El segundo argumento de `form_widget` es un conjunto de variables. La variable más común es `attr`, que es una matriz de atributos *HTML* que puedes aplicar al elemento gráfico *HTML*. En algunos casos, ciertos tipos también tienen otras opciones relacionadas con la plantilla que les puedes pasar. Estas se explican en base a tipo por tipo.

4.10.4 form_row(form.nombre, variables)

Representa la "fila" (row) de un campo determinado, el cual es la combinación de la etiqueta del campo, los errores y el elemento gráfico.

```
{# reproduce una fila de un campo, pero añade una clase "foo" #}
{{ form_row(form.nombre, { 'label': 'foo' }) }}
```

El segundo argumento de `form_row` es un arreglo de variables. Las plantillas de *Symfony* sólo permite redefinir la etiqueta como muestra el ejemplo anterior.

4.10.5 form_rest(form, variables)

Esto reproduce todos los campos que aún no se han presentado en el formulario dado. Es buena idea tenerlo siempre en alguna parte dentro del formulario ya que debe representar los campos ocultos por ti y los campos que se te olvide representar (puesto que va a representar el campo para ti).

```
{{ form_rest(form) }}
```

4.10.6 form_enctype(form)

Si el formulario contiene al menos un campo de carga de archivos, esta reproducirá el atributo requerido `enctype=multipart/form-data"`. Siempre es una buena idea incluirlo en tu etiqueta de formulario:

```
<form action="{{ path('form_submit') }}" method="post" {{ form_enctype(form) }}>
```

4.11 Referencia de restricciones de validación

4.11.1 NotBlank

Valida que un valor no está en blanco, el cual fue definido como distinto a una cadena en blanco y no es igual a `null`. Para forzar que el valor no sea simplemente igual a `null`, consulta la restricción *NotNull* (Página 516).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 515)
Clase	`Symfony\Component\Validator\Constraints\NotBlank`
Validador	`Symfony\Component\Validator\Constraints\NotBlankVa`

Uso básico

Si te quieres asegurar de que la propiedad `nombreDePila` de una clase `autor` no está en blanco, puedes hacer lo siguiente:

- *YAML*

```
properties:
nombreDePila:
    - NotBlank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\NotBlank()
     */
```

```
    protected $nombreDePila;
}
```

Opciones

message

tipo: `string` **predeterminado**:`This value should not be blank` (`Este valor no debe estar en blanco`)

Este es el mensaje que se mostrará si el valor está en blanco.

4.11.2 Blank

Valida que un valor está en blanco, es definida como igual a una cadena en blanco o igual a `null`. Para forzar que un valor estrictamente sea igual a `null`, consulta la restricción *Null* (Página 517). Para forzar que el valor *no* esté en blanco, consulta *NotBlank* (Página 514).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 516)
Clase	`Symfony\Component\Validator\Constraints\Blank`
Validador	`Symfony\Component\Validator\Constraints\BlankValid`

Uso básico

Si, por alguna razón, deseas asegurarte de que la propiedad `nombreDePila` de una clase `Autor` está en blanco, puedes hacer lo siguiente:

- *YAML*

```
properties:
    nombreDePila:
        - Blank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Blank()
     */
    protected $nombreDePila;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value should be blank` (`Este valor debe estar en blanco`)

Este es el mensaje que se mostrará si el valor no está en blanco.

4.11.3 NotNull

Valida que el valor no es estrictamente igual a `null`. Para garantizar que el valor no está simplemente en blanco (no es una cadena en blanco), consulta la restricción *NotBlank* (Página 514).

Aplica a	*propiedad or método* (Página 155)
Opciones	▪ message (Página 516)
Clase	`Symfony\Component\Validator\Constraints\NotNull`
Validador	`Symfony\Component\Validator\Constraints\NotNullVal`

Uso básico

Si te quieres asegurar de que la propiedad `nombreDePila` de una clase `Autor` no es estrictamente igual a `null`, deberías comprobar:

- *YAML*

```
properties:
nombreDePila:
    - NotNull: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\NotNull()
     */
    protected $nombreDePila;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value should not be null` (`Este valor no debe ser null`)

Este es el mensaje que se mostrará, si el valor es `null`.

4.11.4 Null

Valida que un valor es exactamente igual a `null`. Para obligar a que una propiedad no es más que simplemente un valor en blanco (cadena en blanco o `null`), consulta la restricción *Blank* (Página 515). Para asegurarte de que una propiedad no es `null`, consulta la restricción *NotNull* (Página 516).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 517)
Clase	`Symfony\Component\Validator\Constraints\Null`
Validador	`Symfony\Component\Validator\Constraints\NullValida`

Uso básico

Si, por alguna razón, quisieras asegurarte de que la propiedad `nombreDePila de una clase ``Autor` es exactamente igual a `null`, podrías hacer lo siguiente:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        nombreDePila:
            - Null: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Null()
     */
    protected $nombreDePila;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value should be null` (`Este valor debe ser null`)

Este es el mensaje que se mostrará si el valor no es `null`.

4.11.5 True

Valida que un valor es `true`. En concreto, este comprueba si el valor es exactamente `true`, exactamente el número entero `1`, o exactamente la cadena "`1`".

Además consulta *False* (Página 519).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 519)
Clase	`Symfony\Component\Validator\Constraints\True`
Validador	`Symfony\Component\Validator\Constraints\TrueValida`

Uso básico

Puedes aplicar esta restricción a propiedades (por ejemplo, una propiedad `terminosAceptados` en un modelo de registro) o a un método `captador`. Esta es más potente en este último caso, donde puedes afirmar que un método devuelve un valor `true`. Por ejemplo, supongamos que tienes el siguiente método:

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

class Autor
{
    protected $muestra;

    public function isTokenValid()
    {
        return $this->muestra == $this->generateToken();
    }
}
```

A continuación, puedes limitar este método con `True`.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    getters:
        muestraValida:
            - "True": { message: "La muestra no es válida" }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    protected $muestra;

    /**
     * @Assert\True(message = "The token is invalid")
     */
    public function isTokenValid()
    {
        return $this->muestra == $this->generateToken();
    }
}
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- src/Acme/Blogbundle/Resources/config/validation.xml -->
```

```
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Autor">
        <getter property="muestraValida">
            <constraint name="True">
                <option name="message">La muestra no es válida...</option>
            </constraint>
        </getter>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\True;

class Autor
{
    protected $muestra;

    public static function cargaValidadorDeMetadatos(ClassMetadata $metadatos)
    {
        $metadatos->addGetterConstraint('muestraValida', new True(array(
            'message' => 'La muestra no es válida',
        )));
    }

    public function esValidaLaMuestra()
    {
        return $this->muestra == $this->generateToken();
    }
}
```

Si `esValidaLaMuestra()` devuelve falso, la validación fallará.

Opciones

message

tipo: `string` **predeterminado**: `This value should be true` (`Este valor debe ser true`)

Este mensaje se muestra si el dato subyacente no es `true`.

4.11.6 False

Valida que un valor es `false`. En concreto, esta comprueba si el valor es exactamente `false`, exactamente el número entero `0`, o exactamente la cadena "`0`".

Además consulta *True* (Página 517).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 521)
Clase	`Symfony\Component\Validator\Constraints\False`
Validador	`Symfony\Component\Validator\Constraints\FalseValid`

Uso básico

La restricción `False` se puede aplicar a una propiedad o a un método "captador", pero comúnmente, es más útil en este último caso. Por ejemplo, supongamos que quieres garantizar que una propiedad `estado` *no* está en una matriz dinámica de `invalidStates`. En primer lugar, crearías un método "captador":

```
protected $estado;

protectd $estadoNoValido = array();

public function esEstadoNoValido()
{
    return in_array($this->estado, $this->estadoNoValido);
}
```

En este caso, el objeto subyacente es válido sólo si el método `esEstadoNoValido` devuelve `false`:

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor
    getters:
        stateInvalid:
            - "False":
                message: Ingresaste un estado no válido.
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\False()
     */
     public function esEstadoNoValido($messsage = "Ingresaste un estado no válido.")
     {
        // ...
     }
}
```

Prudencia: Cuando utilices *YAML*, asegúrate de rodear `False` entre comillas (`"False"`) o de lo contrario *YAML* lo convertirá en un valor booleano.

Opciones

message

tipo: `string` **predeterminado**: `This value should be false` (`Este valor debe ser false`)

Este mensaje se muestra si el dato subyacente no es `false`.

4.11.7 Type

Valida que un valor es de un tipo de dato específico. Por ejemplo, si una variable debe ser un `array`, puedes utilizar esta restricción con la opción `type` para validarla.

Aplica a	*propiedad o métodp* (Página 155)
Opciones	▪ *type* (Página 521) ▪ message (Página 522)
Clase	`Symfony\Component\Validator\Constraints\Type`
Validador	`Symfony\Component\Validator\Constraints\TypeValida`

Uso básico

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        edad:
            - Type:
                type: integer
                message: El valor {{ valor }} no es un {{ tipo }} válido.
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
 namespace Acme\BlogBundle\Entity;

 use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Type(type="integer", message="El valor {{ valor }} no es un {{ tipo }} válido.")
     */
     protected $edad;
}
```

Opciones

type

tipo: `string` [*opción predeterminada* (Página 154)]

Esta opción requerida es el nombre de clase completo o uno de los tipos de datos *PHP* según lo determinado por las funciones `is_` de *PHP*.

- array
- bool
- callable
- float
- double
- int
- integer
- long
- null
- numeric
- object
- real
- resource
- scalar
- string

message

tipo: `string` **predeterminado**: `This value should be of type {{ type }}` (`Este valor debe ser de tipo {{tipo}}`)

El mensaje si el dato subyacente no es del tipo dado.

4.11.8 Email

Valida que un valor es una dirección de correo electrónico válida. El valor subyacente se convierte en una cadena antes de validarlo.

Aplica a	*propiedad o método* (Página 155)
Opciones	• message (Página 523) • checkMX (Página 523)
Clase	`Symfony\Component\Validator\Constraints\Email`
Validador	`Symfony\Component\Validator\Constraints\EmailValid`

Uso básico

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        email:
```

```
            - Email:
                message: El correo electrónico "{{ value }}" no es una dirección de correo elect
                checkMX: true
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Email(
     *     message = "El correo electrónico '{{ value }}' no es una dirección de correo electrón
     *     checkMX = true
     * )
     */
     protected $email;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid email address` (`Este valor no es una dirección de correo electrónico válida`)

Este mensaje se muestra si los datos subyacentes no son una dirección de correo electrónico válida.

checkMX

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, entonces puedes utilizar la función checkdnsrr de *PHP* para comprobar la validez de los registros *MX* del servidor del correo electrónico dado.

4.11.9 MinLength

Valida que la longitud de una cadena por lo menos es tan larga como el límite dado.

Aplica a	*propiedad o método* (Página 155)
Opciones	• limit (Página 524) • message (Página 524) • charset (Página 524)
Clase	`Symfony\Component\Validator\Constraints\MinLength`
Validador	`Symfony\Component\Validator\Constraints\MinLengthV`

Uso básico

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Blog:
    properties:
        nombreDePila:
            - MinLength: { limit: 3, message: Tu nombre debe tener por lo menos {{ limit }} cara
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Blog.php
use Symfony\Component\Validator\Constraints as Assert;

class Blog
{
    /**
     * @Assert\MinLength(
     *     limit: 3,
     *     message: "Tu nombre debe tener por lo menos {{ limit }} caracteres."
     * )
     */
    protected $nombreDePila;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Blog">
    <property name="nombreDePila">
        <constraint name="MinLength">
            <option name="limit">3</option>
            <option name="message">Tu nombre debe tener por lo menos {{ limit }} caracteres.</op
        </constraint>
    </property>
</class>
```

Opciones

limit

tipo: `integer` [*opción predeterminada* (Página 154)]

Esta opción requerida es el valor `mínimo`. La validación fallará si la longitud de la cadena dada es **menor** de este número.

message

tipo: `string` **predeterminado**: `This value is too short. It should have {{ limit }} characters or more` (`Este valor es demasiado corto. Debería tener {{ limit }} caracteres o más`)

El mensaje que se mostrará si la cadena subyacente tiene una longitud menor que la opción limit (Página 524).

charset

tipo: `charset` **predeterminado**: `UTF-8`

Si está instalada la extensión `mbstring` de *PHP*, entonces se utiliza la función mb_strlen de *PHP* para calcular la longitud de la cadena. El valor de la opción `charset` se pasa como segundo argumento a esa función.

4.11.10 MaxLength

Valida que la longitud de una cadena no es mayor que el límite establecido.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ limit (Página 525) ▪ message (Página 526) ▪ charset (Página 526)
Clase	`Symfony\Component\Validator\Constraints\MaxLength`
Validador	`Symfony\Component\Validator\Constraints\MaxLengthV`

Uso básico

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Blog:
    properties:
        resumen:
            - MaxLength: 100
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Blog.php
use Symfony\Component\Validator\Constraints as Assert;

class Blog
{
    /**
     * @Assert\MaxLength(100)
     */
    protected $resumen;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Blog">
    <property name="resumen">
        <constraint name="MaxLength">
            <value>100</value>
        </constraint>
    </property>
</class>
```

Opciones

limit

tipo: `integer` [*opción predeterminada* (Página 154)]

Esta opción requerida es el valor `máximo`. La validación fallará si la longitud de la cadena dada es **mayor** de este número.

message

tipo: `string` **predeterminado**: `This value is too long. It should have {{ limit }} characters or less` (`Este valor es demasiado largo. Debería tener {{ limite }} caracteres o menos`)

El mensaje que se mostrará si la cadena subyacente tiene una longitud mayor que la opción limit (Página 525).

charset

tipo: `charset` **predeterminado**: `UTF-8`

Si está instalada la extensión `mbstring` de *PHP*, entonces se utiliza la función mb_strlen de *PHP* para calcular la longitud de la cadena. El valor de la opción `charset` se pasa como segundo argumento a esa función.

4.11.11 Url

Valida que un valor es una cadena *URL* válida.

Aplica a	*propiedad o método* (Página 155)
Opciones	• message (Página 527) • protocols (Página 527)
Clase	`Symfony\Component\Validator\Constraints\Url`
Validador	`Symfony\Component\Validator\Constraints\UrlValidat`

Uso básico

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        urlBio:
            - Url:
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
 namespace Acme\BlogBundle\Entity;

 use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Url
     */
     protected $urlBio;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid URL (Este valor no es una URL válida)`

Este mensaje aparece si la *URL* no es válida.

protocols

tipo: `array` **predeterminado**: `array('http', 'https')`

Los protocolos que se consideran válidos. Por ejemplo, si también necesitas que sean válidas *URLs* tipo `ftp://`, redefine la matriz de protocolos, listando `http`, `https`, y también `ftp`.

4.11.12 Regex

Valida que un valor coincide con una expresión regular.

Aplica a	*propiedad o método* (Página 155)
Opciones	■ pattern (Página 528) ■ match (Página 528) ■ message (Página 529)
Clase	`Symfony\Component\Validator\Constraints\Regex`
Validador	`Symfony\Component\Validator\Constraints\RegexValid`

Uso básico

Supongamos que tienes un campó `descripción` y que deseas verificar que comienza con un carácter de palabra válido. La expresión regular para comprobar esto sería `/^\w+/`, la cual indica que estás buscando al menos uno o más caracteres constituyentes de palabra al principio de la cadena:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        descripcion:
            - Regex: "/^\w+/"
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Regex("/^\w+/")
     */
```

```
    protected $description;
}
```

Alternativamente, puedes fijar la opción match (Página 528) a `false` con el fin de afirmar que una determinada cadena *no* coincide. En el siguiente ejemplo, afirmas que el campo `nombreDePila` no contiene ningún número y proporcionas un mensaje personalizado:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        nombreDePila:
            - Regex:
                pattern: "/\d/"
                match:   false
                message: Tu nombre no puede contener un número
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Regex(
     *     pattern="/\d/",
     *     match=false,
     *     message="Tu nombre no puede contener un número"
     * )
     */
    protected $nombreDePila;
}
```

Opciones

pattern

tipo: `string` [*opción predeterminada* (Página 154)]

Esta opción requerida es el patrón de la expresión regular con el cual se comparará lo ingresado. Por omisión, este validador no funcionará si la cadena introducida *no* coincide con la expresión regular (a través de la función preg_match de *PHP*). Sin embargo, si estableces match (Página 528) en `false`, la validación fallará si la cadena ingresada *no* coincide con este patrón.

match

tipo: `Boolean` **predeterminado**: `true`

Si es `true` (o no se ha fijado), este validador pasará si la cadena dada coincide con el patrón (pattern (Página 528)) de la expresión regular. Sin embargo, cuando estableces esta opción en `false`, ocurrirá lo contrario: la validación pasará si la cadena ingresada **no** coincide con el patrón de la expresión regular.

message

tipo: `string` **predeterminado**: `This value is not valid(Este valor no es válido)`

Este es el mensaje que se mostrará si el validador falla.

4.11.13 Ip

Valida que un valor es una dirección *IP* válida. Por omisión, este valida el valor como *IPv4*, pero hay una serie de diferentes opciones para validarlo como *IPv6* y muchas otras combinaciones.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ version (Página 529) ▪ message (Página 530)
Clase	`Symfony\Component\Validator\Constraints\Ip`
Validador	`Symfony\Component\Validator\Constraints\IpValidato`

Uso básico

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        ipAddress:
            - Ip:
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
 namespace Acme\BlogBundle\Entity;

 use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Ip
     */
     protected $direccionIp;
}
```

Opciones

version

tipo: `string` **predeterminado**: `4`

Esta determina exactamente *cómo* se valida la dirección *IP* y puede tomar uno de una serie de diferentes valores:

Todos los rangos

- `v4` - Valida por direcciones *IPv4*

- `v6` - Valida por direcciones *IPv6*
- `all` - Valida todos los formatos *IP*

No hay rangos privados

- `4_no_priv` - Valida por *IPv4*, pero sin rangos *IP* privados
- `6_no_priv` - Valida por *IPv6*, pero sin rangos *IP* privados
- `all_no_priv` - Valida todos los formatos *IP*, pero sin rangos *IP* privados

No hay rangos reservados

- `4_no_res` - Valida por *IPv4*, pero sin rangos *IP* reservados
- `6_no_res` - Valida por *IPv6*, pero sin rangos *IP* reservados
- `all_no_res` - Valida todos los formatos *IP*, pero sin rangos *IP* reservados

Sólo rangos públicos

- `4_public` - Valida por *IPv4*, pero sin rangos privados y reservados
- `6_public` - Valida por *IPv6*, pero sin rangos privados y reservados
- `all_public` - Valida todos los formatos *IP*, pero sin rangos privados y reservados

message

tipo: `string` **predeterminado**: `This is not a valid IP address` (`Esta no es una dirección IP válida`)

Este mensaje se muestra si la cadena no es una dirección *IP* válida.

4.11.14 Max

Valida que un número dado es *menor* que un número máximo.

Aplica a	*propiedad o método* (Página 155)
Opciones	• limit (Página 531) • message (Página 531) • invalidMessage (Página 531)
Clase	`Symfony\Component\Validator\Constraints\Max`
Validador	`Symfony\Component\Validator\Constraints\MaxValidat`

Uso básico

Para verificar que el campo `edad` de una clase no es superior a `50`, puedes agregar lo siguiente:

- *YAML*

```
# src/Acme/EventoBundle/Resources/config/validation.yml
Acme\EventoBundle\Entity\Participante:
    properties:
        edad:
            - Max: { limit: 50, message: Debes tener 50 años o menos para entrar. }
```

- *Annotations*

```
// src/Acme/EventoBundle/Entity/Participante.php
use Symfony\Component\Validator\Constraints as Assert;

class Participante
{
    /**
     * @Assert\Max(limit = "50", message = "Debes tener 50 años o menos para entrar.")
     */
     protected $edad;
}
```

Opciones

limit

tipo: `integer` [*opción predeterminada* (Página 154)]

Esta opción requerida es el valor `máximo`. La validación fallará si el valor es **mayor** que este valor máximo.

message

tipo: `string` **predeterminado**: `This value should be {{ limit }} or less` (`Este valor debe ser {{ limite }} o menor`)

El mensaje que se mostrará si el valor subyacente es mayor que la opción limit (Página 531).

invalidMessage

tipo: `string` **predeterminado**: `This value should be a valid number` (`Este valor debe ser un número válido`)

El mensaje que se mostrará si el valor subyacente no es un número (por la función is_numeric de *PHP*).

4.11.15 Min

Valida que un número dado es *mayor* que un número mínimo.

Aplica a	*propiedad o método* (Página 155)
Opciones	• limit (Página 532) • message (Página 532) • invalidMessage (Página 532)
Clase	`Symfony\Component\Validator\Constraints\Min`
Validador	`Symfony\Component\Validator\Constraints\MinValidat`

Uso básico

Para verificar que el campo `edad` de una clase es `18` o más, puedes agregar lo siguiente:

- *YAML*

```
# src/Acme/EventoBundle/Resources/config/validation.yml
Acme\EventoBundle\Entity\Participante:
    properties:
        edad:
            - Min: { limit: 18, message: Debes tener 18 años o más para entrar. }
```

- *Annotations*

```
// src/Acme/EventoBundle/Entity/Participante.php
use Symfony\Component\Validator\Constraints as Assert;

class Participante
{
    /**
     * @Assert\Min(limit = "18", message = "Debes tener 18 años o más para entrar.")
     */
     protected $edad;
}
```

Opciones

limit

tipo: `integer` [*opción predeterminada* (Página 154)]

Esta opción requerida es el valor `mínimo`. La validación fallará si el valor es **menor** que este valor `mínimo`.

message

tipo: `string` **predeterminado**: `This value should be {{ limit }} or more` (`Este valor debe ser {{ limite }} o más`)

El mensaje que se mostrará si el valor subyacente es menor que la opción limit (Página 532).

invalidMessage

tipo: `string` **predeterminado**: `This value should be a valid number` (`Este valor debe ser un número válido`)

El mensaje que se mostrará si el valor subyacente no es un número (por la función is_numeric de *PHP*).

4.11.16 Date

Valida que un valor sea una fecha válida, es decir, ya sea un objeto `DateTime` o una cadena (o un objeto que se pueda convertir en una cadena) que sigue un formato "AAAA-MM-DD" válido.

Aplica a	*propiedad o método* (Página 155)
Opciones	• message (Página 533)
Clase	`Symfony\Component\Validator\Constraints\Date`
Validador	`Symfony\Component\Validator\Constraints\DateValida`

Uso básico

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        aniversario:
            - Date: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Date()
     */
     protected $aniversario;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid date` (`Este valor no es una fecha válida`)

Este mensaje se muestra si los datos subyacentes no son una fecha válida.

4.11.17 DateTime

Valida que un valor sea una fecha y hora (`datetime`) válidas, es decir, ya sea un objeto `DateTime` o una cadena (o un objeto que se pueda convertir en una cadena) que sigue un formato "AAAA-MM-DD HH:MM:SS" válido.

Aplica a	*propiedad o método* (Página 155)
Opciones	• message (Página 534)
Clase	`Symfony\Component\Validator\Constraints\DateTime`
Validador	`Symfony\Component\Validator\Constraints\DateTimeVa`

Uso básico

- *YAML*

```
# src/Acme/EventoBundle/Resources/config/validation.yml
Acme\BlobBundle\Entity\Autor:
    properties:
        createdAt:
            - DateTime: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\DateTime()
     */
     protected $creadoEl;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid datetime` (`Este valor no representa una fecha y hora válidas`)

Este mensaje se muestra si los datos subyacentes no son una fecha y hora válidas.

4.11.18 Time

Valida que un valor es una cadena de tiempo válida con el formato "HH:MM:SS".

Valida que un valor es una hora válida, es decir, ya sea un objeto `DateTime` o una cadena (o un objeto que se puede convertir en una cadena) que sigue un formato "HH:MM:SS" válido.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 535)
Clase	`Symfony\Component\Validator\Constraints\Time`
Validador	`Symfony\Component\Validator\Constraints\TimeValida`

Uso básico

Supongamos que tienes una clase Evento, con un campo `comenzaraALas` que es el momento del día en que comienza el evento:

- *YAML*

```
# src/Acme/EventoBundle/Resources/config/validation.yml
Acme\EventoBundle\Entity\Evento:
    properties:
        comenzaraALas:
            - Time: ~
```

- *Annotations*

```
// src/Acme/EventoBundle/Entity/Evento.php
namespace Acme\EventoBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;
```

```
class Evento
{
    /**
     * @Assert\Time()
     */
     protected $comenzaraALas;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid time` (`Este valor no es una hora válida`)

Este mensaje se muestra si los datos subyacentes no son una hora válida.

4.11.19 Choice

Esta restricción se utiliza para asegurar que el valor dado es uno de un determinado conjunto de opciones *válidas*. También la puedes utilizar para comprobar que cada elemento de una matriz de elementos es una de esas opciones válidas.

Aplica a	*propiedad o método* (Página 155)
Opciones	■ choices (Página 538) ■ callback (Página 538) ■ multiple (Página 538) ■ min (Página 538) ■ max (Página 539) ■ message (Página 539) ■ multipleMessage (Página 539) ■ minMessage (Página 539) ■ maxMessage (Página 539) ■ strict (Página 539)
Clase	`Symfony\Component\Validator\Constraints\Choice`
Validador	`Symfony\Component\Validator\Constraints\ChoiceVali`

Uso básico

La idea básica de esta restricción es que le proporciones un arreglo de valores válidos (esto lo puedes hacer de varias maneras) y ella compruebe que el valor de la propiedad dada existe en el arreglo.

Si tu lista de opciones válidas es simple, la puedes pasar directamente a través de la opción choices (Página 538):

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        genero:
            - Choice:
```

```
                choices:  [masculino, femenino]
                message:  Elige un género válido.
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="genero">
        <constraint name="Choice">
            <option name="choices">
                <value>masculino</value>
                <value>femenino</value>
            </option>
            <option name="message">Elige un género válido.</option>
        </constraint>
    </property>
</class>
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Choice(choices = {"masculino", "femenino"}, message = "Elige un género válido.")
     */
    protected $genero;
}
```

- *PHP*

```
// src/Acme/BlogBundle/EntityAutor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Choice;

class Autor
{
    protected $genero;

    public static function cargaValidadorDeMetadatos(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('genero', new Choice(
            'choices' => array('masculino', 'femenino'),
            'message' => 'Elige un género válido.',
        ));
    }
}
```

Suministrando opciones con una función retrollamada

También puedes utilizar una función retrollamada para especificar tus opciones. Esto es útil si deseas mantener tus opciones en una ubicación central para que, por ejemplo, puedas acceder fácilmente a las opciones para validación o para construir un elemento de formulario seleccionado.

```
// src/Acme/BlogBundle/Entity/Autor.php
class Autor
```

```
{
    public static function getGeneros()
    {
        return array('masculino', 'femenino');
    }
}
```

Puedes pasar el nombre de este método a la opción `callback` de la restricción `Choice`.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        genero:
            - Choice: { callback: getGeneros }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Choice(callback = "getGenders")
     */
    protected $genero;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="genero">
        <constraint name="Choice">
            <option name="callback">getGeneros</option>
        </constraint>
    </property>
</class>
```

Si la retrollamada estática se almacena en una clase diferente, por ejemplo `Util`, puedes pasar el nombre de clase y el método como una matriz.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    properties:
        genero:
            - Choice: { callback: [Util, getGeneros] }
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="genero">
        <constraint name="Choice">
            <option name="callback">
                <value>Util</value>
```

```
                <value>getGeneros</value>
            </option>
        </constraint>
    </property>
</class>
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\Choice(callback = {"Util", "getGenders"})
     */
    protected $genero;
}
```

Opciones disponibles

choices

tipo: `array` [*opción predeterminada* (Página 154)]

Una opción requerida (a menos que especifiques callback (Página 538)) - esta es la gran variedad de opciones que se deben considerar en el conjunto válido. El valor de entrada se compara contra esta matriz.

Callback

tipo: `string|array|Closure`

Este es un método retrollamado que puedes utilizar en lugar de la opción choices (Página 538) para devolver la matriz de opciones. Consulta Suministrando opciones con una función retrollamada (Página 536) para más detalles sobre su uso.

multiple

tipo: `Boolean` **predeterminado**: `false`

Si esta opción es `true`, se espera que el valor de entrada sea una matriz en lugar de un solo valor escalar. La restricción debe verificar que cada valor de la matriz de entrada se pueda encontrar en el arreglo de opciones válidas. Si no se puede encontrar alguno de los valores ingresados, la validación fallará.

min

tipo: `integer`

Si la opción `multiple` es `true`, entonces puedes usar la opción `min` para forzar que por lo menos se seleccionen XX valores. Por ejemplo, si `min` es 3, pero la matriz de entrada sólo contiene dos elementos válidos, la validación fallará.

max

tipo: `integer`

Si la opción `multiple` es `true`, entonces puedes usar la opción `max` para forzar que no se seleccionen más de XX valores. Por ejemplo, si `max` es 3, pero la matriz ingresada consta de 4 elementos válidos, la validación fallará.

message

tipo: `string` **predeterminado**: `The value you selected is not a valid choice` (`El valor que seleccionaste no es una opción válida`)

Este es el mensaje que recibirás si la opción `multiple` se establece en `false`, y el valor subyacente no es válido en la matriz de opciones.

multipleMessage

tipo: `string` **predeterminado**: `One or more of the given values is invalid` (`Uno o más de los valores dados no es válido`)

Este es el mensaje que recibirás si la opción `multiple` se establece en `true`, y uno de los valores de la matriz subyacente que se está comprobando no está en la matriz de opciones válidas.

minMessage

tipo: `string` **predeterminado**: `You must select at least {{ limit }} choices` (`Debes seleccionar por lo menos {{ limite }} opciones`)

Este es el mensaje de error de validación que se muestra cuando el usuario elige demasiado pocas opciones para la opción min (Página 538).

maxMessage

tipo: `string` **predeterminado**: `You must select at most {{ limit }} choices` (`Debes seleccionar por lo menos {{ limite }} opciones`)

Este es el mensaje de error de validación que se muestra cuando el usuario elige más opciones para la opción max (Página 539).

strict

tipo: `Boolean` **predeterminado**: `false`

Si es `true`, el validador también comprobará el tipo del valor ingresado. En concreto, este valor se pasa como el tercer argumento del método in_array de *PHP* cuando compruebe si un valor está en la matriz de opciones válidas.

4.11.20 Collection

Esta restricción se utiliza cuando los datos subyacentes son una colección (es decir, una matriz o un objeto que implemente `Traversable` y `ArrayAccess`), pero que te gustaría validar las distintas claves de la colección de diferentes

maneras. Por ejemplo, puedes validar la clave `email` usando la restricción `Email` y la clave `inventario` de la colección con la restricción `min`.

Esta restricción también puede asegurarse de que ciertas claves de la colección están presentes y que no se presentan claves adicionales.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ fields (Página 542) ▪ allowExtraFields (Página 542) ▪ extraFieldsMessage (Página 542) ▪ allowMissingFields (Página 542) ▪ missingFieldsMessage (Página 543)
Clase	`Symfony\Component\Validator\Constraints\Collection`
Validador	`Symfony\Component\Validator\Constraints\Collection`

Uso básico

La restricción `Collection` te permite validar individualmente las diferentes claves de una colección. Tomemos el siguiente ejemplo:

```
namespace Acme\BlogBundle\Entity;

class Autor
{
    protected $datosDelPerfil = array(
        'correo_electronico_personal',
        'mini_biografia',
    );

    public function setDatosDelPerfil($clave, $valor)
    {
        $this->datosDelPerfil[$clave] = $valor;
    }
}
```

Para validar que el elemento `correo_electronico_personal` de la propiedad `datosDelPerfil` es una dirección de correo electrónico válida y que el elemento `mini_biografia` no está en blanco, pero no tiene más de 100 caracteres de longitud, debes hacer lo lo siguiente:

- *YAML*

```
properties:
    datosDelPerfil:
        - Collection:
            fields:
                correo_electronico_personal: Email
                mini_biografia:
                    - NotBlank
                    - MaxLength:
                        limit:   100
                        message: ¡Tu minibiografía es muy larga!
            allowMissingfields: true
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;
```

```
class Autor
{
    /**
     * @Assert\Collection(
     *     fields = {
     *         "correo_electronico_personal" = @Assert\Email,
     *         "mini_biografia" = {
     *             @Assert\NotBlank(),
     *             @Assert\MaxLength(
     *                 limit = 100,
     *                 message = "¡Tu minibiografía es muy larga!"
     *             )
     *         }
     *     },
     *     allowMissingfields = true
     * )
     */
     protected $datosDelPerfil = array(
         'correo_electronico_personal',
         'mini_biografia',
     );
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="datosDelPerfil">
        <constraint name="Collection">
            <option name="fields">
                <value key="correo_electronico_personal">
                    <constraint name="Email" />
                </value>
                <value key="mini_biografia">
                    <constraint name="NotBlank" />
                    <constraint name="MaxLength">
                        <option name="limit">100</option>
                        <option name="message">¡Tu minibiografía es muy larga!</option>
                    </constraint>
                </value>
            </option>
            <option name="allowMissingFields">true</option>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Collection;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MaxLength;

class Autor
{
    private $opciones = array();
```

```
        public static function cargaValidadorDeMetadatos(ClassMetadata $metadatos)
        {
            $metadatos->addPropertyConstraint('datosDelPerfil', new Collection(array(
                'fields' => array(
                    'correo_electronico_personal' => arraynew Email(),
                    'apellido' => array(new NotBlank(), new MaxLength(100)),
                ),
                'allowMissingFields' => true,
            )));
        }
    }
```

Presencia y ausencia de campos

De manera predeterminada, esta restricción valida más que simplemente cuando o no los campos individuales de la colección pasan sus restricciones asignadas. De hecho, en si están alguna de las claves de una colección, o si hay alguna clave no reconocida en la colección, lanzará errores de validación.

Si deseas permitir que las claves de la colección estén ausentes o si quisieras permitir claves "extra" en la colección, puedes modificar las opciones allowMissingFields (Página 542) y allowExtraFields (Página 542), respectivamente. En el ejemplo anterior, la opción `allowMissingFields` se fijó en `true`, lo cual significa que si cualquiera de las propiedades `correo_electronico_personal` o `mini_biografia` no estuviera en `$datosPersonales`, no ocurrirá ningún error de validación.

Opciones

fields

tipo: `array` [*opción predeterminada* (Página 154)]

Esta opción es obligatoria, y es una matriz asociativa que define todas las claves de la colección y, para cada clave, exactamente, que validaciones se deben ejecutar contra ese elemento de la colección.

allowExtraFields

tipo: `Boolean` **predeterminado**: `false`

Si esta opción está establecida en `false` y la colección subyacente contiene uno o más elementos que no están incluidos en la opción fields (Página 542), devolverá un error de validación. Si se define como `true`, está bien tener campos adicionales.

extraFieldsMessage

tipo: `Boolean` **predeterminado**:`The fields {{ fields }} were not expected` (`Los campos {{ fields }} no se esperaban`)

El mensaje aparece si allowExtraFields (Página 542) es `false` y se detecta un campo adicional.

allowMissingFields

tipo: `Boolean` **predeterminado**: `false`

Si esta opción está establecida en `false` y uno o más campos de la opción fields (Página 542) no están presentes en la colección subyacente, devolverá un error de validación. Si se define como `true`, está bien si algunos campos de la opción fields (Página 542) no están presentes en la colección subyacente.

missingFieldsMessage

tipo: `Boolean` **predeterminado**: `The fields {{ fields }} are missing (Faltan los campos {{ campos }})`

El mensaje aparece si allowMissingFields (Página 542) es `false` y uno o más campos no se encuentran en la colección subyacente.

4.11.21 UniqueEntity

Valida que un campo en particular (o campos) en una entidad *Doctrine* sea único. Este se utiliza comúnmente, por ejemplo, para prevenir que un nuevo usuario se registre con una dirección de correo electrónico existente en el sistema.

Aplica a	*class* (Página 158)
Opciones	▪ fields (Página 544) ▪ message (Página 544) ▪ em (Página 544)
Clase	`Symfony\Bridge\Doctrine\Validator\Constraints\Uniq`
Validador	`Symfony\Bridge\Doctrine\Validator\Constraints\Uniq`

Uso básico

Supongamos que tienes un `AcmeUsuarioBundle` con una entidad `Usuario` que tiene un campo de `correo` electrónico. Puedes utilizar la restricción `Unique` para garantizar que el campo `correo` siga siendo único en toda tu tabla Usuario:

- *Annotations*

```
// Acme/UsuarioBundle/Entity/Usuario.php
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints as DoctrineAssert;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @DoctrineAssert\UniqueEntity("correo")
 */
class Autor
{
    /**
     * @var string $correo
     *
     * @ORM\Column(name="correo", type="string", length=255, unique=true)
     * @Assert\Email()
     */
    protected $correo;

    // ...
}
```

- *YAML*

```
# src/Acme/UsuarioBundle/Resources/config/validation.yml
constraints:
    - UniqueEntity: correo
```

Opciones

fields

tipo: `array``|``string` [*porción predeterminada* (Página 154)]

Esta opción requerida es el campo (o lista de campos) en el cual esta entidad debe ser única. Por ejemplo, puedes especificar que tanto el campo `correo` electrónico como el `nombre` de usuario en el ejemplo anterior deben ser únicos.

message

tipo: `string` **predeterminado**: `This value is already used.` (`Este valor ya se está utilizando.`)

El mensaje a mostrar cuando esta restricción falla.

em

tipo: `string`

El nombre del gestor de la entidad a utilizar para hacer la consulta para determinar la singularidad. Si se deja en blanco, usará el gestor de entidades predeterminado.

4.11.22 Language

Valida que un valor es un código de idioma válido.

Aplica a	*propiedad o método* (Página 155)
Opciones	• message (Página 545)
Clase	`Symfony\Component\Validator\Constraints\Language`
Validador	`Symfony\Component\Validator\Constraints\LanguageVa`

Uso básico

- *YAML*

```
# src/UsuarioBundle/Resources/config/validation.yml
Acme\UsuarioBundle\Entity\Usuario:
    properties:
        idiomaPreferido:
            - Language:
```

- *Annotations*

```
// src/Acme/UsuarioBundle/Entity/Usuario.php
namespace Acme\UsuarioBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Usuario
{
    /**
     * @Assert\Language
     */
     protected $idiomaPreferido;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid language` (`Este valor no es un idioma válido`)

Este mensaje se muestra si la cadena no es un código de idioma válido.

4.11.23 Locale

Valida que un valor es una región válida.

El "valor" de cada región es o bien el del código de *idioma* ISO639-1 de dos letras (por ejemplo, `es`), o el código de idioma seguido de un guión bajo (_), luego el código de *país* ISO3166 (por ejemplo, `es_ES` para el Español/España).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 546)
Clase	`Symfony\Component\Validator\Constraints\Locale`
Validador	`Symfony\Component\Validator\Constraints\LocaleVali`

Uso básico

- *YAML*

```
# src/UsuarioBundle/Resources/config/validation.yml
Acme\UsuarioBundle\Entity\Usuario:
    properties:
        region:
            - Locale:
```

- *Annotations*

```
// src/Acme/UsuarioBundle/Entity/Usuario.php
namespace Acme\UsuarioBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Usuario
{
```

```
    /**
     * @Assert\Locale
     */
     protected $region;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid locale` (`Este valor no es una región válida`)

Este mensaje se muestra si la cadena no es una región válida.

4.11.24 Country

Valida que un valor es un código de país de dos letras válido.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ message (Página 547)
Clase	`Symfony\Component\Validator\Constraints\Country`
Validador	`Symfony\Component\Validator\Constraints\CountryVal`

Uso básico

- *YAML*

```
# src/UsuarioBundle/Resources/config/validation.yml
Acme\UsuarioBundle\Entity\Usuario:
    properties:
        pais:
            - Country:
```

- *Annotations*

```
// src/Acme/UsuarioBundle/Entity/Usuario.php
namespace Acme\UsuarioBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Usuario
{
    /**
     * @Assert\Country
     */
     protected $pais;
}
```

Opciones

message

tipo: `string` **predeterminado**: `This value is not a valid country` (`Este valor no es un código de país válido`)

Este mensaje se muestra si la cadena no es un código de país válido.

4.11.25 File

Valida que un valor es un "archivo" válido, el cual puede ser uno de los siguientes:

- Una cadena (u objeto con un método `__toString()`) que representa una ruta a un archivo existente;
- Un objeto `Symfony\Component\HttpFoundation\File\File` válido (incluidos los objetos de la clase `Symfony\Component\HttpFoundation\File\UploadedFile`).

Esta restricción se usa comúnmente en formularios con el tipo de formulario *file (archivo)* (Página 480).

Truco: Si el archivo que estás validando es una imagen, prueba la restricción *Image* (Página 551).

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ maxSize (Página 549) ▪ mimeTypes (Página 549) ▪ maxSizeMessage (Página 549) ▪ mimeTypesMessage (Página 550) ▪ notFoundMessage (Página 550) ▪ notReadableMessage (Página 550) ▪ uploadIniSizeErrorMessage (Página 550) ▪ uploadFormSizeErrorMessage (Página 550) ▪ uploadErrorMessage (Página 550)
Clase	`Symfony\Component\Validator\Constraints\File`
Validador	`Symfony\Component\Validator\Constraints\FileValida`

Uso básico

Esta restricción se utiliza comúnmente en una propiedad que se debe pintar en un formulario con tipo de formulario *archivo* (Página 480). Por ejemplo, supongamos que estás creando un formulario de autor donde puedes cargar una "bio" *PDF* para el autor. En tu formulario, la propiedad `archivoBio` sería de tipo `file`. La clase `Autor` podría ser la siguiente:

```
// src/Acme/BlogBundle/Entity/Autor.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Autor
{
    protected $archivoBio;

    public function setArchivoBio(File $archivo = null)
    {
```

```
        $this->archivoBio = $archivo;
    }

    public function getArchivoBio()
    {
        return $this->archivoBio;
    }
}
```

Para garantizar que el objeto `archivoBio` es un `File` válido, y que está por debajo de un determinado tamaño de archivo y es un archivo *PDF* válido, añade lo siguiente:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor
    properties:
        archivoBio:
            - File:
                maxSize: 1024k
                mimeTypes: [application/pdf, application/x-pdf]
                mimeTypesMessage: Por favor sube un archivo PDF válido
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /**
     * @Assert\File(
     *     maxSize = "1024k",
     *     mimeTypes = {"application/pdf", "application/x-pdf"},
     *     mimeTypesMessage = "Por favor sube un archivo PDF válido"
     * )
     */
    protected $archivoBio;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Autor">
    <property name="archivoBio">
        <constraint name="File">
            <option name="maxSize">1024k</option>
            <option name="mimeTypes">
                <value>application/pdf</value>
                <value>application/x-pdf</value>
            </option>
            <option name="mimeTypesMessage">Por favor sube un archivo PDF válido</option>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php
// ...
```

```
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\File;

class Autor
{
    // ...

    public static function cargaValidadorDeMetadatos(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('archivoBio', new File(array(
            'maxSize' => '1024k',
            'mimeTypes' => array(
                'application/pdf',
                'application/x-pdf',
            ),
            'mimeTypesMessage' => 'Por favor sube un archivo PDF válido',
        )));
    }
}
```

La propiedad `archivoBio` es validada para garantizar que se trata de un archivo real. Su tamaño y tipo *MIME* también son validados por las correspondientes opciones especificadas.

Opciones

maxSize

tipo: `mixed`

Si se establece, el tamaño del archivo subyacente debe estar por debajo de ese tamaño de archivo para ser válido. El tamaño del archivo se puede dar en uno de los siguientes formatos:

- **bytes**: Para especificar el `maxSize` en bytes, pasa un valor que sea totalmente numérico (por ejemplo, `4096`);
- **kilobytes**: Para especificar el `maxSize` en kilobytes, pasa un número y el sufijo con una "k" minúscula (por ejemplo, `200k`);
- **megabytes**: Para especificar el `maxSize` en megabytes, pasa un número y sufijo con una "M" (por ejemplo, `4M`).

mimeTypes

tipo: `array` o `string`

Si lo estableces, el validador comprobará que el tipo *mime* del archivo subyacente sea igual al tipo *mime* proporcionado (en el caso de una cadena) o existe en la colección de determinados tipos *mime* (en el caso de una matriz).

maxSizeMessage

tipo: `string` **predeterminado**: `The file is too large ({{ size }}). Allowed maximum size is {{ limit }}` (`El archivo es demasiado grande ({{ tamaño }} ). El tamaño máximo permitido es de {{ limite }}`)

El mensaje mostrado si el archivo es mayor que la opción maxSize (Página 549).

mimeTypesMessage

tipo: `string` **predeterminado**: `The mime type of the file is invalid ({{ type }}). Allowed mime types are {{ types }}` (`El tipo mime del archivo no es válido ({{ tipo }}). Los tipos mime son {{ tipos }}`)

El mensaje mostrado si el tipo *mime* del archivo no es un tipo *mime* válido para la opción mimeTypes (Página 549).

notFoundMessage

tipo: `string` **predeterminado**: `The file could not be found` (`No se pudo encontrar el archivo`)

El mensaje aparece si no se puede encontrar algún archivo en la ruta especificada. Este error sólo es probable si el valor subyacente es una cadena de ruta, puesto que un objeto `File` no se puede construir con una ruta de archivo no válida.

notReadableMessage

tipo: `string` **predeterminado**: `The file is not readable` (`El archivo no es legible`)

El mensaje aparece si el archivo existe, pero la función `is_readable` de *PHP* falla cuando se le pasa la ruta del archivo.

uploadIniSizeErrorMessage

tipo: `string` **predeterminado**: `The file is too large. Allowed maximum size is {{ limit }}` (`El archivo es demasiado grande ({{ tamaño }} ). El tamaño máximo permitido es de {{ limite }}`)

El mensaje que se muestra si el archivo subido es mayor que la configuración de `upload_max_filesize` en *php.ini*.

uploadFormSizeErrorMessage

tipo: `string` **predeterminado**: `The file is too large` (`El archivo es demasiado grande`)

El mensaje que se muestra si el archivo subido es mayor que el permitido por el campo *HTML* para la entrada de archivos.

uploadErrorMessage

tipo: `string` **predeterminado**: `The file could not be uploaded` (`El archivo no se puede subir`)

El mensaje que se muestra si el archivo cargado no se puede subir por alguna razón desconocida, tal como cuando la subida del archivo ha fallado o no se puede escribir en el disco.

4.11.26 Image

La restricción `Image` funciona exactamente igual que la restricción *File* (Página 547), salvo que sus opciones mimeTypes (Página 551) y mimeTypesMessage (Página 551) se configuran automáticamente para trabajar con archivos de imagen específicamente.

Consulta la restricción *File* (Página 547) para la mayor parte de la documentación relativa a esta restricción.

Opciones

Esta restricción comparte todas sus opciones con la restricción *File* (Página 547). Lo hace, sin embargo, modifica dos de los valores predeterminados de la opción:

mimeTypes

tipo: `array` o `string` **predeterminado**: un arreglo de tipos *mime* de imágenes `jpg`, `gif` y `png`

mimeTypesMessage

tipo: `string` **predeterminado**: `This file is not a valid image` (`El archivo no es una imagen válida`)

4.11.27 Callback

El propósito de la afirmación `Callback` es permitirte crear reglas de validación completamente personalizadas y asignar errores de validación a campos específicos de tu objeto. Si estás utilizando la validación con formularios, esto significa que puedes hacer que estos errores personalizados se muestren junto a un campo específico, en lugar de simplemente en la parte superior del formulario.

Este proceso funciona especificando uno o más métodos `Callback`, cada uno de los cuales se llama durante el proceso de validación. Cada uno de estos métodos puede hacer cualquier cosa, incluyendo la creación y asignación de errores de validación.

Nota: Un método retrollamado en sí no *falla* o devuelve cualquier valor. En su lugar, como verás en el ejemplo, un método retrollamado tiene la posibilidad de agregar "violaciones" de validación directamente.

Aplica a	*class* (Página 158)
Opciones	▪ methods (Página 552)
Clase	`Symfony\Component\Validator\Constraints\Callback`
Validador	`Symfony\Component\Validator\Constraints\CallbackVa`

Configurando

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    constraints:
```

```
        - Callback:
            methods:    [isAutorValido]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Callback(methods={"isAutorValido"})
 */
class Autor
{
}
```

El método `Callback`

Al método `callback` se le pasa un objeto `ExecutionContext` especial:: Puedes establecer `"violaciones"` directamente en el objeto y determinar a qué campo de deben atribuir estos errores:

```
// ...
use Symfony\Component\Validator\ExecutionContext;

class Autor
{
    // ...
    private $nombreDePila;

    public function isAutorValido(ExecutionContext $contexto)
    {
        // algo que tiene un arreglo de "nombres falsos"
        $nombresFalsos = array();

        // comprueba si el nombre en realidad es un nombre falso
        if (in_array($this->getNombreDePila(), $nombresFalsos)) {
            $propiedad_ruta = $contexto->getPropertyPath() . '.nombreDePila';
            $contexto->setPropertyPath($property_path);
            $contexto->addViolation('¡Este nombre suena falso completamente!', array(), null);
        }
    }
```

Opciones

`methods`

tipo: `array` **predeterminado**: `array()` [*opción predeterminada* (Página 154)]

Este es un arreglo de los métodos que se deben ejecutar durante el proceso de validación. Cada método puede tener uno de los siguientes formatos:

1. **Cadena de nombre del método**

   Si el nombre de un método es una cadena simple (por ejemplo, `isAutorValido`), ese método será llamado en el mismo objeto que se está validando y `ExecutionContext` será el único argumento (ve el ejemplo anterior).

2. **Arreglo estático de retrollamada**

También puedes especificar cada método como una matriz de retrollamada estándar:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Autor:
    constraints:
        - Callback:
            methods:
                -    [Acme\BlogBundle\MiClaseValidadorEstatico, isAutorValido]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Callback(methods={
 *     { "Acme\BlogBundle\MiClaseValidadorEstatico", "isAutorValido"}
 * })
 */
class Autor
{
}
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Autor.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Callback;

class Autor
{
    public $nombre;

    public static function cargaValidadorDeMetadatos(ClassMetadata $metadatos)
    {
        $metadatos->addConstraint(new Callback(array(
            'methods' => array('isAutorValido'),
        )));
    }
}
```

En este caso, el método estático `isAutorValido` será llamado en la clase `Acme\BlogBundle\MyStaticValidatorClass`. Este se pasa tanto al objeto original que se está validando (por ejemplo, `Autor`), así como a `ExecutionContext`:

```
namespace Acme\BlogBundle;

use Symfony\Component\Validator\ExecutionContext;
use Acme\BlogBundle\Entity\Autor;

class MiClaseValidadorEstatico
{
    static public function isAutorValido(Autor $autor, ExecutionContext $contexto)
    {
        // ...
```

```
        }
    }
```

Truco: Si especificas la restricción `Retrollamada` a través de *PHP*, entonces también tienes la opción de hacer tu retrollamada o bien un cierre *PHP* o una retrollamada no estática. Esto *no* es posible actualmente, sin embargo, para especificar un *servicio* como una restricción. Para validar usando un servicio, debes *crear una restricción de validación personalizada* (Página 323) y añadir la nueva restricción a tu clase.

4.11.28 `Valid`

Esta restricción se utiliza para permitir la validación de objetos que se incrustan como propiedades en un objeto que se está validando. Esto te permite validar un objeto y todos los subobjetos asociados con él.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ traverse (Página 557)
Clase	`Symfony\Component\Validator\Constraints\Type`

Uso básico

En el siguiente ejemplo, creamos dos clases `Autor` y `Domicilio` donde ambas tienen restricciones en sus propiedades. Además, `Autor` almacena una instancia de `Domicilio` en la propiedad `$domicilio`.

```
// src/Acme/HolaBundle/Domicilio.php
class Domicilio
{
    protected $calle;
    protected $codigoPostal;
}

// src/Acme/HolaBundle/Autor.php
class Autor
{
    protected $nombreDePila;
    protected $apellido;
    protected $domicilio;
}
```

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/validation.yml
Acme\HolaBundle\Domicilio:
    properties:
        calle:
            - NotBlank: ~
        codigoPostal:
            - NotBlank: ~
            - MaxLength: 5

Acme\HolaBundle\Autor:
    properties:
        nombreDePila:
```

```
            - NotBlank: ~
            - MinLength: 4
        apellido:
            - NotBlank: ~
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/validation.xml -->
<class name="Acme\HolaBundle\Domicilio">
    <property name="calle">
        <constraint name="NotBlank" />
    </property>
    <property name="codigoPostal">
        <constraint name="NotBlank" />
        <constraint name="MaxLength">5</constraint>
    </property>
</class>

<class name="Acme\HolaBundle\Autor">
    <property name="nombreDePila">
        <constraint name="NotBlank" />
        <constraint name="MinLength">4</constraint>
    </property>
    <property name="apellido">
        <constraint name="NotBlank" />
    </property>
</class>
```

- *Annotations*

```
// src/Acme/HolaBundle/Domicilio.php
use Symfony\Component\Validator\Constraints as Assert;

class Domicilio
{
    /**
     * @Assert\NotBlank()
     */
    protected $calle;

    /**
     * @Assert\NotBlank
     * @Assert\MaxLength(5)
     */
    protected $codigoPostal;
}

// src/Acme/HolaBundle/Autor.php
class Autor
{
    /**
     * @Assert\NotBlank
     * @Assert\MinLength(4)
     */
    protected $nombreDePila;

    /**
     * @Assert\NotBlank
     */
```

```
    protected $apellido;

    protected $domicilio;
}
```

- *PHP*

```
// src/Acme/HolaBundle/Domicilio.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MaxLength;

class Domicilio
{
    protected $calle;

    protected $codigoPostal;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('calle', new NotBlank());
        $metadatos->addPropertyConstraint('codigoPostal', new NotBlank());
        $metadatos->addPropertyConstraint('codigoPostal', new MaxLength(5));
    }
}

// src/Acme/HolaBundle/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class Autor
{
    protected $nombreDePila;

    protected $apellido;

    protected $domicilio;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('nombreDePila', new NotBlank());
        $metadatos->addPropertyConstraint('nombreDePila', new MinLength(4));
        $metadatos->addPropertyConstraint('apellido', new NotBlank());
    }
}
```

Con esta asignación puedes validar satisfactoriamente a un autor con una dirección no válida. Para evitarlo, agrega la restricción `Valid` a la propiedad `$domicilio`.

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/validation.yml
Acme\HolaBundle\Autor:
    properties:
        domicilio:
            - Valid: ~
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/validation.xml -->
<class name="Acme\HolaBundle\Autor">
    <property name="domicilio">
        <constraint name="Valid" />
    </property>
</class>
```

- *Annotations*

```
// src/Acme/HolaBundle/Autor.php
use Symfony\Component\Validator\Constraints as Assert;

class Autor
{
    /* ... */

    /**
     * @Assert\Valid
     */
    protected $domicilio;
}
```

- *PHP*

```
// src/Acme/HolaBundle/Autor.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Valid;

class Autor
{
    protected $domicilio;

    public static function loadValidatorMetadata(ClassMetadata $metadatos)
    {
        $metadatos->addPropertyConstraint('domicilio', new Valid());
    }
}
```

Si ahora validas a un autor con una dirección no válida, puedes ver que la validación de los campos de `Domicilio` fracasa.

> AcmeHolaBundleAutor.domicilio.codigoPostal: Este valor es demasiado largo. Debe tener 5 caracteres o menos

Opciones

`traverse`

type: `string` **predeterminado**: `true`

Si esta restricción se aplica a una propiedad que contiene una matriz de objetos, cada objeto de la matriz será válido sólo si esta opción está establecida en `true`.

4.11.29 `All`

Cuando se aplica a una matriz (o un objeto `atravesable`), esta restricción te permite aplicar un conjunto de restricciones a cada elemento de la matriz.

Aplica a	*propiedad o método* (Página 155)
Opciones	▪ constraints (Página 558)
Clase	`Symfony\Component\Validator\Constraints\All`
Validador	`Symfony\Component\Validator\Constraints\AllValidat`

Uso básico

Supongamos que tienes una matriz de cadenas, y que deseas validar cada entrada de esa matriz:

- *YAML*

```
# src/UsuarioBundle/Resources/config/validation.yml
Acme\UsuarioBundle\Entity\Usuario:
    properties:
        favoriteColors:
            - All:
                - NotBlank:  ~
                - MinLength: 5
```

- *Annotations*

```
// src/Acme/UsuarioBundle/Entity/Usuario.php
namespace Acme\UsuarioBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Usuario
{
    /**
     * @Assert\All({
     *     @Assert\NotBlank
     *     @Assert\MinLength(5),
     * })
     */
     protected $coloresFavoritos = array();
}
```

Ahora, cada entrada en la matriz `coloresFavoritos` será validada para que no esté en blanco y que por lo menos sea de 5 caracteres.

Opciones

constraints

tipo: `array` [*opción predeterminada* (Página 154)]

Esta opción requerida es el arreglo de restricciones de validación que deseas aplicar a cada elemento de la matriz subyacente.

El Validador está diseñado para validar objetos contra *restricciones*. En la vida real, una restricción podría ser: "El pastel no se debe quemar". En *Symfony2*, las restricciones son similares: son aserciones de que una condición es verdadera.

4.11.30 Restricciones compatibles

Las siguientes restricciones están disponibles nativamente en *Symfony2*:

4.11.31 Restricciones básicas

Estas son las restricciones básicas: las utilizamos para afirmar cosas muy básicas sobre el valor de las propiedades o el valor de retorno de los métodos en tu objeto.

- *NotBlank* (Página 514)
- *Blank* (Página 515)
- *NotNull* (Página 516)
- *Null* (Página 517)
- *True* (Página 517)
- *False* (Página 519)
- *Type* (Página 521)

4.11.32 Restricciones de cadena

- *Email* (Página 522)
- *MinLength* (Página 523)
- *MaxLength* (Página 525)
- *Url* (Página 526)
- *Regex* (Página 527)
- *Ip* (Página 529)

4.11.33 Restricciones de número

- *Max* (Página 530)
- *Min* (Página 531)

4.11.34 Restricciones de fecha

- *Date* (Página 532)
- *DateTime* (Página 533)
- *Time* (Página 534)

4.11.35 Restricciones de colección

- *Choice* (Página 535)
- *Collection* (Página 539)
- *UniqueEntity* (Página 543)

- *Language* (Página 544)
- *Locale* (Página 545)
- *Country* (Página 546)

4.11.36 Restricciones de archivo

- *File* (Página 547)
- *Image* (Página 551)

4.11.37 Otras restricciones

- *Callback* (Página 551)
- *All* (Página 557)
- *Valid* (Página 554)

4.12 Etiquetas de inyección de dependencias

Etiquetas;

- `data_collector`
- `form.type`
- `form.type_extension`
- `form.type_guesser`
- `kernel.cache_warmer`
- `kernel.event_listener`
- `monolog.logger`
- `monolog.processor`
- `templating.helper`
- `routing.loader`
- `translation.loader`
- `twig.extension`
- `validator.initializer`

4.12.1 Habilitando ayudantes de plantilla *PHP* personalizados

Para habilitar un ayudante de plantilla, añádelo como un servicio regular en una de tus configuraciones, etiquétalo con `templating.helper` y define un atributo `alias` (el ayudante será accesible a través de este alias en las plantillas):

- *YAML*

```
services:
    templating.helper.nombre_de_tu_ayudante:
        class: Fully\Qualified\Helper\Class\Name
        tags:
            - { name: templating.helper, alias: nombre_alias }
```

- *XML*

```
<service id="templating.helper.nombre_de_tu_ayudante" class="Fully\Qualified\Helper\Class\Name">
    <tag name="templating.helper" alias="nombre_alias" />
</service>
```

- *PHP*

```
$contenedor
    ->register('templating.helper.nombre_de_tu_ayudante', 'Fully\Qualified\Helper\Class\Name')
    ->addTag('templating.helper', array('alias' => 'nombre_alias'))
;
```

4.12.2 Habilitando extensiones *Twig* personalizadas

Para habilitar una extensión de *Twig*, añádela como un servicio regular en una de tus configuraciones, y etiquétalo con `twig.extension`:

- *YAML*

```
services:
    twig.extension.nombre_de_tu_extensión:
        class: Fully\Qualified\Extension\Class\Name
        tags:
            - { name: twig.extension }
```

- *XML*

```
<service id="twig.extension.nombre_de_tu_extensión" class="Fully\Qualified\Extension\Class\Name"
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
$contenedor
    ->register('twig.extension.nombre_de_tu_extensión', 'Fully\Qualified\Extension\Class\Name')
    ->addTag('twig.extension')
;
```

4.12.3 Habilitando escuchas personalizados

Para habilitar un escucha personalizado, añádelo como un servicio regular en uno de tus archivos de configuración, y etiquétalo con `kernel.event_listener`. Debes proporcionar el nombre del evento a tu servicio escucha, así como el método que a invocar:

- *YAML*

```
services:
    kernel.listener.your_listener_name:
        class: Fully\Qualified\Listener\Class\Name
```

```
        tags:
            - { name: kernel.event_listener, event: xxx, method: onXxx }
```

- *XML*

```
<service id="kernel.listener.your_listener_name" class="Fully\Qualified\Listener\Class\Name">
    <tag name="kernel.event_listener" event="xxx" method="onXxx" />
</service>
```

- *PHP*

```
$contenedor
    ->register('kernel.listener.your_listener_name', 'Fully\Qualified\Listener\Class\Name')
    ->addTag('kernel.event_listener', array('event' => 'xxx', 'method' => 'onXxx'))
;
```

Nota: También puedes especificar la prioridad como un entero positivo o negativo, lo cual te permite asegurarte de que tu escucha siempre será invocado antes o después de otro.

4.12.4 Habilitando motores de plantilla personalizados

Para activar un motor de plantillas personalizado, añádelo como un servicio regular en una de tus configuraciones, etiquétalo con `templating.engine`:

- *YAML*

```
services:
    templating.engine.nombre_de_tu_motor:
        class: Fully\Qualified\Engine\Class\Name
        tags:
            - { name: templating.engine }
```

- *XML*

```
<service id="templating.engine.nombre_de_tu_motor" class="Fully\Qualified\Engine\Class\Name">
    <tag name="templating.engine" />
</service>
```

- *PHP*

```
$contenedor
    ->register('templating.engine.nombre_de_tu_motor', 'Fully\Qualified\Engine\Class\Name')
    ->addTag('templating.engine')
;
```

4.12.5 Habilitando cargadores de enrutado personalizados

Para habilitar un gestor de enrutado personalizado, añádelo como un servicio regular en una de tus configuraciones, y etiquétalo con `routing.loader`:

- *YAML*

```
services:
    routing.loader.nombre_de_tu_cargador:
        class: Fully\Qualified\Loader\Class\Name
```

```
        tags:
            - { name: routing.loader }
```

- *XML*

```
<service id="routing.loader.nombre_de_tu_cargador" class="Fully\Qualified\Loader\Class\Name">
    <tag name="routing.loader" />
</service>
```

- *PHP*

```
$contenedor
    ->register('routing.loader.nombre_de_tu_cargador', 'Fully\Qualified\Loader\Class\Name')
    ->addTag('routing.loader')
;
```

4.12.6 Usando canal de registro personalizado con `Monolog`

`Monolog` te permite compartir sus controladores entre varios canales registradores. El servicio registrador utiliza el canal `app` pero puedes cambiar de canal cuando inyectes el registrador en un servicio.

- *YAML*

```
services:
    my_service:
        class: Fully\Qualified\Loader\Class\Name
        arguments: [@logger]
        tags:
            - { name: monolog.logger, channel: acme }
```

- *XML*

```
<service id="my_service" class="Fully\Qualified\Loader\Class\Name">
    <argument type="service" id="logger" />
    <tag name="monolog.logger" channel="acme" />
</service>
```

- *PHP*

```
$definicion = new Definition('Fully\Qualified\Loader\Class\Name', array(new Reference('logger'))
$definicion->addTag('monolog.logger', array('channel' => 'acme'));
$contenedor->register('my_service', $definicion);;
```

Nota: Esto sólo funciona cuando el servicio registrador es un argumento del constructor, no cuando se inyecta a través de un definidor.

4.12.7 Agregando un procesador para `Monolog`

`Monolog` te permite agregar procesadores en el registrador o en los controladores para añadir datos adicionales en los registros. Un procesador recibe el registro como argumento y lo tiene que devolver después de añadir alguna información adicional en el atributo `extra` del registro.

Vamos a ver cómo puedes utilizar el `IntrospectionProcessor` integrado para agregar el archivo, la línea, la clase y el método en que se activó el registrador.

Puedes agregar un procesador globalmente:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" />
</service>
```

- *PHP*

```
$definicion = new Definition('Monolog\Processor\IntrospectionProcessor');
$definicion->addTag('monolog.processor');
$contenedor->register('my_service', $definicion);
```

Truco: Si el servicio no es ejecutable (con `__invoke`) puedes agregar el atributo `method` en la etiqueta para utilizar un método específico.

También puedes agregar un procesador para un controlador específico utilizando el atributo `handler`:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, handler: firephp }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" handler="firephp" />
</service>
```

- *PHP*

```
$definicion = new Definition('Monolog\Processor\IntrospectionProcessor');
$definicion->addTag('monolog.processor', array('handler' => 'firephp');
$contenedor->register('my_service', $definicion);
```

También puedes agregar un procesador para un canal registrador específico usando el atributo `channel`. Esto registrará el procesador únicamente para el canal registrador `security` utilizado en el componente de seguridad:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, channel: security }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" channel="security" />
</service>
```

- *PHP*

```
$definicion = new Definition('Monolog\Processor\IntrospectionProcessor');
$definicion->addTag('monolog.processor', array('channel' => 'security');
$contenedor->register('my_service', $definicion);
```

Nota: No puedes utilizar ambos atributos `handler` y `channel` para la misma etiqueta debido a que los controladores son compartidos entre todos los canales.

4.13 YAML

La página web de YAML dice que es "un estándar para la serialización de datos humanamente legible para todos los lenguajes de programación". YAML es un lenguaje simple que describe datos. Como PHP, que tiene una sintaxis para tipos simples, como cadenas, booleanos, reales, o enteros. Pero a diferencia de *PHP*, que distingue entre matrices (secuencias) y `hashes` (asignaciones).

El **:namespace:'Symfony\\Component\\Yaml'** Componente *Symfony2* sabe cómo analizar *YAML* y volcar una matriz *PHP* a *YAML*.

Nota: Si bien el formato *YAML* puede describir complejas estructuras anidadas de datos, este capítulo sólo describe el conjunto mínimo de características necesarias para usar YAML como un formato de archivo de configuración.

4.13.1 Leyendo archivos YAML

El método **:method:'Symfony\\Component\\Yaml\\Parser::parse'** analiza una cadena YAML y la convierte en una matriz de *PHP*:

```
use Symfony\Component\Yaml\Parser;

$yaml = new Parser();
$valor = $yaml->parse(file_get_contents('/ruta/a/file.yaml'));
```

Si se produce un error durante el análisis, el analizador emite una excepción que indica el tipo de error y la línea en la cadena original YAML donde ocurrió el error:

```
try {
    $valor = $yaml->parse(file_get_contents('/ruta/a/file.yaml'));
} catch (\InvalidArgumentException $e) {
    // ocurrió un error durante el análisis
    echo "Unable to parse the YAML string: ".$e->getMessage();
}
```

Truco: Debido a que el analizador es reentrante, puedes utilizar el mismo objeto analizador para cargar diferentes cadenas YAML.

Al cargar un archivo *YAML*, a veces es mejor usar el contenedor del método **:method:'Symfony\\Component\\Yaml\\Yaml::load'**:

```
use Symfony\Component\Yaml\Yaml;

$cargador = Yaml::parse('/ruta/al/archivo.yml');
```

El método estático `Yaml::parse()` toma una cadena YAML o un archivo que contiene YAML. Internamente, este llama al método `Parser::parse()`, pero con algunas cosas adicionales como:

- Ejecuta el archivo *YAML* como si fuera un archivo *PHP*, por lo tanto puedes incrustar ordenes PHP en los archivos YAML;
- Cuando un archivo no se puede analizar, este agrega automáticamente el nombre del archivo al mensaje de error, lo cual simplifica la depuración cuando tu aplicación está cargando varios archivos YAML.

4.13.2 Escribiendo archivos YAML

El método **:method:'Symfony\\Component\\Yaml\\Dumper::dump'** vierte cualquier matriz *PHP* en su representación de YAML:

```
use Symfony\Component\Yaml\Dumper;

$array = array('foo' => 'bar', 'bar' => array('foo' => 'bar', 'bar' => 'baz'));

$dumper = new Dumper();
$yaml = $dumper->dump($array);
file_put_contents('/ruta/a/file.yaml', $yaml);
```

Nota: Hay algunas limitaciones: el vertedor no es capaz de volcar los recursos y verter objetos PHP se considera una característica alfa.

Si sólo necesitas volcar una matriz, puedes utilizar el método estático **:method:'Symfony\\Component\\Yaml\\Yaml::dump'**:

```
$yaml = Yaml::dump($array, $inline);
```

El formato *YAML* apoya las dos representaciones de matriz YAML. De forma predeterminada, el vertedor utiliza la representación en línea:

```
{ foo: bar, bar: { foo: bar, bar: baz } }
```

Pero el segundo argumento del método `dump()` personaliza el nivel en el cual la salida cambia de la representación expandida a la de en línea:

```
echo $dumper->dump($array, 1);

foo: bar
bar: { foo: bar, bar: baz }

echo $dumper->dump($array, 2);

foo: bar
bar:
    foo: bar
    bar: baz
```

4.13.3 La sintaxis YAML

Cadenas

```
Una cadena en YAML
```

```
'Una cadena entre comillas simples en YAML'
```

Truco: En una cadena entre comillas simples, una comilla simple `'` debe ser doble:

```
'Una comilla simple '' en una cadena entre comillas simples'
```

```
"Una cadena entre comillas dobles en YAML\n"
```

Los estilos de citado son útiles cuando una cadena empieza o termina con uno o más espacios relevantes.

Truco: El estilo entre comillas dobles proporciona una manera de expresar cadenas arbitrarias, utilizando secuencias de escape `\`. Es muy útil cuando se necesita incrustar un `\n` o un carácter Unicode en una cadena.

Cuando una cadena contiene saltos de línea, puedes usar el estilo literal, indicado por la tubería (`|`), para indicar que la cadena tendrá una duración de varias líneas. En literales, se conservan los saltos de línea:

```
|
  \/ /| |\/| |
  / / | |  | |__
```

Alternativamente, puedes escribir cadenas con el estilo de plegado, denotado por >, en el cual cada salto de línea es sustituido por un espacio:

```
>
  Esta es una oración muy larga
  que se extiende por varias líneas en el YAML
  pero que se reproduce como una cadena
  sin retornos de carro.
```

Nota: Observa los dos espacios antes de cada línea en los ejemplos anteriores. Ellos no aparecen en las cadenas PHP resultantes.

Números

```
# un entero
12

# un octal
014

# un hexadecimal
0xC

# un float
13.4
```

```
# un número exponencial
1.2e+34

# infinito
.inf
```

Nulos

Los nulos en YAML se pueden expresar con `null` o `~`.

Booleanos

Los booleanos en YAML se expresan con `true` y `false`.

fechas

YAML utiliza la norma ISO-8601 para expresar fechas:

```
2001-12-14t21:59:43.10-05:00

# fecha simple
2002-12-14
```

Colecciones

Rara vez se utiliza un archivo *YAML* para describir un simple escalar. La mayoría de las veces, describe una colección. Una colección puede ser una secuencia o una asignación de elementos. Ambas, secuencias y asignaciones se convierten en matrices PHP.

Las secuencias usan un guión seguido de un espacio (`-`):

```
- PHP
- Perl
- Python
```

El archivo *YAML* anterior es equivalente al siguiente código *PHP*:

```
array('PHP', 'Perl', 'Python');
```

Las asignaciones usan dos puntos (`:`) seguidos de un espacio para marcar cada par clave/valor:

```
PHP: 5.2
MySQL: 5.1
Apache: 2.2.20
```

el cual es equivalente a este código *PHP*:

```
array('PHP' => 5.2, 'MySQL' => 5.1, 'Apache' => '2.2.20');
```

Nota: En una asignación, una clave puede ser cualquier escalar válido.

El número de espacios entre los dos puntos y el valor no importa:

```
PHP:    5.2
MySQL:  5.1
Apache: 2.2.20
```

YAML utiliza sangría con uno o más espacios para describir colecciones anidadas:

```
"symfony 1.4":
    PHP:      5.2
    Doctrine: 1.2
"*Symfony2*":
    PHP:      5.3
    Doctrine: 2.0
```

El YAML anterior es equivalente al siguiente código *PHP*:

```
array(
    'symfony 1.4' => array(
        'PHP'      => 5.2,
        'Doctrine' => 1.2,
    ),
    'Symfony2' => array(
        'PHP'      => 5.3,
        'Doctrine' => 2.0,
    ),
);
```

Hay una cosa importante que tienes que recordar cuando utilices sangría en un archivo *YAML*: *La sangría se debe hacer con uno o más espacios, pero nunca con tabulaciones*.

Puedes anidar secuencias y asignaciones a tu gusto:

```
'Chapter 1':
    - Introduction
    - Event Types
'Chapter 2':
    - Introduction
    - Helpers
```

YAML también puedes utilizar estilos de flujo para colecciones, utilizando indicadores explícitos en lugar de sangría para denotar el alcance.

Puedes escribir una secuencia como una lista separada por comas entre corchetes (`[]`):

```
[PHP, Perl, Python]
```

Puedes escribir una asignación como una lista separada por comas de clave/valor dentro de llaves (`{}`):

```
{ PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

Puedes mezclar y combinar estilos para conseguir mayor legibilidad:

```
'Chapter 1': [Introduction, Event Types]
'Chapter 2': [Introduction, Helpers]

"symfony 1.4": { PHP: 5.2, Doctrine: 1.2 }
"*Symfony2*":    { PHP: 5.3, Doctrine: 2.0 }
```

Comentarios

En *YAML* puedes añadir comentarios prefijando con una almohadilla (`#`):

```
# Comentario en una línea
"*Symfony2*": { PHP: 5.3, Doctrine: 2.0 } # Comentario al final de la línea
```

Nota: Los comentarios simplemente son ignorados por el analizador de YAML y no necesitan sangría de acuerdo al nivel de anidamiento actual de una colección.

Archivos dinámicos YAML

En *Symfony2*, un archivo *YAML* puede contener código *PHP* que se evalúa justo antes de que ocurra el análisis:

```
1.0:
    version: <?php echo file_get_contents('1.0/VERSION')."\n" ?>
1.1:
    version: "<?php echo file_get_contents('1.1/VERSION') ?>"
```

Ten cuidado de no estropearlo con la sangría. Ten en cuenta los siguientes consejos sencillos al añadir código *PHP* en un archivo *YAML*:

- La declaración `<?php ?>` siempre debe comenzar la línea o estar integrada en un valor.
- Si una declaración `<?php ?>` termina una línea, necesitas producir una nueva línea ("n") de manera explícita.

4.14 Requisitos para que funcione *Symfony2*

Para ejecutar *Symfony2*, el sistema debe cumplir con una lista de requisitos. Fácilmente puedes ver si el sistema pasa todos los requisitos ejecutando `web/config.php` en tu distribución de *Symfony*. Debido a que el CLI a menudo utiliza un archivo de configuración `php.ini` diferente, también es una buena idea revisar tus requisitos desde la línea de ordenes por medio de:

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

A continuación mostramos la lista de requisitos obligatorios y opcionales.

4.14.1 Obligatorio

- *PHP* debe ser una versión mínima de *PHP 5.3.2*
- Tu php.ini debe tener configurado el valor date.timezone

4.14.2 Opcional

- Necesitas tener instalado el módulo *PHP-XML*
- Debes tener por lo menos la versión 2.6.21 de `libxml`
- Debe estar habilitado el segmentador de *PHP*
- Las funciones `mbstring` deben estar habilitadas
- `iconv` debe estar habilitada
- *POSIX* debe estar habilitado
- `Intl` tiene que estar instalado

- *APC* (u otro opcode de caché debe estar instalado)
- `php.ini` con la configuración recomendada
  - short_open_tags: off
  - magic_quotes_gpc: off
  - register_globals: off
  - session.autostart: off

4.14.3 *Doctrine*

Si deseas utilizar *Doctrine*, necesitarás tener instalado PDO. Además, es necesario tener instalado el controlador de PDO para el servidor de base de datos que desees utilizar.

- **Opciones de configuración**:

  Alguna vez te preguntaste ¿qué opciones de configuración tengo a mi disposición en archivos como `app/config/config.yml`? En esta sección, toda la configuración disponible separadas por claves (por ejemplo, `framework`) que define cada parte susceptible de configuración de tu *Symfony2*.

  - *framework* (Página 443)
  - *doctrine* (Página 447)
  - *security* (Página 452)
  - *assetic* (Página 446)
  - *swiftmailer* (Página 455)
  - *twig* (Página 455)
  - *monolog* (Página 457)
  - *web_profiler* (Página 458)

- **Formularios y Validación**
  - *Referencia del campo tipo Form* (Página 459)
  - *Referencia de restricciones de validación* (Página 514)
  - *Referencia de las funciones de plantilla Twig* (Página 513)

- **Otras áreas**
  - *Etiquetas de inyección de dependencias* (Página 560)
  - *YAML* (Página 565)
  - *Requisitos para que funcione Symfony2* (Página 570)

- **Opciones de configuración**:

  Alguna vez te preguntaste ¿qué opciones de configuración tengo a mi disposición en archivos como `app/config/config.yml`? En esta sección, toda la configuración disponible separadas por claves (por ejemplo, `framework`) que define cada parte susceptible de configuración de tu *Symfony2*.

  - *framework* (Página 443)
  - *doctrine* (Página 447)
  - *security* (Página 452)
  - *assetic* (Página 446)

- *swiftmailer* (Página 455)
- *twig* (Página 455)
- *monolog* (Página 457)
- *web_profiler* (Página 458)

- **Formularios y Validación**
  - *Referencia del campo tipo Form* (Página 459)
  - *Referencia de restricciones de validación* (Página 514)
  - *Referencia de las funciones de plantilla Twig* (Página 513)
- **Otras áreas**
  - *Etiquetas de inyección de dependencias* (Página 560)
  - *YAML* (Página 565)
  - *Requisitos para que funcione Symfony2* (Página 570)

Parte V

Paquetes

La *Edición estándar de Symfony2* viene con una configuración de seguridad sencilla adaptada a las necesidades más comunes. Aprende más acerca de ellas:

Paquetes SE de *Symfony*

5.1 SensioFrameworkExtraBundle

El `FrameworkBundle` predeterminado de *Symfony2* implementa una plataforma *MVC*, básica pero robusta y flexible. SensioFrameworkExtraBundle la extiende añadiendo agradables convenciones y anotaciones. Esto permite controladores más concisos.

5.1.1 Instalando

Descarga el paquete y ponlo bajo el espacio de nombres `Sensio\Bundle\`. Luego, como con cualquier otro paquete, inclúyelo en la clase de tu núcleo:

```
public function registerBundles()
{
    $bundles = array(
        ...

        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
    );

    ...
}
```

5.1.2 Configurando

Todas las características proporcionadas por el paquete están habilitadas por omisión, cuando registras el paquete en la clase de tu núcleo.

La configuración predeterminada es la siguiente:

- *YAML*

```
sensio_framework_extra:
    router:  { annotations: true }
    request: { converters: true }
    view:    { annotations: true }
    cache:   { annotations: true }
```

- *XML*

```
<!-- xmlns:sensio-framework-extra="http://www.symfony-project.org/schema/dic/sensio-framework-ex
<sensio-framework-extra:config>
    <router annotations="true" />
    <request converters="true" />
    <view annotations="true" />
    <cache annotations="true" />
</sensio-framework-extra:config>
```

- *PHP*

```
// Carga el perfilador
$contenedor->loadFromExtension('sensio_framework_extra', array(
    'router'  => array('annotations' => true),
    'request' => array('converters' => true),
    'view'    => array('annotations' => true),
    'cache'   => array('annotations' => true),
));
```

Puedes desactivar algunas anotaciones y convenciones definiendo uno o más valores en `false`.

5.1.3 Anotaciones para controladores

Las anotaciones son una buena manera de configurar controladores fácilmente, desde las rutas hasta la configuración de la caché.

Incluso si las anotaciones no son una característica natural de *PHP*, aún tienen varias ventajas sobre los métodos de configuración clásicos de *Symfony2*:

- El código y la configuración están en el mismo lugar (la clase controlador)
- Fácil de aprender y usar;
- Concisas para escribir;
- Adelgazan tu controlador (puesto que su única responsabilidad es conseguir los datos del modelo).

Truco: Si utilizas las clases `vista`, las anotaciones son una buena manera de evitar la creación de clases `vista` para casos simples y comunes.

Las siguientes anotaciones están definidas por el paquete:

`@Route` y `@Method`

Uso

La anotación `@Route` asigna un patrón de ruta a un controlador:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class ComunicadoController extends Controller
{
    /**
     * @Route("/")
     */
    public function indexAction()
```

```
    {
        // ...
    }
}
```

La acción `index` del controlador de `Comunicado` ya está asignada a la dirección `/`. Esto es equivalente a la siguiente configuración *YAML*:

```
hogar_blog:
    pattern:  /
    defaults: { _controller: SensioBlogBundle:Comunicado:index }
```

Al igual que cualquier patrón de ruta, puedes definir marcadores de posición, requisitos y valores predeterminados:

```
/**
 * @Route("/{id}", requirements={"id" = "\d+"}, defaults={"foo" = "bar"})
 */
public function showAction($id)
{
}
```

También puedes combinar más de una *URL* definiendo una anotación `@Route` adicional:

```
/**
 * @Route("/", defaults={"id" = 1})
 * @Route("/{id}")
 */
public function showAction($id)
{
}
```

Activación

Las rutas se deben importar para estar activas como cualquier otro recurso de enrutado (observa el tipo `annotation`):

```
# app/config/routing.yml

# importa las rutas desde una clase controlador
comunicado:
    resource: "@SensioBlogBundle/Controller/ComunicadoController.php"
    type:     annotation
```

También puedes importar un directorio completo:

```
# importa las rutas desde un directorio de controladores
blog:
    resource: "@SensioBlogBundle/Controller"
    type:     annotation
```

Como para cualquier otro recurso, puedes "montar" las rutas bajo un determinado prefijo:

```
comunicado:
    resource: "@SensioBlogBundle/Controller/ComunicadoController.php"
    prefix:   /blog
    type:     annotation
```

Nombre de ruta

A una ruta definida con la anotación `@Route` se le asigna un nombre predeterminado, el cual está compuesto por el nombre del paquete, el nombre del controlador y el nombre de la acción. En el caso del ejemplo anterior sería `sensio_blog_comunicado_index`;

Puedes utilizar el atributo `name` para reemplazar este nombre de ruta predeterminado:

```
/**
 * @Route("/", name="hogar_blog")
 */
public function indexAction()
{
    // ...
}
```

Prefijo de ruta

Una anotación `@Route` en una clase controlador define un prefijo para todas las rutas de acción:

```
/**
 * @Route("/blog")
 */
class ComunicadoController extends Controller
{
    /**
     * @Route("/{id}")
     */
    public function showAction($id)
    {
    }
}
```

La acción `show` ahora se asigna al patrón `/blog/{id}`.

Método de la ruta

Hay un atajo en la anotación `@Method` para especificar el método *HTTP* permitido para la ruta. Para usarlo, importa el espacio de nombres de la anotación `Method`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;

/**
 * @Route("/blog")
 */
class ComunicadoController extends Controller
{
    /**
     * @Route("/editar/{id}")
     * @Method({"GET", "POST"})
     */
    public function editarAction($id)
    {
    }
}
```

La acción `editar` ahora es asignada al patrón `/blog/editar/{id}` si el método *HTTP* utilizado es *GET* o *POST*.

La anotación `@Method` sólo se toma en cuenta cuando una acción se anota con `@Route`.

@ParamConverter

Uso

La anotación `@ParamConverter` llama a `converters` para convertir parámetros de la petición a objetos. Estos objetos se almacenan como atributos de la petición y por lo tanto se puedan inyectar en los argumentos del método controlador:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @Route("/blog/{id}")
 * @ParamConverter("comunicado", class="SensioBlogBundle:Comunicado")
 */
public function showAction(Comunicado $comunicado)
{
}
```

Suceden varias cosas bajo el capó:

- El convertidor intenta obtener un objeto `SensioBlogBundle:Comunicado` desde los atributos de la petición (los atributos de la petición provienen de los marcadores de posición de la ruta - aquí `id`);
- Si no se encontró algún objeto `comunicado`, se genera una respuesta `404`;
- Si se encuentra un objeto `Comunicado`, se define un nuevo atributo `comunicado` para la petición (accesible a través de `$peticion->attributes->get('comunicado')`);
- Al igual que cualquier otro atributo de la petición, este se inyecta automáticamente en el controlador cuando está presente en la firma del método.

Si utilizas el tipo como el del ejemplo anterior, incluso puedes omitir la anotación `@ParamConverter` por completo:

```
// automático con la firma del método
public function showAction(Comunicado $comunicado)
{
}
```

Convertidores integrados

El paquete tiene un solo convertidor integrado, uno de *Doctrine*.

Convertidor *Doctrine* De manera predeterminada, el convertidor de *Doctrine* utiliza el valor *predeterminado* del administrador de la entidad. Este se puede configurar con la opción `entity_manager`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @Route("/blog/{id}")
 * @ParamConverter("comunicado", class="SensioBlogBundle:Comunicado", options={"entity_manager" = "fo
 */
```

```
public function showAction(Comunicado $comunicado)
{
}
```

Creando un convertidor

Todos los convertidores deben implementar la `Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter\Pa`

```
namespace Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ConfigurationInterface;
use Symfony\Component\HttpFoundation\Request;

interface ParamConverterInterface
{
    function apply(Request $peticion, ConfigurationInterface $configuracion);

    function supports(ConfigurationInterface $configuración);
}
```

El método `supports()` debe devolver `true` cuando sea capaz de convertir la configuración dada (una instancia de `ParamConverter`).

La instancia de `ParamConverter` tiene tres piezas de información sobre la anotación:

- `name`: El atributo nombre;
- `class`: El atributo nombre de la clase (puede ser cualquier cadena que represente el nombre de la clase);
- `options`: Un arreglo de opciones

El método `apply()` se llama cuando una configuración es compatible. Basándonos en los atributos de la petición, debemos establecer un atributo llamado `$configuracion->getName()`, que almacene un objeto de la clase `$configuracion->getClass()`.

Truco: Utiliza la clase `DoctrineConverter` como plantilla para tus propios convertidores.

`@Template`

Uso

La anotación `@Template` asocia un controlador con un nombre de plantilla:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Template("SensioBlogBundle:Comunicado:show")
 */
public function showAction($id)
{
    // consigue el Comunicado
    $comunicado = ...;

    return array('comunicado' => $comunicado);
}
```

Cuando utilizas la anotación `@Template`, el controlador debe devolver un arreglo de parámetros para pasarlo a la vista en lugar de un objeto `Respuesta`.

Truco: Si la acción devuelve un objeto `Respuesta`, la anotación `@Template` simplemente se omite.

Si la plantilla se nombra después del controlador y los nombres de acción, como en el caso del ejemplo anterior, puedes omitir incluso el valor de la anotación:

```
/**
 * @Template
 */
public function showAction($id)
{
    // Consigue el Comunicado
    $comunicado = ...;

    return array('comunicado' => $comunicado);
}
```

Y si los únicos parámetros para pasar a la plantilla son los argumentos del método, puedes utilizar el atributo `vars` en lugar de devolver un arreglo. Esto es muy útil en combinación con la *anotación @ParamConverter* (Página 581):

```
/**
 * @ParamConverter("comunicado", class="SensioBlogBundle:Comunicado")
 * @Template("SensioBlogBundle:Comunicado:show", vars={"comunicado"})
 */
public function showAction(Comunicado $comunicado)
{
}
```

que, gracias a las convenciones, es equivalente a la siguiente configuración:

```
/**
 * @Template(vars={"comunicado"})
 */
public function showAction(Comunicado $comunicado)
{
}
```

Puedes hacer que sea aún más conciso puesto que todos los argumentos del método se pasan automáticamente a la plantilla si el método devuelve `null` y no se define el atributo `vars`:

```
/**
 * @Template
 */
public function showAction(Comunicado $comunicado)
{
}
```

@Cache

Uso

La anotación `@cache` facilita la definición del almacenamiento en caché *HTTP*:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;

/**
 * @Cache(expires="tomorrow")
 */
public function indexAction()
{
}
```

También puedes utilizar la anotación en una clase para definir el almacenamiento en caché para todos sus métodos:

```
/**
 * @Cache(expires="tomorrow")
 */
class BlogController extends Controller
{
}
```

Cuando hay un conflicto entre la configuración de la clase y la configuración del método, esta última reemplaza a la anterior:

```
/**
 * @Cache(expires="tomorrow")
 */
class BlogController extends Controller
{
    /**
     * @Cache(expires="+2 days")
     */
    public function indexAction()
    {
    }
}
```

Atributos

Aquí está una lista de los atributos aceptados y su encabezado *HTTP* equivalente:

Anotación	Método de respuesta
`@Cache(expires="tomorrow")`	`$respuesta->setExpires()`
`@Cache(smaxage="15")`	`$respuesta->setSharedMaxAge()`
`@Cache(maxage="15")`	`$respuesta->setMaxAge()`

Nota: El atributo `expires` toma cualquier fecha válida entendida por la función `strtotime()` de *PHP*.

Este ejemplo muestra en acción todas las anotaciones disponibles:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @Route("/blog")
 * @Cache(expires="tomorrow")
 */
class AnotacionController extends Controller
```

```
{
    /**
     * @Route("/")
     * @Template
     */
    public function indexAction()
    {
        $comunicados = ...;

        return array('comunicados' => $comunicados);
    }

    /**
     * @Route("/{id}")
     * @Method("GET")
     * @ParamConverter("comunicado", class="SensioBlogBundle:Comunicado")
     * @Template("SensioBlogBundle:Anotacion:comunicado", vars={"comunicado"})
     * @Cache(smaxage="15")
     */
    public function showAction(Comunicado $comunicado)
    {
    }
}
```

En la medida en que el método `showAction` siga algunas convenciones, puedes omitir algunas anotaciones:

```
/**
 * @Route("/{id}")
 * @Cache(smaxage="15")
 */
public function showAction(Comunicado $comunicado)
{
}
```

5.2 SensioGeneratorBundle

El paquete `SensioGeneratorBundle` extiende la interfaz de la línea de ordenes predeterminada de *Symfony2*, ofreciendo nuevas ordenes interactivas e intuitivas para generar esqueletos de código como paquetes, clases de formulario o controladores `CRUD` basados en un esquema de *Doctrine*.

5.2.1 Instalando

Descarga el paquete y ponlo bajo el espacio de nombres `Sensio\Bundle\`. Luego, como con cualquier otro paquete, inclúyelo en tu clase núcleo:

```
public function registerBundles()
{
    $bundles = array(
        ...

        new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle(),
    );

    ...
}
```

5.2.2 Lista de ordenes disponibles

El paquete `SensioGeneratorBundle` vienen con cuatro nuevas ordenes que puedes ejecutar en modo interactivo o no. El modo interactivo te hace algunas preguntas para configurar los parámetros para que la orden genere el código definitivo. La lista de nuevas ordenes se enumera a continuación:

Generando el esqueleto para un nuevo paquete

Uso

El `generate:bundle` genera una estructura para un nuevo paquete y lo activa automáticamente en la aplicación.

De manera predeterminada, la orden se ejecuta en modo interactivo y hace preguntas para determinar el nombre del paquete, ubicación, formato y estructura de configuración predeterminada:

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

Para desactivar el modo interactivo, utiliza la opción `--no-interaction` pero no olvides suministrar todas las opciones necesarias:

```
php app/console generate:bundle --namespace=Acme/Bundle/BlogBundle --no-interaction
```

Opciones disponibles

- `--namespace`: El espacio de nombres a crear para el paquete. El espacio de nombres debe comenzar con un nombre de "proveedor", tal como el nombre de tu empresa, el nombre de tu proyecto, o el nombre de tu cliente, seguido por uno o más subespacio de nombres para una categoría opcional, el cual debe terminar con el nombre del paquete en sí mismo (debe tener `Bundle` como sufijo):

  ```
  php app/console generate:bundle --namespace=Acme/Bundle/BlogBundle
  ```

- `--bundle-name`: El nombre opcional del paquete. Esta debe ser una cadena que termine con el sufijo `Bundle`:

  ```
  php app/console generate:bundle --bundle-name=AcmeBlogBundle
  ```

- `--dir`: El directorio en el cual guardar el paquete. Por convención, la orden detecta y utiliza el directorio `src/` de tu aplicación:

  ```
  php app/console generate:bundle --dir=/var/www/miproyecto/src
  ```

- `--format`: (**annotation**) [valores: yml, xml, php o annotation] Determina el formato de enrutado a usar para los archivos de configuración generados. De manera predeterminada, la orden utiliza el formato de `anotación`. Elegir el formato de `anotación` espera que el paquete''SensioFrameworkExtraBundle'' ya esté instalado:

  ```
  php app/console generate:bundle --format=annotation
  ```

- `--structure`: (**no**) [valores: yes|no] Cuando o no generar una estructura de directorios completa incluyendo directorios públicos vacíos para documentación, activos web y diccionarios de traducción:

  ```
  php app/console generate:bundle --structure=yes
  ```

Generando un controlador `CRUD` basado en una entidad de *Doctrine*

Uso

La orden `generate:doctrine:crud` genera un controlador básico para una determinada entidad ubicada en un determinado paquete. Este controlador te permite realizar cinco operaciones básicas en un modelo.

- Listar todos los registros,
- Mostrar un determinado registro identificado por su clave primaria,
- Crear un nuevo registro,
- Editar un registro existente,
- Eliminar un registro existente.

De manera predeterminada, la orden se ejecuta en modo interactivo y hace preguntas para determinar el nombre de la entidad, el prefijo de la ruta o cuando o no generar acciones de escritura:

```
php app/console generate:doctrine:crud
```

Para desactivar el modo interactivo, utiliza la opción `--no-interaction` pero no olvides suministrar todas las opciones necesarias:

```
php app/console generate:doctrine:crud --entity=AcmeBlogBundle:Comunicado --format=annotation --with-
```

Opciones disponibles

- `--entity`: El nombre de la entidad dado en forma atajo de notación que contiene el nombre del paquete en el que se encuentra la entidad y el nombre de la entidad. Por ejemplo: `AcmeBlogBundle:Comunicado`:

  ```
  php app/console generate:doctrine:crud --entity=AcmeBlogBundle:Comunicado
  ```

- `--route-prefix`: El prefijo que se utilizará para cada ruta que identifica una acción:

  ```
  php app/console generate:doctrine:crud --route-prefix=acme_comunicado
  ```

- `--with-write`: (**no**) [valores: yes|no] Cuando o no generar las acciones `new`, `create`, `edit`, `update` y `delete`:

  ```
  php app/console generate:doctrine:crud --with-write
  ```

- `--format`: (**annotation**) [valores: yml, xml, php o annotation] Determina el formato de enrutado a usar para los archivos de configuración generados. De manera predeterminada, la orden utiliza el formato de `anotación`. Elegir el formato de `anotación` espera que el paquete‘‘SensioFrameworkExtraBundle‘‘ ya esté instalado:

  ```
  php app/console generate:doctrine:crud --format=annotation
  ```

Generando una nueva entidad de resguardo *Doctrine*

Uso

La orden `generate:doctrine:entity` genera una nueva entidad de resguardo *Doctrine*, incluyendo la definición de asignaciones y propiedades de clase, captadores y definidores.

De manera predeterminada, la orden se ejecuta en modo interactivo y hace preguntas para determinar el nombre del paquete, ubicación, formato y estructura de configuración predeterminada:

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

La orden se puede ejecutar en modo no interactivo usando la opción `--non-interaction` sin olvidar todas las opciones necesarias:

```
php app/console generate:doctrine:entity --non-interaction --entity=AcmeBlogBundle:Comunicado --field
```

Opciones disponibles

- `--entity`: El nombre de la entidad dado en forma atajo de notación que contiene el nombre del paquete en el que se encuentra la entidad y el nombre de la entidad. Por ejemplo: `AcmeBlogBundle:Comunicado`:

  ```
  php app/console generate:doctrine:entity --entity=AcmeBlogBundle:Comunicado
  ```

- `--fields`: La lista de campos a generar en la clase entidad:

  ```
  php app/console generate:doctrine:entity --fields="titulo:string(100) cuerpo:text"
  ```

- `--format`: (**annotation**) [valores: yml, xml, php o annotation] Esta opción determina el formato a usar para los archivos de configuración generados como enrutado. De manera predeterminada, la orden utiliza el formato `anotación`:

  ```
  php app/console generate:doctrine:entity --format=annotation
  ```

- `--with-repository`: Esta opción indica si se debe o no generar la clase *Doctrine* relacionada con `EntityRepository`:

  ```
  php app/console generate:doctrine:entity --with-repository
  ```

Generando una nueva clase de tipo *Form* basada en una entidad *Doctrine*

Uso

La orden `generate:doctrine:form` genera una clase de tipo `form` básica usando los metadatos de asignación de una determinada clase `entidad`:

```
php app/console generate:doctrine:form AcmeBlogBundle:Comunicado
```

Argumentos obligatorios

- `entity`: El nombre de la entidad dado en forma atajo de notación que contiene el nombre del paquete en el que se encuentra la entidad y el nombre de la entidad. Por ejemplo: `AcmeBlogBundle:Comunicado`:

  ```
  php app/console generate:doctrine:form AcmeBlogBundle:Comunicado
  ```

5.3 Descripción

Este paquete te permite proteger con anotaciones las llamadas a métodos a nivel de servicio.

Generalmente puedes salvaguardar todos los métodos públicos, protegidos que no son estáticos, y no son finales. Los métodos privados no se pueden proteger de esta manera.

También puedes declarar anotaciones en métodos abstractos, clases padre, o interfaces.

5.3.1 ¿Cómo funciona?

El paquete primero debe recolectar desde las anotaciones todos los metadatos de seguridad disponibles para tus servicios. Los metadatos serán utilizados para construir las clases delegadas que tienen incorporados los controles de seguridad solicitados. Estas clases delegadas sustituirán a tu clase Servicio original. Todo esto se hace automáticamente para ti, no es necesario borrar manualmente ninguna caché si realizas cambios en los metadatos.

5.3.2 Rendimiento

Aunque prácticamente no habrá ninguna diferencia de rendimiento en tu entorno de producción, el rendimiento en el entorno de desarrollo depende significativamente de la configuración (consulta la sección de configuración).

En general, encontrarás que cuando cambias los archivos de un servicio protegido al cargar la primer página después de cambiar el archivo irá en aumento. Esto es porque la caché de este servicio se tendrá que reconstruir, y posiblemente, se deba generar una clase delegada. La subsecuente carga de la página será muy rápida.

5.3.3 Instalando

Consigue una copia del código:

```
git submodule add https://github.com/schmittjoh/SecurityExtraBundle.git vendor/bundles/JMS/SecurityEx
```

Luego registra el paquete en tu núcleo:

```
// en AppKernel::registerBundles()
$bundles = array(
    // ...
    new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    // ...
);
```

Además, este paquete necesita la biblioteca `Metadata`:

```
git submodule add https://github.com/schmittjoh/metadata.git vendor/metadata
```

Asegúrate de registrar los espacios de nombres en tu cargador automático:

```
// app/autoload.php
$cargador->registerNamespaces(array(
    // ...
    'JMS'              => __DIR__.'/../vendor/bundles',
    'Metadata'         => __DIR__.'/../vendor/metadata/src',
    // ...
));
```

5.3.4 Configurando

A continuación, se encuentra la configuración predeterminada:

```
# app/config/config.yml
jms_security_extra:
    # Si cinfiguras a tus controladores como servicios, lo debes fijar a false;
    # de lo contrario las comprobaciones de seguridad se realizaran dos veces.
    secure_controllers: true

    # Si quieres proteger todos tus servicios (true), o únicamente asegurar
    # servicios específicos (false); ve más abajo
    secure_all_services: false

    # Habilitar esta opción agregará un atributo especial "IS_IDDQD" adicional.
    # Cualquiera con este atributo efectivamente evadirá todas las comprobaciones de seguridad.
    enable_iddqd_attribute: false
```

De manera predeterminada, las comprobaciones de seguridad no están habilitadas para ningún servicio. Puedes activar la seguridad para tus servicios, ya sea protegiendo todos los servicios como se muestra arriba, o únicamente determinados servicios añadiendo una etiqueta a esos servicios:

```
<service id="foo" class="Bar">
    <tag name="security.secure_service"/>
</service>
```

Si habilitas la seguridad para todos los servicios, ten en cuenta que la primera vez que cargues la primer será muy lenta dependiendo de la cantidad de servicios que hayas definido.

5.3.5 Anotaciones

@Secure

Esta anotación te permite definir quién está autorizado a invocar un método:

```
<?php

use JMS\SecurityExtraBundle\Annotation\Secure;

class MiServicio
{
    /**
     * @Secure(roles="ROLE_USER, ROLE_FOO, ROLE_ADMIN")
     */
    public function metodoProtegido()
    {
        // ...
    }
}
```

@SecureParam

Esta anotación te permite definir restricciones para los parámetros que se le pasan al método. Esto sólo es útil si los parámetros son objetos del dominio:

```
<?php

use JMS\SecurityExtraBundle\Annotation\SecureParam;

class MiServicio
```

```
{
    /**
     * @SecureParam(name="comentario", permissions="EDIT, DELETE")
     * @SecureParam(name="post", permissions="OWNER")
     */
    public function metodoProtegido($comentario, $comunicado)
    {
        // ...
    }
}
```

@SecureReturn

Esta anotación te permite definir restricciones para el valor que devuelve el método. Esto también es útil sólo si el valor devuelto es un objeto del dominio:

```
    <?php

use JMS\SecurityExtraBundle\Annotation\SecureReturn;

class MiServicio
{
    /**
     * @SecureReturn(permissions="VIEW")
     */
    public function metodoProtegido()
    {
        // ...

        return $objetoDelDominio;
    }
}
```

@RunAs

Esta anotación te permite especificar los roles que se añadirán sólo mientras subsista la invocación del método. Estos roles no serán tomadas en consideración antes o después de las decisiones de acceso a la invocación.

Esto se suele utilizar para implementar una capa de dos niveles para el servicio, donde tienes servicios públicos y privados, y los servicios privados únicamente son invocados a través de un servicio público específico:

```
    <?php

use JMS\SecurityExtraBundle\Annotation\Secure;
use JMS\SecurityExtraBundle\Annotation\RunAs;

class MiServicioPrivado
{
    /**
     * @Secure(roles="ROLE_PRIVATE_SERVICE")
     */
    public function unMetodoParaSerInvocadoUnicamentePorUnCanalEspecifico()
    {
        // ...
    }
}
```

```
class MiServicioPublico
{
    protected $miServicioPrivado;

    /**
     * @Secure(roles="ROLE_USER")
     * @RunAs(roles="ROLE_PRIVATE_SERVICE")
     */
    public function sePuedeInvocarDesdeOtroServicio()
    {
        return $this->miServicioPrivado->unMetodoParaSerInvocadoUnicamentePorUnCanalEspecifico();
    }
}
```

@SatisfiesParentSecurityPolicy

Esto lo debes definir en un método que sustituya a un método que tiene metadatos de seguridad. Está ahí para asegurarse de que estás consciente de que la seguridad del método reemplazado no se puede hacerse valer más, y que tienes que copiar todas las anotaciones si deseas mantenerlas.

5.4 DoctrineFixturesBundle

Los accesorios se utilizan para cargar un conjunto controlado de datos en una base de datos. Puedes utilizar estos datos para pruebas o podrían ser los datos iniciales necesarios para ejecutar la aplicación sin problemas. *Symfony2* no tiene integrada forma alguna de administrar accesorios, pero *Doctrine2* cuenta con una biblioteca para ayudarte a escribir accesorios para el *ORM* (Página 115) u *ODM* (Página 601) de *Doctrine*.

5.4.1 Instalando y configurando

Si todavía no tienes configurada en *Symfony2* la biblioteca Doctrine Data Fixtures, sigue estos pasos para hacerlo.

Si estás utilizando la distribución estándar, agrega lo siguiente a tu archivo `deps`:

```
[doctrine-fixtures]
    git=http://github.com/doctrine/data-fixtures.git

[DoctrineFixturesBundle]
    git=http://github.com/symfony/DoctrineFixturesBundle.git
    target=/bundles/Symfony/Bundle/DoctrineFixturesBundle
```

Actualiza las bibliotecas de proveedores:

```
$ php bin/vendors install
```

Si todo funcionó, ahora puedes encontrar la biblioteca `doctrine-fixtures` en `vendor/doctrine-fixtures`.

Registra el espacio de nombres `Doctrine\Common\DataFixtures` en `app/autoload.php`.

```
// ...
$cargador->registerNamespaces(array(
    // ...
    'Doctrine\\Common\\DataFixtures' => __DIR__.'/../vendor/doctrine-fixtures/lib',
    'Doctrine\\Common' => __DIR__.'/../vendor/doctrine-common/lib',
```

```
    // ...
));
```

Prudencia: Asegúrate de registrar el nuevo espacio de nombres *antes* de `Doctrine\Common`. De lo contrario, *Symfony* buscará clases accesorio dentro del directorio `Doctrine\Common`. El autocargador de *Symfony*, siempre busca en primer lugar una clase dentro del directorio del espacio de nombres coincidente, los espacios de nombres más específicos *siempre* deben estar primero.

Por último, registra el paquete `DoctrineFixturesBundle` en `app/AppKernel.php`.

```
// ...
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\DoctrineFixturesBundle\DoctrineFixturesBundle(),
        // ...
    );
    // ...
}
```

5.4.2 Escribiendo accesorios simples

Los accesorios de *Doctrine2* son clases *PHP* que pueden crear y persistir objetos a la base de datos. Al igual que todas las clases en *Symfony2*, los accesorios deben vivir dentro de uno de los paquetes de tu aplicación.

Para un paquete situado en `src/Acme/HolaBundle`, las clases accesorio deben vivir dentro de `src/Acme/HolaBundle/DataFixtures/ORM` o `src/Acme/HolaBundle/DataFixtures/MongoDB`, para *ORM* y *ODM* respectivamente, esta guía asume que estás utilizando el *ORM* - pero, los accesorios se pueden agregar con la misma facilidad si estás utilizando *ODM*.

Imagina que tienes una clase `Usuario`, y te gustaría cargar un nuevo `Usuario`:

```
// src/Acme/HolaBundle/DataFixtures/ORM/CargaDatosUsuario.php
namespace Acme\HolaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Acme\HolaBundle\Entity\Usuario;

class CargaDatosUsuario implements FixtureInterface
{
    public function load($gestor)
    {
        $usuarioAdmin = new Usuario();
        $usuarioAdmin->setNombreUsuario('admin');
        $usuarioAdmin->setContrasena('prueba');

        $gestor->persist($usuarioAdmin);
        $gestor->flush();
    }
}
```

En *Doctrine2*, los accesorios son sólo objetos en los que cargas datos interactuando con tus entidades como lo haces normalmente. Esto te permite crear el accesorio exacto que necesitas para tu aplicación.

La limitación más importante es que no puedes compartir objetos entre accesorios. Más adelante, veremos la manera de superar esta limitación.

5.4.3 Ejecutando accesorios

Una vez que has escrito tus accesorios, los puedes cargar a través de la línea de ordenes usando la orden `doctrine:fixtures:load`

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

Si estás utilizando *ODM*, en su lugar usa la orden `doctrine:mongodb:fixtures:load`:

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

La tarea verá dentro del directorio `DataFixtures/ORM` (o `DataFixtures/MongoDB` para *ODM*) de cada paquete y ejecutará cada clase que implemente la `FixtureInterface`.

Ambas ordenes vienen con unas cuantas opciones:

- `--fixtures=/ruta/al/accesorio` - Usa esta opción para especificar manualmente el directorio de donde se deben cargar las clases accesorio;
- `--append` - Utiliza esta opción para añadir datos en lugar de eliminarlos antes de cargarlos (borrar primero es el comportamiento predeterminado);
- `--em=manager_name` - Especifica manualmente el gestor de la entidad a utilizar para cargar los datos.

Nota: Si utilizas la tarea `doctrine:mongodb:fixtures:load`, reemplaza la opción `--em=` con `--dm=` para especificar manualmente el gestor de documentos.

Un ejemplo de uso completo podría tener este aspecto:

```
php app/console doctrine:fixtures:load --fixtures=/ruta/al/accesorio1 --fixtures=/ruta/al/accesorio2
```

5.4.4 Compartiendo objetos entre accesorios

Escribir un accesorio básico es simple. Pero, ¿si tienes varias clases de accesorios y quieres poder referirte a los datos cargados en otras clases accesorio? Por ejemplo, ¿qué pasa si cargas un objeto `Usuario` en un accesorio, y luego quieres mencionar una referencia en un accesorio diferente con el fin de asignar dicho usuario a un grupo particular?

La biblioteca de accesorios de *Doctrine* maneja esto fácilmente permitiéndote especificar el orden en que se cargan los accesorios.

```
// src/Acme/HolaBundle/DataFixtures/ORM/CargaDatosUsuario.php
namespace Acme\HolaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Acme\HolaBundle\Entity\Usario;

class CargaDatosUsuario extends AbstractFixture implements OrderedFixtureInterface
{
    public function load($gestor)
    {
        $usuarioAdmin = new Usuario();
        $usuarioAdmin->setNombreUsuario('admin');
        $usuarioAdmin->setContrasena('prueba');

        $gestor->persist($usuarioAdmin);
        $gestor->flush();
```

```
        $this->addReference('usuario-admin', $usuarioAdmin);
    }

    public function getOrden()
    {
        return 1; // el orden en el que se deben cargar los accesorios
    }
}
```

La clase accesorio ahora implementa `OrderedFixtureInterface`, la cual dice a *Doctrine* que deseas controlar el orden de tus accesorios. Crea otra clase accesorio y haz que se cargue después de `CargaDatosUsuario` devolviendo un orden de 2:

```
// src/Acme/HolaBundle/DataFixtures/ORM/CargaDatosGrupo.php
namespace Acme\HolaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Acme\HolaBundle\Entity\Grupo;

class CargaDatosGrupo extends AbstractFixture implements OrderedFixtureInterface
{
    public function load($gestor)
    {
        $grupoAdmin = new Grupo();
        $grupoAdmin->setNombreGrupo('admin');

        $gestor->persist($grupoAdmin);
        $gestor->flush();

        $this->addReference('grupo-admin', $grupoAdmin);
    }

    public function getOrden()
    {
        return 2; // el orden en el que se deben cargar los accesorios
    }
}
```

Ambas clases accesorio extienden `AbstractFixture`, lo cual te permite crear objetos y luego ponerlos como referencias para que se puedan utilizar posteriormente en otros accesorios. Por ejemplo, posteriormente puedes referirte a los objetos `$usuarioAdmin` y `$grupoAdmin` a través de las referencias `usuario-admin` y `grupo-admin`:

```
// src/Acme/HolaBundle/DataFixtures/ORM/CargaDatosGrupoUsuario.php
namespace Acme\HolaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Acme\HolaBundle\Entity\GrupoUsuario;

class CargaDatosGrupoUsuario extends AbstractFixture implements OrderedFixtureInterface
{
    public function load($gestor)
    {
        $adminGrupoUsuario = new GrupoUsuario();
        $adminGrupoUsuario->setUser($gestor->merge($this->getReference('usuario-admin')));
        $adminGrupoUsuario->setGroup($gestor->merge($this->getReference('grupo-admin')));

        $gestor->persist($adminGrupoUsuario);
```

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

    public function getOrden()
    {
        return 3;
    }
}
```

Los accesorios ahora se ejecutan en el orden ascendente del valor devuelto por `getOrden()`. Cualquier objeto que se establece con el método `setReference()` se puede acceder a través de `getReference()` en las clases accesorio que tienen un orden superior.

Los accesorios te permiten crear cualquier tipo de dato que necesites a través de la interfaz normal de *PHP* para crear y persistir objetos. Al controlar el orden de los accesorios y establecer referencias, puedes manejar casi todo por medio de accesorios.

5.4.5 Usando el contenedor en los accesorios

En algunos casos necesitarás acceder a algunos servicios para cargar los accesorios. *Symfony2* hace esto realmente sencillo: el contenedor se inyectará en todas las clases accesorio que implementen `Symfony\Component\DependencyInjection\ContainerAwareInterface`.

Vamos a rescribir el primer accesorio para codificar la contraseña antes de almacenarla en la base de datos (una muy buena práctica). Esto utilizará el generador de codificadores para codificar la contraseña, asegurando que está codificada en la misma forma que la utiliza el componente de seguridad al efectuar la verificación:

```
// src/Acme/HolaBundle/DataFixtures/ORM/CargaDatosUsuario.php
namespace Acme\HolaBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Acme\HolaBundle\Entity\Usuario;

class CargaDatosUsuario implements FixtureInterface, ContainerAwareInterface
{
    private $contenedor;

    public function setContainer(ContainerInterface $contenedor = null)
    {
        $this->contenedor = $contenedor;
    }

    public function load($gestor)
    {
        $usuarioAdmin = new Usuario();
        $usuarioAdmin->setNombreUsuario('admin');
        $usuarioAdmin->setSalt(md5(time()));

        $codificador = $this->contenedor->get('security.encoder_factory')->getEncoder($usuarioAdmin);
        $usuarioAdmin->setPassword($codificador->encodePassword('prueba', $usuarioAdmin->getSalt()));

        $gestor->persist($usuarioAdmin);
        $gestor->flush();
    }
}
```

Como puedes ver, todo lo que necesitas hacer es agregar `ContainerAwareInterface` a la clase y luego crear un nuevo método `setContainer()` que implemente esa interfaz. Antes de que se ejecute el accesorio, *Symfony* automáticamente llamará al método `setContainer()`. Siempre y cuando guardes el contenedor como una propiedad en la clase (como se muestra arriba), puedes acceder a él en el método `load()`.

5.5 DoctrineMigrationsBundle

La funcionalidad de migración de base de datos, es una extensión de la capa de abstracción de bases de datos y te ofrece la posibilidad de desplegar programáticamente nuevas versiones del esquema de la base de datos de forma segura y estandarizada.

Truco: Puedes leer más sobre las migraciones de base de datos de *Doctrine* en la documentación del proyecto.

5.5.1 Instalando

Las migraciones de *Doctrine* para *Symfony* se mantienen en el DoctrineMigrationsBundle. Asegúrate de que tienes configuradas en tu proyecto ambas bibliotecas `doctrine-migrations` y `DoctrineMigrationsBundle`. Sigue estos pasos para instalar las bibliotecas en la distribución estándar de *Symfony*.

Agrega lo siguiente a `deps`. Esto registrará el paquete Migraciones y la biblioteca *doctrine-migrations* como dependencias en tu aplicación:

```
[doctrine-migrations]
    git=http://github.com/doctrine/migrations.git

[DoctrineMigrationsBundle]
    git=http://github.com/symfony/DoctrineMigrationsBundle.git
    target=/bundles/Symfony/Bundle/DoctrineMigrationsBundle
```

Actualiza las bibliotecas de proveedores:

```
$ php bin/vendors install
```

A continuación, asegúrate de que el nuevo espacio de nombres `Doctrine\DBAL\Migrations` se carga automáticamente a través del archivo `autoload.php`. El nuevo espacio de nombres `Migrations` *debe* estar colocado encima de la entrada `Doctrine\\DBAL` de manera que el cargador automático busque esas clases dentro del directorio migraciones:

```
// app/autoload.php
$loader->registerNamespaces(array(
    //...
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/../vendor/doctrine-migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/../vendor/doctrine-dbal/lib',
));
```

Por último, asegúrate de activar el paquete en `AppKernel.php` incluyendo lo siguiente:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        //...
        new Symfony\Bundle\DoctrineMigrationsBundle\DoctrineMigrationsBundle(),
```

```
    );
}
```

5.5.2 Uso

Toda la funcionalidad de las migraciones se encuentra en unas cuantas ordenes de consola:

```
doctrine:migrations
  :diff     Genera una migración comparando tu base de datos actual
            a tu información asignada.
  :execute  Ejecuta manualmente una sola versión de migración hacia
            arriba o abajo.
  :generate Genera una clase de migración en blanco.
  :migrate  Executa una migración a una versión especificada o la
            última versión disponible.
  :status   Ve el estado de un conjunto de migraciones.
  :version  Añade y elimina versiones de migración manualmente desde
            la tabla de versiones.
```

Empieza consiguiendo la situación de las migraciones en tu aplicación ejecutando la orden `status`:

```
php app/console doctrine:migrations:status

 == Configuración

    >> Nombre:                                Migraciones de HolaBundle
    >> Fuente de configuración:               configurada manualmente
    >> Nombre de la tabla de versión:         hola_bundle_migration_versions
    >> Espacio de nombres de las migraciones: Application\Migrations
    >> Directorio de las migraciones:         ruta/a/tu/proyecto/app/DoctrineMigrations
    >> Versión actual:                        0
    >> Última versión:                        0
    >> Migraciones realizadas:                0
    >> Migraciones disponibles:               0
    >> Nuevas Migraciones:                    0
```

Ahora, podemos empezar a trabajar con las migraciones generando una nueva clase de migración en blanco. Más adelante, aprenderás cómo puedes generar migraciones automáticamente con *Doctrine*.

```
php app/console doctrine:migrations:generate
Nueva clase migración generada para "/ruta/a/tu/proyecto/app/DoctrineMigrations/Version20100621140655
```

Echa un vistazo a la clase migración recién generada y verás algo como lo siguiente:

```
namespace Application\Migrations;

use Doctrine\DBAL\Migrations\AbstractMigration,
    Doctrine\DBAL\Schema\Schema;

class Version20100621140655 extends AbstractMigration
{
    public function up(Schema $schema)
    {

    }

    public function down(Schema $schema)
    {
```

```
    }
}
```

Si ahora ejecutas la orden `status` te mostrará que tienes una nueva migración por ejecutar:

```
php app/console doctrine:migrations:status

 == Configuración

   >> Nombre:                                Aplicación de migraciones
   >> Fuente de configuración:               configurada manualmente
   >> Nombre de la tabla de versión:         migration_versions
   >> Espacio de nombres de las migraciones: Application\Migrations
   >> Directorio de las migraciones:         /ruta/a/tu/proyecto/app/DoctrineMigrations
   >> Versión actual:                        0
   >> Última versión:                        2011-06-21 14:06:55 (20100621140655)
   >> Migraciones realizadas:                0
   >> Migraciones disponibles:               1
   >> Nuevas Migraciones:                    1

== Versiones de migración

   >> 2011-06-21 14:06:55 (20110621140655)               no migrada
```

Ahora puedes agregar algo de código de migración a los métodos `up()` y `down()`, y finalmente cuando estés listo migrar:

```
php app/console doctrine:migrations:migrate
```

Para más información sobre cómo escribir migraciones en sí mismas (es decir, la manera de rellenar los métodos `up()` y `down()`), consulta la documentación oficial de las Migraciones de *Doctrine*.

Ejecutando migraciones al desplegar tu aplicación

Por supuesto, el objetivo final al escribir migraciones es poder utilizarlas para actualizar de manera fiable la estructura de tu base de datos cuando despliegues tu aplicación. Al ejecutar las migraciones localmente (o en un servidor de pruebas), puedes asegurarte de que las migraciones trabajan según lo previsto.

Cuando finalmente despliegues tu aplicación, sólo tienes que recordar ejecutar la orden `doctrine:migrations:migrate`. Internamente, *Doctrine* crea una tabla `migration_versions` dentro de la base de datos y allí lleva a cabo el seguimiento de las migraciones que se han ejecutado. Por lo tanto, no importa cuantas migraciones hayas creado y ejecutado localmente, cuando se ejecuta la orden durante el despliegue, *Doctrine* sabrá exactamente qué migraciones no se han ejecutado todavía mirando la tabla `migration_versions` de tu base de datos en producción. Independientemente de qué servidor esté activado, siempre puedes ejecutar esta orden de forma segura para realizar sólo las migraciones que todavía no se han llevado a cabo en *esa* base de datos particular.

5.5.3 Generando migraciones automáticamente

En realidad, no deberías tener que escribir migraciones manualmente, puesto que la biblioteca de migraciones puede generar las clases de la migración automáticamente comparando tu información asignada a *Doctrine* (es decir, cómo se *debe* ver tu base de datos) con la estructura de la base de datos actual.

Por ejemplo, supongamos que creas una nueva entidad `Usuario` y agregas información asignándola al *ORM* de *Doctrine*:

- *Annotations*

```
// src/Acme/HolaBundle/Entity/Usuario.php
namespace Acme\HolaBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="hola_usuario")
 */
class Usuario
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

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

- *YAML*

```
# src/Acme/HolaBundle/Resources/config/doctrine/Usuario.orm.yml
Acme\HolaBundle\Entity\Usuario:
    type: entity
    table: hola_usuario
    id:
        id:
            type: integer
            generator:
                strategy: AUTO
    fields:
        nombre:
            type: string
            length: 255
```

- *XML*

```
<!-- src/Acme/HolaBundle/Resources/config/doctrine/Usuario.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">

    <entity name="Acme\HolaBundle\Entity\Usuario" table="hola_usuario">
        <id name="id" type="integer" column="id">
            <generator strategy="AUTO"/>
        </id>
        <field name="nombre" column="nombre" type="string" length="255" />
    </entity>

</doctrine-mapping>
```

Con esta información, *Doctrine* ya está listo para ayudarte a persistir tu nuevo objeto `Usuario` hacia y desde la tabla

hola_usuario. Por supuesto, ¡esta tabla no existe aún! Genera una nueva migración para esta tabla automáticamente ejecutando la siguiente orden:

```
php app/console doctrine:migrations:diff
```

Deberás ver un mensaje informado que se ha generado una nueva clase migración basada en las diferencias del esquema. Si abres ese archivo, encontrarás que tiene el código SQL necesario para crear la tabla hola_usuario. A continuación, ejecuta la migración para agregar la tabla a tu base de datos:

```
php app/console doctrine:migrations:migrate
```

La moraleja de la historia es la siguiente: después de cada cambio que realices en tu información de asignación a *Doctrine*, ejecuta la orden doctrine:migrations:diff para generar automáticamente las clases de la migración.

Si lo haces desde el principio de tu proyecto (es decir, de modo que incluso las primeras tablas fueran cargadas a través de una clase migración), siempre podrás crear una base de datos actualizada y ejecutar las migraciones a fin de tener tu esquema de base de datos totalmente actualizado. De hecho, este es un flujo de trabajo fácil y confiable para tu proyecto.

5.6 DoctrineMongoDBBundle

5.6.1 DoctrineMongoDBBundle Configuration

Configuración de ejemplo

```
# app/config/config.yml
doctrine_mongodb:
    connections:
        default:
            server: mongodb://localhost:27017
            options:
                connect: true
    default_database: hola_%kernel.environment%
    document_managers:
        default:
            mappings:
                AcmeDemoBundle: ~
            metadata_cache_driver: array # array, apc, xcache, memcache
```

Si deseas utilizar memcache para memorizar los metadatos, es necesario configurar la instancia Memcache, por ejemplo, puedes hacer lo siguiente:

- *YAML*

```
# app/config/config.yml
doctrine_mongodb:
    default_database: hola_%kernel.environment%
    connections:
        default:
            server: mongodb://localhost:27017
            options:
                connect: true
    document_managers:
        default:
            mappings:
                AcmeDemoBundle: ~
```

```
            metadata_cache_driver:
                type: memcache
                class: Doctrine\Common\Cache\MemcacheCache
                host: localhost
                port: 11211
                instance_class: Memcache
```

- *XML*

```
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine_mongodb="http://symfony.com/schema/dic/doctrine/odm/mongodb"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine/odm/mongodb http://symfony.com/sc

    <doctrine_mongodb:config default-database="hola_%kernel.environment%">
        <doctrine_mongodb:document-manager id="default">
            <doctrine_mongodb:mapping name="AcmeDemoBundle" />
            <doctrine_mongodb:metadata-cache-driver type="memcache">
                <doctrine_mongodb:class>Doctrine\Common\Cache\MemcacheCache</doctrine_mongodb:cl
                <doctrine_mongodb:host>localhost</doctrine_mongodb:host>
                <doctrine_mongodb:port>11211</doctrine_mongodb:port>
                <doctrine_mongodb:instance-class>Memcache</doctrine_mongodb:instance-class>
            </doctrine_mongodb:metadata-cache-driver>
        </doctrine_mongodb:document-manager>
        <doctrine_mongodb:connection id="default" server="mongodb://localhost:27017">
            <doctrine_mongodb:options>
                <doctrine_mongodb:connect>true</doctrine_mongodb:connect>
            </doctrine_mongodb:options>
        </doctrine_mongodb:connection>
    </doctrine_mongodb:config>
</container>
```

Configurando la asignación

La definición explícita de todos los documentos asignados es la única configuración necesaria para *ODM* y hay varias opciones de configuración que puedes controlar. Existen las siguientes opciones de configuración para una asignación:

- `type` Uno de `annotations`, `xml`, `yml`, `php` o `staticphp`. Esta especifica cual tipo de metadatos usa el tipo de tu asignación.
- `dir` Ruta a los archivos de entidad o asignación (dependiendo del controlador). Si esta ruta es relativa, se supone que es relativa a la raíz del paquete. Esto sólo funciona si el nombre de tu asignación es un nombre de paquete. Si deseas utilizar esta opción para especificar rutas absolutas debes prefijar la ruta con los parámetros del núcleo existentes en el *DIC* (por ejemplo %kernel.root_dir %).
- `prefix` Un prefijo de espacio de nombres común que comparten todos los documentos de esta asignación. Este prefijo no debe entrar en conflicto con otros prefijos definidos por otras asignaciones, de otra manera *Doctrine* no podrá encontrar algunos de tus documentos. Esta opción, por omisión, es el espacio de nombres del paquete + `Document`, por ejemplo, para un paquete de la aplicación llamada `AcmeHolaBundle`, el prefijo sería `Acme\HolaBundle\Document`.
- `alias` *Doctrine* ofrece una forma simple para rebautizar el espacio de nombres de los documentos, los nombres más cortos se utilizan en las consultas o para acceder al repositorio.

- `is_bundle` Esta opción es un valor derivado de `dir` y por omisión se establece en `true` si `dir` es relativo provisto por un `file_exists()` verifica que devuelve `false`. Este es `false` si al comprobar la existencia devuelve `true`. En este caso has especificado una ruta absoluta y es más probable que los archivos de metadatos estén en un directorio fuera del paquete.

Para evitar tener que configurar un montón de información para tus asignaciones, debes seguir los siguientes convenios:

1. Pon todos tus documentos en un directorio `Document/` dentro de tu paquete. Por ejemplo `Acme/HolaBundle/Document/`.
2. Si estás usando asignación `xml`, `php` o `yml` coloca todos tus archivos de configuración en el directorio `Resources/config/doctrine/` con el sufijo `mongodb.xml`, `mongodb.yml` o `mongodb.php` respectivamente.
3. Asume anotaciones si es un `Document/` pero no se encuentra el directorio `Resources/config/doctrine/`.

La siguiente configuración muestra un montón de ejemplos de asignación:

```
doctrine_mongodb:
    document_managers:
        default:
            mappings:
                MiPaquete1: ~
                MiPaquete2: yml
                MiPaquete3: { type: annotation, dir: Documents/ }
                MiPaquete4: { type: xml, dir: Resources/config/doctrine/mapping }
                MiPaquete5:
                    type: yml
                    dir: mi-asignacion-de-paquete-dir
                    alias: BundleAlias
                doctrine_extensions:
                    type: xml
                    dir: %kernel.root_dir%/../src/vendor/DoctrineExtensions/lib/DoctrineExtensions/Do
                    prefix: DoctrineExtensions\Documents\
                    alias: DExt
```

Múltiples conexiones

Si necesitas múltiples conexiones y gestores de documentos puedes utilizar la siguiente sintaxis:

Ahora puedes recuperar los servicios configurados conectando servicios:

```
$conn1 = $contenedor->get('doctrine.odm.mongodb.conn1_connection');
$conn2 = $contenedor->get('doctrine.odm.mongodb.conn2_connection');
```

Y también puedes recuperar los gestores de servicios de documentos configurados que utilizan la conexión de servicios anterior:

```
$dm1 = $contenedor->get('doctrine.odm.mongodb.dm1_document_manager');
$dm2 = $contenedor->get('doctrine.odm.mongodb.dm2_document_manager');
```

Configuración predeterminada completa

- *YAML*

```
doctrine_mongodb:
    document_managers:

        # Prototipo
        id:
            connection:           ~
            database:             ~
            logging:              true
            auto_mapping:         false
            metadata_cache_driver:
                type:                 ~
                class:                ~
                host:                 ~
                port:                 ~
                instance_class:       ~
            mappings:

                # Prototipo
                name:
                    mapping:              true
                    type:                 ~
                    dir:                  ~
                    prefix:               ~
                    alias:                ~
                    is_bundle:            ~
    connections:

        # Prototipo
        id:
            server:               ~
            options:
                connect:              ~
                persist:              ~
                timeout:              ~
                replicaSet:           ~
                username:             ~
                password:             ~
    proxy_namespace:      Proxies
    proxy_dir:            %kernel.cache_dir%/doctrine/odm/mongodb/Proxies
    auto_generate_proxy_classes:  false
    hydrator_namespace:   Hydrators
    hydrator_dir:         %kernel.cache_dir%/doctrine/odm/mongodb/Hydrators
    auto_generate_hydrator_classes:  false
    default_document_manager:  ~
    default_connection:   ~
    default_database:     default
```

5.6.2 Cómo implementar un sencillo formulario de inscripción con *MongoDB*

Algunos formularios tienen campos adicionales cuyos valores no es necesario almacenar en la base de datos. En este ejemplo, vamos a crear un formulario de registro con algunos campos adicionales, y un campo "términos aceptados" (como casilla de verificación) incluido en el formulario que en realidad almacena la información de la cuenta. Vamos a utilizar *MongoDB* para almacenar los datos.

El modelo de Usuario simple

Por lo tanto, en esta guía comenzaremos con el modelo para un documento `Usuario`:

```
// src/Acme/CuentaBundle/Document/Usuario.php
namespace Acme\CuentaBundle\Document;

use Doctrine\ODM\MongoDB\Mapping\Annotations as MongoDB;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bundle\DoctrineMongoDBBundle\Validator\Constraints\Unique as MongoDBUnique;

/**
 * @MongoDB\Document(collection="usuarios")
 * @MongoDBUnique(path="correo")
 */
class Usuario
{
    /**
     * @MongoDB\Id
     */
    protected $id;

    /**
     * @MongoDB\Field(type="string")
     * @Assert\NotBlank()
     * @Assert\Correo()
     */
    protected $correo;

    /**
     * @MongoDB\Field(type="string")
     * @Assert\NotBlank()
     */
    protected $pase;

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

    public function getCorreo()
    {
        return $this->correo;
    }

    public function setEmail($correo)
    {
        $this->correo = $correo;
    }

    public function getContrasena()
    {
        return $this->pase;
    }

    // cifrado estúpidamente simple (¡por favor no copies esto!)
    public function setContrasena($pase)
    {
        $this->pase = sha1($pase);
```

```
    }
}
```

Este documento `Usuario` contiene tres campos y dos de ellos (correo y contraseña) se deben mostrar en el formulario. La propiedad correo debe ser única en la base de datos, por lo tanto añadimos esta validación en lo alto de la clase.

Nota: Si deseas integrar este Usuario en el sistema de seguridad, es necesario implementar la *Interfaz de usuario* (Página 198) del componente de Seguridad.

Creando un formulario para el modelo

A continuación, crea el formulario para el modelo `Usuario`:

```
// src/Acme/CuentaBundle/Form/Type/UsuarioType.php
namespace Acme\CuentaBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\FormBuilder;

class UsuarioType extends AbstractType
{
    public function buildForm(FormBuilder $generador, array $Opciones)
    {
        $generador->add('correo', 'correo');
        $generador->add('password', 'repeated', array(
           'first_name' => 'password',
           'second_name' => 'confirm',
           'type' => 'password'
        ));
    }

    public function getDefaultOptions(array $opciones)
    {
        return array('data_class' => 'Acme\CuentaBundle\Document\Usuario');
    }

    public function getNombre()
    {
        return 'usuario';
    }
}
```

Acabamos de añadir dos campos: correo y contraseña (repetido para confirmar la contraseña introducida). La opción `data_class` le indica al formulario el nombre de la clase de los datos (es decir, el documento `Usuario`).

Truco: Para explorar más cosas sobre el componente Formulario, lee esta documentación *archivo* (Página 160).

Incorporando el formulario Usuario en un formulario de inscripción

El formulario que vamos a usar para la página de registro no es el mismo que el formulario utilizado para simplemente modificar al `Usuario` (es decir, `UsuarioType`). El formulario de registro contiene más campos como "acepto las condiciones", cuyo valor no se almacenará en la base de datos.

En otras palabras, creas un segundo formulario de inscripción, el cual incorpora el formulario `Usuario` y añades el campo extra necesario. Empecemos creando una clase simple que representa la "inscripción":

```
// src/Acme/CuentaBundle/Form/Model/Alta.php
namespace Acme\CuentaBundle\Form\Model;

use Symfony\Component\Validator\Constraints as Assert;

use Acme\CuentaBundle\Document\Usuario;

class Alta
{
    /**
     * @Assert\Type(type="Acme\CuentaBundle\Document\Usuario")
     */
    protected $usuario;

    /**
     * @Assert\NotBlank()
     * @Assert\True()
     */
    protected $terminosAceptados;

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

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

    public function getTerminosAceptados()
    {
        return $this->terminosAceptados;
    }

    public function setTerminosAceptados($terminosAceptados)
    {
        $this->terminosAceptados = (boolean)$terminosAceptados;
    }
}
```

A continuación, crea el formulario para el modelo `Registro`:

```
// src/Acme/CuentaBundle/Form/Type/RegistrationType.php
namespace Acme\CuentaBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\FormBuilder;

class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilder $generador, array $opciones)
    {
        $generador->add('usuario', new UsuarioType());
        $generador->add('terminos', 'checkbox', array('property_path' => 'terminosAceptados'));
```

```
    }

    public function getNombre()
    {
        return 'registro';
    }
}
```

No necesitas utilizar métodos especiales para integrar el `UsuarioType` en el formulario. Un formulario es un campo, también - por lo tanto lo puedes añadir como cualquier otro campo, con la expectativa de que la propiedad `Usuario` correspondiente mantendrá una instancia de la clase `UsuarioType`.

Manejando el envío del formulario

A continuación, necesitas un controlador para manejar el formulario. Comienza creando un controlador simple para mostrar el formulario de inscripción:

```
// src/Acme/CuentaBundle/Controller/CuentaController.php
namespace Acme\CuentaBundle\Controller;

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

use Acme\CuentaBundle\Form\Type\RegistroType;
use Acme\CuentaBundle\Form\Model\Registro;

class CuentaController extends Controller
{
    public function registroAction()
    {
        $formulario = $this->createForm(new RegistroType(), new Registro());

        return $this->render('AcmeCuentaBundle:Cuenta:registro.html.twig', array('form' => $formulari
    }
}
```

y su plantilla:

```
{# src/Acme/CuentaBundle/Resources/views/Cuenta/registro.html.twig #}

<form action="{{ path('crea') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

Finalmente, crea el controlador que maneja el envío del formulario. Esto realiza la validación y guarda los datos en *MongoDB*:

```
public function creaAction()
{
    $dm = $this->get('doctrine.odm.mongodb.default_document_manager');

    $formulario = $this->createForm(new RegistroType(), new Registro());

    $formulario->bindRequest($this->getRequest());

    if ($formulario->isValid()) {
```

```
        $registro = $formulario->getData();

        $dm->persist($registro->getUsuario());
        $dm->flush();

        return $this->redirect(...);
    }

    return $this->render('AcmeCuentaBundle:Cuenta:registro.html.twig', array('form' => $formulario->c
}
```

¡Eso es todo! Tu formulario ahora valida, y te permite guardar el objeto `Usuario` a *MongoDB*.

El asignador de objeto a documento MongoDB (*ODM* por Object Document Mapper) es muy similar al *ORM* de *Doctrine2* en su filosofía y funcionamiento. En otras palabras, similar al *ORM de Doctrine2* (Página 115), con el *ODM* de *Doctrine*, sólo tratas con objetos *PHP* simples, los cuales luego se persisten de forma transparente hacia y desde *MongoDB*.

Truco: Puedes leer más acerca del *ODM* de *Doctrine MongoDB* en la documentación del proyecto.

Está disponible el paquete que integra el *ODM MongoDB de Doctrine* en *Symfony*, por lo tanto es fácil configurarlo y usarlo.

Nota: Este capítulo lo debes de sentir muy parecido al capítulo *ORM de Doctrine2* (Página 115), que habla de cómo puedes utilizar el *ORM* de *Doctrine* para guardar los datos en bases de datos relacionales (por ejemplo, *MySQL*). Este es a propósito - si persistes en una base de datos relacional por medio del *ORM* o a través del *ODM MongoDB*, las filosofías son muy parecidas.

5.6.3 Instalando

Para utilizar el *ODM MongoDB*, necesitarás dos bibliotecas proporcionadas por *Doctrine* y un paquete que las integra en *Symfony*. Si estás usando la distribución estándar de *Symfony*, agrega lo siguiente al archivo `deps` en la raíz de tu proyecto:

```
[doctrine-mongodb]
    git=http://github.com/doctrine/mongodb.git

[doctrine-mongodb-odm]
    git=http://github.com/doctrine/mongodb-odm.git

[DoctrineMongoDBBundle]
    git=http://github.com/symfony/DoctrineMongoDBBundle.git
    target=/bundles/Symfony/Bundle/DoctrineMongoDBBundle
```

Ahora, actualiza las bibliotecas de proveedores ejecutando:

```
$ php bin/vendors install
```

A continuación, agrega los espacios de nombres `Doctrine\ODM\MongoDB` y `Doctrine\MongoDB` al archivo `app/autoload.php` para que estas bibliotecas se puedan cargar automáticamente. Asegúrate de añadirlas en cualquier lugar *por encima* del espacio de nombres *Doctrine* (cómo se muestra aquí):

```
// app/autoload.php
$loader->registerNamespaces(array(
    // ...
```

```
    'Doctrine\\ODM\\MongoDB'     => __DIR__.'/../vendor/doctrine-mongodb-odm/lib',
    'Doctrine\\MongoDB'          => __DIR__.'/../vendor/doctrine-mongodb/lib',
    'Doctrine'                   => __DIR__.'/../vendor/doctrine/lib',
    // ...
));
```

A continuación, registra la biblioteca de anotaciones añadiendo las siguientes acciones al cargador (debajo de la línea `AnnotationRegistry::registerFile` existente):

```
// app/autoload.php
AnnotationRegistry::registerFile(
    __DIR__.'/../vendor/doctrine-mongodb-odm/lib/Doctrine/ODM/MongoDB/Mapping/Annotations/DoctrineAn
);
```

Por último, activa el nuevo paquete en el núcleo:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\DoctrineMongoDBBundle\DoctrineMongoDBBundle(),
    );

    // ...
}
```

¡Enhorabuena! Estás listo para empezar a trabajar.

5.6.4 Configurando

Para empezar, necesitarás una estructura básica que configure el gestor de documentos. La forma más fácil es habilitar el `auto_mapping`, el cual activará al *ODM MongoDB* a través de tu aplicación:

```
# app/config/config.yml
doctrine_mongodb:
    connections:
        default:
            server: mongodb://localhost:27017
            options:
                connect: true
    default_database: test_database
    document_managers:
        default:
            auto_mapping: true
```

Nota: Por supuesto, también tienes que asegurarte de que se ejecute en segundo plano el servidor *MongoDB*. Para más información, consulta la Guía de inicio rápido de *MongoDB*.

5.6.5 Un ejemplo sencillo: un producto

La mejor manera de entender el *ODM* de *Doctrine MongoDB* es verlo en acción. En esta sección, recorreremos cada paso necesario para empezar a persistir documentos hacia y desde *MongoDB*.

Código del ejemplo

Si quieres seguir el ejemplo de este capítulo, crea el paquete `AcmeTiendaBundle` ejecutando la orden:

```
php app/console generate:bundle --namespace=Acme/TiendaBundle
```

Creando una clase Documento

Supongamos que estás construyendo una aplicación donde necesitas mostrar tus productos. Sin siquiera pensar en *Doctrine* o *MongoDB*, ya sabes que necesitas un objeto `Producto` para representar los productos. Crea esta clase en el directorio `Document` de tu `AcmeTiendaBundle`:

```
// src/Acme/TiendaBundle/Document/Producto.php
namespace Acme\TiendaBundle\Document;

class Producto
{
    protected $nombre;

    protected $precio;
}
```

La clase - a menudo llamada "documento", es decir, *una clase básica que contiene los datos* - es simple y ayuda a cumplir con el requisito del negocio de que tu aplicación necesita productos. Esta clase, todavía no se puede persistir a *Doctrine MongoDB* - es sólo una clase *PHP* simple.

Agregando información de asignación

Doctrine te permite trabajar con *MongoDB* de una manera mucho más interesante que solo recuperar datos de un lado a otro como una matriz. En cambio, *Doctrine* te permite persistir *objetos* completos a *MongoDB* y recuperar objetos enteros desde *MongoDB*. Esto funciona asignando una clase *PHP* y sus propiedades a las entradas de una colección *MongoDB*.

Para que *Doctrine* sea capaz de hacer esto, sólo tienes que crear "metadatos", o la configuración que le dice a *Doctrine* exactamente cómo se deben *asignar* a *MongoDB* la clase `Producto` y sus propiedades. Estos metadatos se pueden especificar en una variedad de formatos diferentes, incluyendo *YAML*, *XML* o directamente dentro de la clase `Producto` a través de anotaciones:

- *Annotations*

```
// src/Acme/TiendaBundle/Document/Producto.php
namespace Acme\TiendaBundle\Document;

use Doctrine\ODM\MongoDB\Mapping\Annotations as MongoDB;

/**
 * @MongoDB\Document
 */
class Producto
{
    /**
     * @MongoDB\Id
     */
    protected $id;
```

```
    /**
     * @MongoDB\String
     */
    protected $nombre;

    /**
     * @MongoDB\Float
     */
    protected $precio;
}
```

- *YAML*

```
# src/Acme/TiendaBundle/Resources/config/doctrine/Producto.mongodb.yml
Acme\TiendaBundle\Document\Producto:
    fields:
        id:
            id:  true
        nombre:
            type: string
        precio:
            type: float
```

- *XML*

```
<!-- src/Acme/TiendaBundle/Resources/config/doctrine/Producto.mongodb.xml -->
<doctrine-mongo-mapping xmlns="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping
                  http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping.xsd">

    <document name="Acme\TiendaBundle\Document\Producto">
        <field fieldName="id" id="true" />
        <field fieldName="nombre" type="string" />
        <field fieldName="precio" type="float" />
    </document>
</doctrine-mongo-mapping>
```

Doctrine te permite elegir entre una amplia variedad de tipos de campo diferentes, cada uno con sus propias opciones. Para más información sobre los tipos de campo disponibles, consulta la sección *Referencia de tipos de campo Doctrine* (Página 618).

Ver También:

También puedes consultar la Documentación de asignación básica de *Doctrine* para todos los detalles sobre la información de asignación. Si utilizas anotaciones, tendrás que prefijar todas tus anotaciones con `MongoDB\` (por ejemplo, `MongoDB\Cadena`), lo cual no se muestra en la documentación de *Doctrine*. También tendrás que incluir la declaración `use Doctrine\ODM\MongoDB\Mapping\Annotations as MongoDB;`, que *importa* el prefijo `MongoDB` para las anotaciones.

Generando captadores y definidores

A pesar de que *Doctrine* ya sabe cómo persistir un objeto `Producto` a *MongoDB*, la clase en sí todavía no es realmente útil. Puesto que `Producto` es sólo una clase *PHP* regular, es necesario crear métodos captadores y definidores (por ejemplo, `getNombre()`, `setNombre()`) para poder acceder a sus propiedades (ya que las propiedades son `protegidas`). Afortunadamente, *Doctrine* puede hacer esto por ti con la siguiente orden:

```
php app/console doctrine:mongodb:generate:documents AcmeTiendaBundle
```

Esta orden se asegura de que se generen todos los captadores y definidores para la clase `Producto`. Esta es una orden segura - la puedes ejecutar una y otra vez: sólo genera captadores y definidores que no existen (es decir, no sustituye métodos existentes).

Nota: A *Doctrine* no le importa si tus propiedades son `protegidas` o `privadas`, o si una propiedad tiene o no una función captadora o definidora. Aquí, los captadores y definidores se generan sólo porque los necesitarás para interactuar con tu objeto *PHP*.

Persistiendo objetos a *MongoDB*

Ahora que tienes asignado un documento `Producto` completo, con métodos captadores y definidores, estás listo para persistir los datos a *MongoDB*. Desde el interior de un controlador, esto es bastante fácil. Agrega el siguiente método al `DefaultController` del paquete:

```
1  // src/Acme/TiendaBundle/Controller/DefaultController.php
2  use Acme\TiendaBundle\Document\Producto;
3  use Symfony\Component\HttpFoundation\Response;
4  // ...
5
6  public function creaAction()
7  {
8      $producto = new Producto();
9      $producto->setNombre('Un Foo Bar');
10     $producto->setPrecio('19.99');
11
12     $dm = $this->get('doctrine.odm.mongodb.document_manager');
13     $dm->persist($producto);
14     $dm->flush();
15
16     return new Response('Producto creado '.$producto->getId());
17 }
```

Nota: Si estás siguiendo este ejemplo, tendrás que crear una ruta que apunte a esta acción para verla trabajar.

Vamos a recorrer este ejemplo:

- **Líneas 8-10** En esta sección, creas una instancia y trabajas con el objeto `$producto` como cualquier otro objeto *PHP* normal;
- **Línea 12** Esta línea recupera el objeto *gestor de documentos*, el cual es responsable de manejar el proceso de persistir y recuperar objetos hacia y desde *MongoDB*;
- **Línea 13** El método `persist()` dice a *Doctrine* que "maneje" el objeto `$producto`. Esto en realidad no resulta en una consulta que se deba hacer a *MongoDB* (todavía).
- **Línea 14** Cuando se llama al método `flush()`, *Doctrine* mira todos los objetos que está gestionando para ver si se necesita persistirlos a *MongoDB*. En este ejemplo, el objeto `$producto` aún no se ha persistido, por lo que el gestor de documentos hace una consulta a *MongoDB*, la cual añade una nueva entrada.

Nota: De hecho, ya que *Doctrine* está consciente de todos los objetos gestionados, cuando llamas al método `flush()`, se calcula un conjunto de cambios y ejecuta la operación más eficiente posible.

Al crear o actualizar objetos, el flujo de trabajo siempre es el mismo. En la siguiente sección, verás cómo *Doctrine* es lo suficientemente inteligente como para actualizar las entradas, si ya existen en *MongoDB*.

Truco: *Doctrine* proporciona una biblioteca que te permite cargar en tu proyecto mediante programación los datos de prueba (es decir, "datos accesorios"). Para más información, consulta *DoctrineFixturesBundle* (Página 592).

Recuperando objetos desde *MongoDB*

Recuperar un objeto de *MongoDB* incluso es más fácil. Por ejemplo, supongamos que has configurado una ruta para mostrar un `Producto` específico en función del valor de su `id`:

```
public function showAction($id)
{
    $producto = $this->get('doctrine.odm.mongodb.document_manager')
        ->getRepository('AcmeTiendaBundle:Producto')
        ->find($id);

    if (!$producto) {
        throw $this->createNotFoundException('Ningún producto encontrado con id '.$id);
    }

    // haz algo, como pasar el objeto $producto a una plantilla
}
```

Al consultar por un determinado tipo de objeto, siempre utilizas lo que se conoce como "repositorio". Puedes pensar en un repositorio como una clase *PHP*, cuyo único trabajo consiste en ayudarte a buscar los objetos de una determinada clase. Puedes acceder al objeto repositorio de una clase documento vía:

```
$repositorio = $this->get('doctrine.odm.mongodb.document_manager')
    ->getRepository('AcmeTiendaBundle:Producto');
```

Nota: La cadena `AcmeTiendaBundle:Producto` es un método abreviado que puedes utilizar en cualquier lugar de *Doctrine* en lugar del nombre completo de la clase del documento (es decir, `Acme\TiendaBundle\Document\Producto`). Mientras tu documento viva en el espacio de nombres `Document` de tu paquete, esto va a funcionar.

Una vez que tengas tu repositorio, tienes acceso a todo tipo de útiles métodos:

```
// consulta por la clave principal (generalmente "id")
$producto = $repositorio->find($id);

// nombres de método dinámicos para buscar basándose en un valor de columna
$producto = $repositorio->findOneById($id);
$producto = $repositorio->findOneByName('foo');

// recupera *todos* los productos
$productos = $repositorio->findAll();

// busca un grupo de productos basándose en el valor de una columna arbitraria
$productos = $repositorio->findByPrice(19.99);
```

Nota: Por supuesto, también puedes realizar consultas complejas, acerca de las cuales aprenderás más en la sección *Consultando objetos* (Página 123).

También puedes tomar ventaja de los útiles métodos `findBy` y `findOneBy` para recuperar objetos fácilmente basándote en varias condiciones:

```
// consulta por un producto que coincide en nombre y precio
$producto = $repositorio->findOneBy(array('nombre' => 'foo', 'precio' => 19.99));

// consulta por todos los productos que coinciden con el nombre, y los ordena por precio
$producto = $repositorio->findBy(
    array('nombre' => 'foo'),
    array('precio', 'ASC')
);
```

Actualizando un objeto

Una vez que hayas extraído un objeto de *Doctrine*, actualizarlo es relativamente fácil. Supongamos que tienes una ruta que asigna un identificador de producto a una acción de actualización de un controlador:

```
public function updateAction($id)
{
    $dm = $this->get('doctrine.odm.mongodb.document_manager');
    $producto = $dm->getRepository('AcmeTiendaBundle:Producto')->find($id);

    if (!$producto) {
        throw $this->createNotFoundException('Ningún producto encontrado con id '.$id);
    }

    $producto->setNombre('¡Nuevo nombre de producto!');
    $dm->flush();

    return $this->redirect($this->generateUrl('portada'));
}
```

La actualización de un objeto únicamente consiste de tres pasos:

1. Recuperar el objeto desde *Doctrine*;
2. Modificar el objeto;
3. Llamar a `flush()` en el gestor del documento;

Ten en cuenta que `$dm->persist($producto)` no es necesario. Recuerda que este método simplemente dice a *Doctrine* que maneje o "vea" el objeto `$producto`. En este caso, ya que recuperaste el objeto `$producto` desde *Doctrine*, este ya está gestionado.

Eliminando un objeto

La eliminación de un objeto es muy similar, pero requiere una llamada al método `remove()` del gestor de documentos:

```
$dm->remove($producto);
$dm->flush();
```

Como es de esperar, el método `remove()` notifica a *Doctrine* que deseas eliminar el documento propuesto de *MongoDB*. La operación real de eliminar sin embargo, no se ejecuta efectivamente hasta que invocas al método `flush()`.

5.6.6 Consultando por objetos

Como vimos anteriormente, la clase incorporada repositorio te permite consultar por uno o varios objetos basándote en una serie de diferentes parámetros. Cuando esto es suficiente, esta es la forma más sencilla de consultar documentos. Por supuesto, también puedes crear consultas más complejas.

Usando el generador de consultas

El *ODM* de *Doctrine* viene con un objeto "Generador" de consultas, el cual te permite construir una consulta para exactamente los documentos que deseas devolver. Si usas un *IDE*, también puedes tomar ventaja del autocompletado a medida que escribes los nombres de métodos. Desde el interior de un controlador:

```
$productos = $this->get('doctrine.odm.mongodb.document_manager')
    ->createQueryBuilder('AcmeTiendaBundle:Producto')
    ->field('nombre')->equals('foo')
    ->limit(10)
    ->sort('precio', 'ASC')
    ->getQuery()
    ->execute()
```

En este caso, devuelve 10 productos con el nombre "foo", ordenados de menor a mayor precio.

El objeto `QueryBuilder` contiene todos los métodos necesarios para construir tu consulta. Para más información sobre el generador de consultas de *Doctrine*, consulta la documentación del Generador de consultas de *Doctrine*. Para una lista de las condiciones disponibles que puedes colocar en la consulta, ve la documentación específica a los Operadores condicionales.

Repositorio de clases personalizado

En la sección anterior, comenzaste a construir y utilizar consultas más complejas desde el interior de un controlador. A fin de aislar, probar y reutilizar esas consultas, es buena idea crear una clase repositorio personalizada para tu documento y allí agregar métodos con la lógica de la consulta.

Para ello, agrega el nombre de la clase del repositorio a la definición de asignación.

- *Annotations*

```
// src/Acme/TiendaBundle/Document/Producto.php
namespace Acme\TiendaBundle\Document;

use Doctrine\ODM\MongoDB\Mapping\Annotations as MongoDB;

/**
 * @MongoDB\Document(repositoryClass="Acme\TiendaBundle\Repository\ProductoRepository")
 */
class Producto
{
    //...
}
```

- *YAML*

```
# src/Acme/TiendaBundle/Resources/config/doctrine/Producto.mongodb.yml
Acme\TiendaBundle\Document\Producto:
    repositoryClass: Acme\TiendaBundle\Repository\ProductoRepository
    # ...
```

- *XML*

```
<!-- src/Acme/TiendaBundle/Resources/config/doctrine/Producto.mongodb.xml -->
<!-- ... -->
<doctrine-mongo-mapping xmlns="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping
                  http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping.xsd">

    <document name="Acme\TiendaBundle\Document\Producto"
            repository-class="Acme\TiendaBundle\Repository\ProductoRepository">
        <!-- ... -->
    </document>

</doctrine-mong-mapping>
```

Doctrine puede generar la clase repositorio para ti ejecutando:

```
php app/console doctrine:mongodb:generate:repositories AcmeTiendaBundle
```

A continuación, agrega un nuevo método - `findAllOrderedByName()` - a la clase repositorio recién generada. Este método deberá consultar por todos los documentos `Producto`, ordenados alfabéticamente.

```
// src/Acme/TiendaBundle/Repository/ProductoRepository.php
namespace Acme\TiendaBundle\Repository;

use Doctrine\ODM\MongoDB\DocumentRepository;

class ProductoRepository extends DocumentRepository
{
    public function findAllOrderedByName()
    {
        return $this->createQueryBuilder()
            ->sort('nombre', 'ASC')
            ->getQuery()
            ->execute();
    }
}
```

Puedes utilizar este nuevo método al igual que los métodos de búsqueda predeterminados del repositorio:

```
$producto = $this->get('doctrine.odm.mongodb.document_manager')
    ->getRepository('AcmeTiendaBundle:Producto')
    ->findAllOrderedByName();
```

Nota: Al utilizar una clase repositorio personalizada, todavía tienes acceso a los métodos de búsqueda predeterminados como `find()` y `findAll()`.

5.6.7 Extensiones *Doctrine*: `Timestampable`, `Sluggable`, etc.

Doctrine es bastante flexible, y dispone de una serie de extensiones de terceros que te permiten realizar fácilmente tareas repetitivas y comunes en tus entidades. Estas incluyen cosas tales como `Sluggable`, `Timestampable`, `registrable`, `traducible` y `Tree`.

Para más información sobre cómo encontrar y utilizar estas extensiones, ve el artículo sobre el uso de *extensiones comunes de *Doctrine** (Página 300).

5.6.8 Referencia de tipos de campo *Doctrine*

Doctrine dispone de una gran cantidad de tipos de campo. Cada uno de estos asigna un tipo de dato *PHP* a un determinado tipo de MongoDB. Los siguientes son sólo *algunos* de los tipos admitidos por *Doctrine*:

- `string`
- `int`
- `float`
- `date`
- `timestamp`
- `boolean`
- `file`

Para más información, consulta la sección Asignando tipos en la documentación de *Doctrine*.

5.6.9 Ordenes de consola

La integración *ODM* de *Doctrine2* ofrece varias ordenes de consola en el espacio de nombres `doctrine:mongodb`. Para ver la lista de ordenes puedes ejecutar la consola sin ningún tipo de argumento:

```
php app/console
```

Mostrará una lista con las ordenes disponibles, muchas de las cuales comienzan con el prefijo `doctrine:mongodb`. Puedes encontrar más información sobre cualquiera de estas ordenes (o cualquier orden de *Symfony*) ejecutando la orden `help`. Por ejemplo, para obtener detalles acerca de la tarea `doctrine:mongodb:query`, ejecuta:

```
php app/console help doctrine:mongodb:query
```

Nota: Para poder cargar accesorios en *MongoDB*, necesitas tener instalado el paquete `DoctrineFixturesBundle`. Para saber cómo hacerlo, lee el artículo "*DoctrineFixturesBundle* (Página 592)" de la documentación.

5.6.10 Configurando

Para información más detallada sobre las opciones de configuración disponibles cuando utilizas el *ODM* de *Doctrine*, consulta la sección *Referencia MongoDB* (Página 601).

Registrando escuchas y suscriptores de eventos

Doctrine te permite registrar escuchas y suscriptores que recibirán una notificación cuando se produzcan diferentes eventos al interior del *ODM Doctrine*. Para más información, consulta la sección Documentación de eventos de *Doctrine*.

En *Symfony*, puedes registrar un escucha o suscriptor creando un *servicio* y, a continuación *marcarlo* (Página 251) con una etiqueta específica.

- **escucha de eventos**: Usa la etiqueta `doctrine.odm.mongodb.<conexión>_event_listener`, donde el nombre `<conexión>` es sustituido por el nombre de la conexión (por lo general `default`). Además, asegúrate de agregar una clave `evento` para la etiqueta que especifica el evento que escucha. Suponiendo que la conexión se llama `default`, entonces:

- *YAML*

```
services:
    mi_escucha_doctrine:
        class:   Acme\HolaBundle\Listener\MiEscuchaDoctrine
        # ...
        tags:
            -  { name: doctrine.odm.mongodb.default_event_listener, event: postPersist }
```

- *XML*

```
<service id="mi_escucha_doctrine" class="Acme\HolaBundle\Listener\MiEscuchaDoctrine">
    <!-- ... -->
    <tag name="doctrine.odm.mongodb.default_event_listener" event="postPersist" />
</service>.
```

- *PHP*

```
$definicion = new Definition('Acme\HolaBundle\Listener\MiEscuchaDoctrine');
// ...
$definicion->addTag('doctrine.odm.mongodb.default_event_listener');
$contenedor->setDefinition('mi_escucha_doctrine', $definicion);
```

- **Suscriptor de evento**: Utiliza la etiqueta `doctrine.odm.mongodb.<conexión>_event_subscriber`. Ninguna otra clave es necesaria en la etiqueta.

5.6.11 Resumen

Con *Doctrine*, te puedes enfocar en los objetos y la forma en que son útiles en tu aplicación y en segundo lugar preocuparte por la persistencia a través de *MongoDB*. Esto se debe a que *Doctrine* te permite utilizar cualquier objeto *PHP* para almacenar los datos y confía en la información de asignación de metadatos para asignar los datos de un objeto a una colección *MongoDB*.

Y aunque *Doctrine* gira en torno a un concepto simple, es increíblemente poderoso, permitiéndote crear consultas complejas y suscribirte a los eventos que te permiten realizar diferentes acciones conforme los objetos recorren su ciclo de vida en la persistencia.

5.6.12 Aprende más en el recetario

- *Cómo implementar un sencillo formulario de inscripción con MongoDB* (Página 604)
- *SensioFrameworkExtraBundle* (Página 577)
- *SensioGeneratorBundle* (Página 585)
- *JMSSecurityExtraBundle* (Página 588)
- *DoctrineFixturesBundle* (Página 592)
- *DoctrineMigrationsBundle* (Página 597)
- *DoctrineMongoDBBundle* (Página 601)
- *SensioFrameworkExtraBundle* (Página 577)
- *SensioGeneratorBundle* (Página 585)
- *JMSSecurityExtraBundle* (Página 588)

- *DoctrineFixturesBundle* (Página 592)
- *DoctrineMigrationsBundle* (Página 597)
- *DoctrineMongoDBBundle* (Página 601)

Parte VI

Colaborando

Colabora con *Symfony2*:

CAPÍTULO 6

Colaborando

6.1 Aportando código

6.1.1 Informando de errores

Cada vez que encuentres un error en *Symfony2*, te rogamos que lo informes. Nos ayuda a hacer un mejor *Symfony2*.

Prudencia: Si piensas que has encontrado un problema de seguridad, por favor, en su lugar, usa el *procedimiento* (Página 627) especial.

Antes de enviar un error:

- Revisa cuidadosamente la documentación oficial para ver si no estás usando incorrectamente la plataforma;
- Pide ayuda en la lista de correo de usuarios, el foro, o en el canal IRC #symfony si no estás seguro de que el problema realmente sea un error.

Si tu problema definitivamente se ve como un error, informa el fallo usando el rastreador oficial siguiendo algunas reglas básicas:

- Utiliza el campo título para describir claramente el problema;
- Describe los pasos necesarios para reproducir el error con breves ejemplos de código (proporcionando una prueba de unidad que ilustre mejor el error);
- Indica lo más detalladamente posible tu entorno (sistema operativo, versión de *PHP*, versión de *Symfony*, extensiones habilitadas, ...);
- *(opcional)* Adjunta un *parche* (Página 625).

6.1.2 Enviando un parche

Los parches son la mejor manera de proporcionar una corrección de error o de proponer mejoras a *Symfony2*.

Configuración inicial

Antes de trabajar en *Symfony2*, configura un entorno amigable con el siguiente software:

- Git;
- *PHP* versión 5.3.2 o superior;
- *PHPUnit 3.5.11* o superior.

Configura la información del usuario con tu nombre real y una dirección de correo electrónico operativa:

```
$ git config --global user.name "Tu nombre"
$ git config --global user.correo tu@ejemplo.com
```

Truco: Si eres nuevo en Git, es muy recomendable que leas el excelente libro ProGit, que además es libre.

Obtén el código fuente de *Symfony2*:

- Crea una cuenta GitHub e ingresa;
- Consigue el repositorio Symfony2 (haz clic en el botón "Fork");
- Después de concluida la "acción de bifurcar el núcleo", clona tu bifurcación a nivel local (esto creará un directorio *symfony*):

```
$ git clone git@github.com:NOMBREUSUARIO/symfony.git
```

- Añade el repositorio anterior como `remoto`:

```
$ cd symfony
$ git remote add upstream git://github.com/symfony/symfony.git
```

Ahora que *Symfony2* está instalado, comprueba que todas las pruebas unitarias pasan en tu entorno como se explica en el *documento* (Página 628) dedicado.

Trabajando en un parche

Cada vez que desees trabajar en un parche por un fallo o una mejora, necesitas crear una rama del tema:

La rama debe estar basada en la rama `master` si deseas agregar una nueva función. Pero si deseas corregir un fallo, utiliza la versión más antigua, pero aún mantenida de *Symfony* donde ocurre el fallo (como `2.0`).

Crea la rama del tema con la siguiente orden:

```
$ git checkout -b NOMBRE_RAMA master
```

Truco: Utiliza un nombre descriptivo para tu rama (`tarjeta_XXX` donde `XXX` es el número de tarjeta es una buena convención para la corrección de errores).

La orden anterior automáticamente cambia el código de la rama recién creada (para ver la sección en que estás trabajando utiliza `git branch`).

Trabaja en el código tanto como quieras y consigna tanto como desees, pero ten en cuenta lo siguiente:

- Sigue los *estándares* (Página 629) de codificación (utiliza `git diff --check` para comprobar si hay espacios finales);
- Añade pruebas unitarias para probar que el error se corrigió o que la nueva característica realmente funciona;
- Haz consignaciones atómicas y separadas lógicamente (usa el poder de `git rebase` para tener un historial limpio y lógico);
- Escribe mensajes de consignación sustanciales.

Truco: Un buen mensaje de consignación sustancial está compuesto por un resumen (la primera línea), seguido opcionalmente por una línea en blanco y una descripción más detallada. El resumen debe comenzar con el componente en el que estás trabajando entre corchetes (`[DependencyInjection]`, `[FrameworkBundle]`, ...). Utiliza un verbo (`fixed...`, `added...`, ...) para iniciar el resumen y no agregues un punto al final.

Enviando un parche

Antes de presentar tu revisión, actualiza tu rama (es necesario si te toma cierto tiempo terminar los cambios):

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
$ git checkout NOMBRE_RAMA
$ git rebase master
```

Al ejecutar la orden `rebase`, posiblemente tengas que arreglar conflictos de fusión. `git status` te mostrará los archivos *sin fusionar*. Resuelve todos los conflictos, y luego continua el rebase:

```
$ git add ... # Añade archivos resueltos
$ git rebase --continue
```

Comprueba que todas las pruebas todavía pasan y empuja tu rama remota:

```
$ git push origin NOMBRE_RAMA
```

Ahora puedes hablar de tu parche en la lista de correo dev o hacer una petición de atracción (que se debe hacer en el repositorio `symfony/synfony`). Para facilitar el trabajo del equipo central, siempre incluye los componentes modificados en tu mensaje de petición de atracción, como en:

```
[Yaml] foo bar
[Form] [Validator] [FrameworkBundle] foo bar
```

Si vas a enviar un correo electrónico a la lista de correo, no olvides hacer referencia a la *URL* de tu rama (`https://github.com/NOMBREDEUSUARIO/symfony.git NOMBRE_RAMA`) o la *URL* de la petición de atracción.

Basándote en la retroalimentación de la lista de correo o a través de la petición de atracción en GitHub, posiblemente tengas que rehacer el parche. Antes de volver a presentar la revisión, rebasa con el maestro, sin fusionar y fuerza el empuje al origen:

```
$ git rebase -f upstream/master
$ git push -f origin NOMBRE_RAMA
```

Nota: Todos los parches que envíes se deben liberar bajo la licencia MIT, a menos que explícitamente lo especifiques en el código.

Todas las correcciones de fallos se fusionaran en las ramas de mantenimiento también se fusionaran regularmente en las ramas más recientes. Por ejemplo, si envías un parche para la rama `2.0`, el parche también será aplicado por el equipo central a la rama `master`.

6.1.3 Informando un problema de seguridad

¿Encontraste un problema de seguridad en *Symfony2*? No utilices la lista de correo o el gestor de fallos. Todas las cuestiones de seguridad, en su lugar se deben enviar a **security [arroba] symfony-project.com**. Los correos electrónicos

enviados a esta dirección se reenvían a la lista de correo privado del equipo del núcleo.

Para cada informe, en primer lugar, trata de confirmar la vulnerabilidad. Cuando esté confirmada, el equipo del núcleo trabaja en una solución siguiendo estos pasos:

1. Envía un reconocimiento al informante;
2. Trabaja en un parche;
3. Escribe un post que describe la vulnerabilidad, las posibles explotaciones, y cómo aplicar el parche/actualizar las aplicaciones afectadas;
4. Aplica el parche en todas las versiones mantenidas de *Symfony*;
5. Publica el post en el blog oficial de *Symfony*.

Nota: Mientras estemos trabajando en un parche, por favor, no reveles el tema públicamente.

6.1.4 Corriendo las pruebas de *Symfony2*

Antes de presentar un *parche* (Página 625) para su inclusión, es necesario ejecutar el banco de pruebas *Symfony2* para comprobar que no ha roto nada.

PHPUnit

Para ejecutar el banco de pruebas de *Symfony2*, primero instala PHPUnit 3.5.0 o superior:

```
$ pear channel-discover pear.phpunit.de
$ pear channel-discover components.ez.no
$ pear channel-discover pear.symfony-project.com
$ pear install phpunit/PHPUnit
```

Dependencias (opcional)

Para ejecutar el banco de pruebas completo, incluyendo las pruebas supeditadas con dependencias externas, *Symfony2* tiene que ser capaz de cargarlas automáticamente. De forma predeterminada, se cargan automáticamente desde *vendor/* en la raíz del directorio principal (consulta la sección *autoload.php.dist*).

El banco de pruebas necesita las siguientes bibliotecas de terceros:

- *Doctrine*
- *Swiftmailer*
- *Twig*
- *Monolog*

Para instalarlas todas, ejecuta el archivo *vendors.php*:

```
$ php vendors install
```

Nota: Ten en cuenta que el guión toma algún tiempo para terminar.

Después de la instalación, puedes actualizar los proveedores en cualquier momento con la siguiente orden.

```
$ php vendors update
```

Ejecutando

En primer lugar, actualiza los proveedores (consulta más arriba).

A continuación, ejecuta el banco de pruebas desde el directorio raíz de *Symfony2* con la siguiente orden:

```
$ phpunit
```

La salida debe mostrar *OK*. Si no es así, es necesario averiguar qué está pasando y si las pruebas se rompen a causa de tus modificaciones.

Truco: Ejecuta el banco de pruebas antes de aplicar las modificaciones para comprobar que funcionan bien en tu configuración.

Cobertura de código

Si agregas una nueva característica, también necesitas comprobar la cobertura de código usando la opción *coverage-html*:

```
$ phpunit --coverage-html=cov/
```

Verifica la cobertura de código abriendo en un navegador la página generada *cov/index.html*.

Truco: La cobertura de código sólo funciona si tienes activado XDebug e instaladas todas las dependencias.

6.1.5 Estándares de codificación

Cuando aportes código a *Symfony2*, debes seguir sus estándares de codificación. Para no hacer el cuento largo, aquí está la regla de oro: **Imita el código Symfony2 existente**. La mayoría de los Paquetes de código abierto y librerías utilizadas por *Symfony2* también siguen las mismas pautas, y también deberías hacerlo.

Recuerda que la principal ventaja de los estándares es que cada pieza de código se ve y se siente familiar, no se trata de tal o cual sea más legible.

Ya que una imagen - o algún código - vale más que mil palabras, he aquí un pequeño ejemplo que contiene la mayoría de las funciones que se describen a continuación:

```
<?php

/*
 * Este archivo es parte del paquete Symfony.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * Para obtener la información completa de derechos de autor y licencia,
 * consulta el archivo LICENSE que se distribuye con el código fuente.
 */

namespace Acme;

class Foo
```

```
{
    const ALGUNA_CONST = 42;

    private $foo;

    /**
     * @param string $ficticio Alguna descripción del argumento
     */
    public function __construct($ficticio)
    {
        $this->foo = $this->transform($ficticio);
    }

    /**
     * @param string $ficticio Alguna descripción del argumento
     * @return string|null Entrada transformada
     */
    private function transform($ficticio)
    {
        if (true === $ficticio) {
            return;
        } elseif ('string' === $ficticio) {
            $ficticio = substr($ficticio, 0, 5);
        }

        return $ficticio;
    }
}
```

Estructura

- Nunca utilices las etiquetas cortas (<*?*>);
- No termines los archivos de clase con la etiqueta de cierre habitual *?*>;
- La sangría se hace por pasos de cuatro espacios (las tabulaciones no están permitidas);
- Utiliza el carácter de salto de línea (*0x0A*) para terminar las líneas;
- Añade un solo espacio después de cada delimitador coma;
- No pongas espacios después de un paréntesis de apertura ni antes de uno de cierre;
- Añade un solo espacio alrededor de los operadores (==, *&&*, ...);
- Añade un solo espacio antes del paréntesis de apertura de una palabra clave de control (*if*, *else*, *for*, *while*, ...);
- Añade una línea en blanco antes de las declaraciones *return*;
- No agregues espacios en blanco al final de las líneas;
- Usa llaves para indicar la estructura del cuerpo de control, independientemente del número de declaraciones que contenga;
- Coloca las llaves en su propia línea para clases, métodos y en la declaración de funciones;
- Separa las declaraciones condicionales (`if`, `else`, ...) y la llave de apertura con un solo espacio y sin ninguna línea en blanco;
- Declara expresamente la visibilidad para clases, métodos y propiedades (el uso de *var* está prohibido);

- Usa minúsculas para escribir las constantes nativas de *PHP*: `false`, `true` y `null`. Lo mismo ocurre con *array()*;
- Usa cadenas en mayúsculas para constantes con palabras separadas por subrayados;
- Define una clase por archivo;
- Declara las propiedades de clase antes que los métodos;
- Declara los métodos públicos en primer lugar, a continuación, los protegidos y finalmente los privados.

Convenciones de nomenclatura

- Usa mayúsculas intercaladas, no subrayados, para variables, funciones y nombres de métodos;
- Usa subrayas para opciones, argumentos, nombre de parámetros;
- Utiliza espacios de nombres (namespaces) para todas las clases;
- Utiliza *Symfony* como primer nivel del espacio de nombres;
- Sufija interfaces con *Interface*;
- Utiliza caracteres alfanuméricos y subrayados para los nombres de archivo;
- No olvides consultar en el documento más detallado *Convenciones* (Página 631) para más consideraciones subjetivas de nomenclatura.

Documentación

- Añade bloques PHPDoc a todas las clases, métodos y funciones;
- Las anotaciones *@package* y *@subpackage* no se utilizan.

Licencia

- *Symfony* se distribuye bajo la licencia *MIT*, y el bloque de la licencia tiene que estar presente en la parte superior de todos los archivos *PHP*, antes del espacio de nombres.

6.1.6 Convenciones

El documento *estándares* (Página 629) describe las normas de codificación de los proyectos *Symfony2* y los paquetes internos de terceros. Este documento describe los estándares de codificación y convenciones utilizadas en la plataforma básica para que sea más consistente y predecible. Los puedes seguir en tu propio código, pero no es necesario.

Nombres de método

Cuando un objeto tiene una relación "principal" con muchas "cosas" relacionadas (objetos, parámetros, ...), los nombres de los métodos están normalizados:

- `get()`
- `set()`
- `has()`
- `all()`

- `replace()`
- `remove()`
- `clear()`
- `isEmpty()`
- `add()`
- `register()`
- `count()`
- `keys()`

El uso de estos métodos sólo se permite cuando es evidente que existe una relación principal:

- un `CookieJar` tiene muchos objetos `Cookie`;
- un `Container` de servicio tiene muchos servicios y muchos parámetros (debido a que servicios es la relación principal, utilizamos la convención de nomenclatura para esta relación);
- una Consola `Input` tiene muchos argumentos y muchas opciones. No hay una relación "principal", por lo tanto no aplica la convención de nomenclatura.

Para muchas relaciones cuando la convención no aplica, en su lugar se deben utilizar los siguientes métodos (donde `XXX` es el nombre de aquello relacionado):

Relación principal	Otras relaciones
`get()`	`getXXX()`
`set()`	`setXXX()`
n/a	`replaceXXX()`
`has()`	`hasXXX()`
`all()`	`getXXXs()`
`replace()`	`setXXXs()`
`remove()`	`removeXXX()`
`clear()`	`clearXXX()`
`isEmpty()`	`isEmptyXXX()`
`add()`	`addXXX()`
`register()`	`registerXXX()`
`count()`	`countXXX()`
`keys()`	n/a

Nota: Si bien "setXXX" y "replaceXXX" son muy similares, hay una notable diferencia: "setXXX" puede sustituir o agregar nuevos elementos a la relación. Por otra parte "replaceXXX" específicamente tiene prohibido añadir nuevos elementos, pero mayoritariamente lanza una excepción en estos casos.

6.1.7 Licencia *Symfony2*

Symfony2 se libera bajo la licencia MIT.

De acuerdo con Wikipedia:

> "Es una licencia permisiva, lo cual significa que permite la reutilización dentro de software propietario a condición de que la licencia se distribuya con el software. Esta también es compatible con la licencia GPL, lo cual significa que la licencia GPL permite la combinación y redistribución, con el software que utiliza la licencia MIT".

La Licencia

6.2 Aportando documentación

6.2.1 Colaborando en la documentación

La documentación es tan importante como el código. Esta sigue exactamente los mismos principios: una vez y sólo una, pruebas, facilidad de mantenimiento, extensibilidad, optimización y reconstrucción sólo por nombrar algunos. Y, por supuesto, la documentación tiene errores, errores tipográficos, guías difíciles de leer y mucho más.

Colaborando

Antes de colaborar, necesitas familiarizarte con: el *lenguaje de marcado* (Página 634) empleado en la documentación.

La documentación *Symfony2* se encuentra alojada en GitHub:

```
https://github.com/symfony/symfony-docs
```

Si deseas enviar un parche bifurcado al repositorio oficial en GitHub y luego reproducir tu bifurcación:

```
$ git clone git://github.com/TUNOMBRE/symfony-docs.git
```

A continuación, crea una rama dedicada a tus cambios (para mantener la organización):

```
$ git checkout -b mejorando_foo_y_bar
```

Ahora puedes hacer los cambios directamente en esta rama y consignarlos ahí. Cuando hayas terminado, impulsa esta rama a *tu* GitHub e inicia una petición de atracción. La petición de atracción debe ser entre tu rama `mejorando_foo_y_bar` y la rama `maestra` de `Symfony-docs`.

GitHub aborda el tema de las peticiones de atracción en detalle.

Nota: La documentación de *Symfony2* está bajo una licencia Creative Commons Attribution-Share Alike 3.0 Unported *Licencia* (Página 647).

Informando un problema

La contribución más fácil que puedes hacer es informar algún problema: un error, un error gramatical, un error en el código de ejemplo, una explicación omitida, y así sucesivamente.

Pasos a seguir:

- Reporta un error en el rastreador de errores;
- *(Opcional)* Envía un parche.

Traduciendo

Lee el *documento dedicado* (Página 636).

6.2.2 Formato de documentación

La documentación *Symfony2* utiliza reStructuredText como lenguaje de marcado y Sphinx para construir la documentación (en *HTML*, *PDF*, ...).

reStructuredText

reStructuredText "es un sistema analizador y sintaxis de marcado de texto, fácil de leer, lo que ves es lo que obtienes".

Puedes aprender más acerca de su sintaxis leyendo los documentos existentes de *Symfony2* o leyendo el Primer reStructuredText en el sitio web de Sphinx.

Si estás familiarizado con Markdown, ten cuidado que las cosas a veces se ven muy similares, pero son diferentes:

- Las listas se inician al comienzo de una línea (no permite sangría);
- Los bloques de código en línea utilizan comillas dobles (``` ``como estas`` ```).

Sphinx

Sphinx es un sistema de construcción que añade algunas herramientas agradables para crear la documentación desde documentos reStructuredText. Como tal, agrega nuevas directivas e interpreta texto en distintos roles al marcado reST estándar.

Resaltado de sintaxis

Todo el código de los ejemplos utiliza PHP como resaltado del lenguaje por omisión. Puedes cambiarlo con la directiva `code-block`:

```
.. code-block:: yaml

    { foo: bar, bar: { foo: bar, bar: baz } }
```

Si el código *PHP* comienza con `<?php`, entonces necesitas usar `html+php` como pseudolenguaje a resaltar:

```
.. code-block:: html+php

    <?php echo $this->foobar(); ?>
```

Nota: Una lista de lenguajes apoyados está disponible en el sitio web Pygments.

Bloques de configuración

Cada vez que muestres una configuración, debes utilizar la directiva `configuration-block` para mostrar la configuración en todos los formatos de configuración compatibles (`YAML`, `XML` y `PHP`)

```
.. configuration-block::

    .. code-block:: yaml

        # Configuración en YAML

    .. code-block:: xml

        <!-- Configuración en XML //-->

    .. code-block:: php

        // Configuración en PHP
```

El fragmento reST anterior se reproduce de la siguiente manera:

- *YAML*

  ```
  # Configuración en YAML
  ```

- *XML*

  ```
  <!-- Configuración en XML //-->
  ```

- *PHP*

  ```
  // Configuración en PHP
  ```

La lista de formatos apoyados actualmente es la siguiente:

Formato de marcado	Muestra
html	*HTML*
xml	*XML*
php	*PHP*
yaml	*YAML*
jinja	*Twig*
html+jinja	*Twig*
jinja+html	*Twig*
php+html	*PHP*
html+php	*PHP*
ini	*INI*
php-annotations	Anotaciones

Probando la documentación

Para probar la documentación antes de consignarla:

- Instala Sphinx;
- Ejecuta el programa de instalación rápida de Sphinx;
- Instala la extensión *configuration-block* de Sphinx (ver más abajo);
- Ejecuta `make html` y revisa el código *HTML* generado en el directorio `build`.

Instalando la extensión configuration-block de Sphinx

- Descarga la extensión desde el repositorio fuente de configuration-block
- Copia el `configurationblock.py` al directorio `_exts` bajo tu directorio fuente (donde está ubicado `conf.py`)
- Agrega lo siguiente al archivo `conf.py`:

```
# ...
sys.path.append(os.path.abspath('_exts'))

# ...
# agrega configurationblock a la lista de extensiones
extensions = ['configurationblock']
```

6.2.3 Traduciendo

La documentación de *Symfony2* está escrita en Inglés y hay muchas personas involucradas en el proceso de traducción.

Colaborando

En primer lugar, familiarízate con el *lenguaje de marcado* (Página 634) empleado en la documentación.

En segundo lugar, sigue las *buenas prácticas de la traducción profesional* (Página 637), utilizando las mejores herramientas libres disponibles hoy día.

A continuación, suscríbete a la lista de correo docs de Symfony, debido a que la colaboración sucede allí.

Por último, busca el repositorio *maestro* para el idioma con el que deseas contribuir. Esta es la lista oficial de los repositorios *maestros*:

- *Inglés*: http://github.com/symfony/symfony-docs
- *Ruso*: http://github.com/avalanche123/symfony-docs-ru
- *Rumano*: http://github.com/sebio/symfony-docs-ro
- *Japonés*: https://github.com/symfony-japan/symfony-docs-ja

---

Nota: Si quieres contribuir traducciones para un nuevo lenguaje, lee la *sección dedicada* (Página 637).

Uniéndote al equipo de traducción

Si quieres ayudar a traducir algunos documentos a tu idioma o corregir algunos errores, considera unirte, es un proceso muy sencillo:

- Preséntate en la lista de correo docs de Symfony;
- *(opcional)* Solicita los documentos en que puedes trabajar;
- Bifurca el repositorio *principal* de tu idioma (haciendo clic en el botón "*Fork*" en la página GitHub);
- Traduce algunos documentos;
- Haz una petición de atracción (haciendo clic en la "petición de atracción" de tu página en GitHub);
- El director del equipo acepta tus modificaciones y las combina en el repositorio principal;
- El sitio web de documentación se actualiza todas las noches desde el repositorio principal.

Añadiendo un nuevo idioma

Esta sección ofrece algunas pautas para comenzar la traducción de la documentación de *Symfony2* para un nuevo idioma.

Debido a que iniciar una traducción conlleva mucho trabajo, habla acerca de tu plan en la lista de correo docs de Symfony y trata de encontrar personas motivadas dispuestas a ayudar.

Cuando el equipo esté listo, nomina un administrador del equipo, quién será el responsable del repositorio *maestro*.

Crea el repositorio y copia los documentos en *Inglés*.

El equipo ahora puede iniciar el proceso de traducción.

Cuando el equipo confíe en que el repositorio está en un estado coherente y estable (se ha traducido todo, o los documentos sin traducir se han retirado de los toctrees – archivos con el nombre `index.rst` y `map.rst.inc`), el administrador del equipo puede pedir que el repositorio se añada a la lista de repositorios *maestro* oficiales enviando un correo electrónico a Fabien (fabien arroba symfony.com).

Manteniendo

La traducción no termina cuando se ha traducido todo. La documentación es un objeto en movimiento (se agregan nuevos documentos, se corrigen errores, se reorganizan párrafos, ...). El equipo de traducción tiene que seguir de cerca los cambios del repositorio en Inglés y aplicarlos a los documentos traducidos tan pronto como sea posible.

> **Prudencia:** Los idiomas sin mantenimiento se quitan de la lista oficial de repositorios de documentación puesto que la obsolescencia es peligrosa.

6.2.4 Traduciendo profesionalmente

Hasta ahora, el proceso de traducción - en lo que respecta a nuestros abnegados traductores - había sido muy pesado e incluso en ocasiones doloroso, puesto que se hacía **casi** a mano, era demasiado agotador tanto física como mentalmente, pero afortunadamente hoy día contamos con mejores herramientas especializadas en esta área y como bono adicional son **libres**.

En el proceso de traducir cualquier documento es imprescindible tener a nuestra disposición las mejores herramientas, en esta sección vamos a configurar algunas de ellas específicamente para traducir la documentación de *Symfony2*.

Nuestro arsenal

Nos vamos a armar con unas cuantas - de entre las muchas existentes - herramientas libres, específicamente necesitamos aprovisionarnos con suficiente equipo para:

Traducción Contamos con OmegaT (Página 638), una herramienta con memoria de traducción.

Control de versiones Tenemos el magnífico sistema GIT.

Revisión y corrección de errores Para facilitarnos la vida tenemos el sistema de texto a voz Balabolka en varios idiomas.

OmegaT

La herramienta libre de memoria de traducción OmegaT, utilizada para traducir la documentación, tiene las siguientes características que la convierten en la herramienta ideal para facilitar nuestro trabajo de traducción:

- Está escrita en Java, por lo tanto es multiplataforma.
- Reza un dicho popular: "Divide y vencerás", OmegaT segmenta el texto a traducir basándose principalmente en la puntuación gramatical, pero también puedes especificar tus propias reglas de segmentación.
- Usa memorias de traducción para almacenar los segmentos traducidos útiles para cualquier otro proyecto.
- No traduce por ti, pero cuenta con sugerencias de traducción proporcionadas por Google translate, Apertum y Belazar.
- Cuenta con propagación de coincidencias, es decir, los segmentos iguales se traducen una sola vez.
- Puedes escribir tus glosarios de términos usando OmegaT y también utilizar glosarios existentes.
- Nunca modifica los archivos fuente, al iniciar un proyecto sólo tienes que especificar el directorio raíz y la herramienta analiza todos los archivos en ese directorio y sus subdirectorios.
- En cualquier momento puedes crear los documentos finales (traducidos o no), OmegaT crea una estructura de directorios como la estructura original escribiendo los nuevos documentos traducidos basándose en los segmentos en la memoria de traducción.

Consiguiendo OmegaT

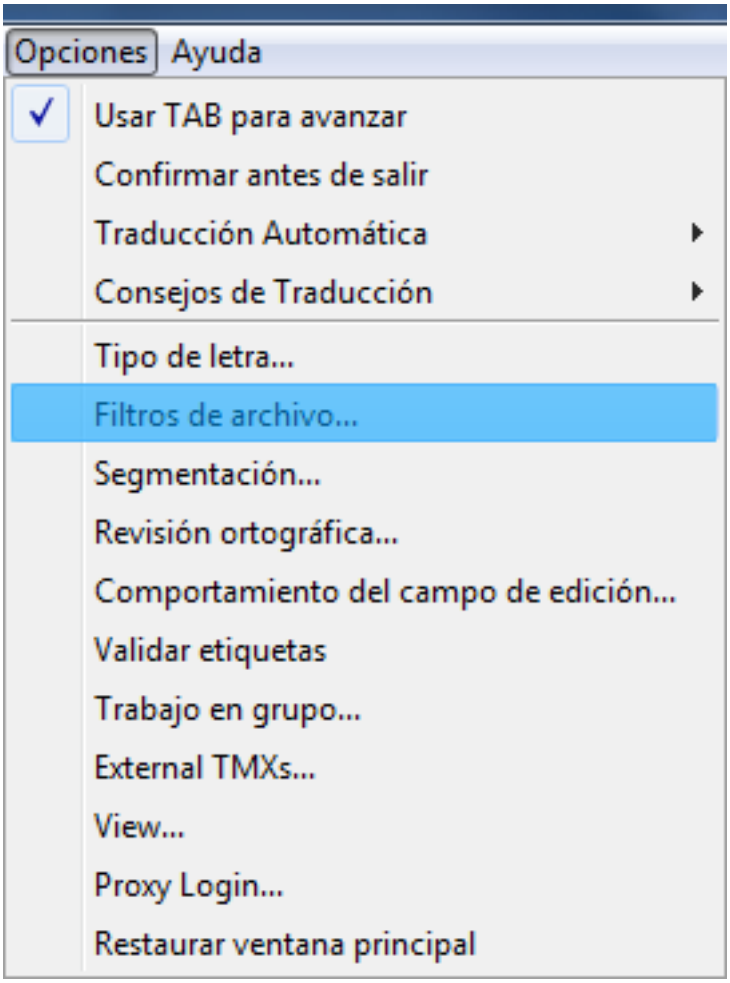
En primer lugar, descarga desde la página web de OmegaT (Página 638) la distribución para tu sistema operativo y sigue las instrucciones de instalación. Una vez instalado, al arrancar, lo primero que te muestra es una guía de inicio rápido, es muy recomendable que la leas para familiarizarte con la herramienta de traducción, además, puedes leer el manual que contiene información mucho más detallada.

Ajustando

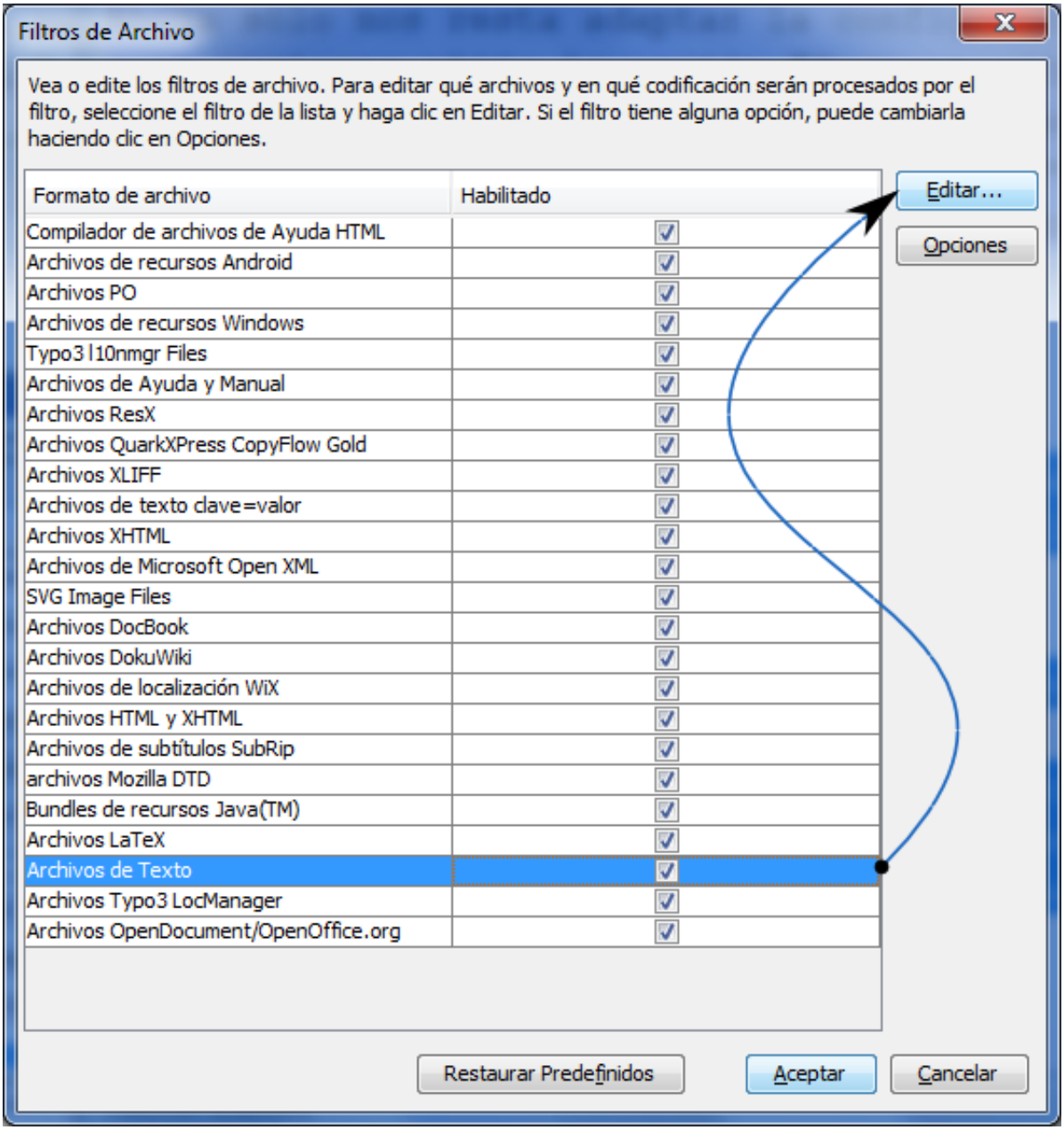
Ahora, es necesario realizar algunos pasos para hacer que OmegaT reconozca los archivos `*.rst` e `*.inc`. Debido a que OmegaT utiliza `filtros de archivo` basándose en el tipo de archivo y aún no hemos creado un filtro para los archivos `reStructuredText` (no tardaremos demasiado en ello), debemos cambiar un filtro existente para adaptarlo a nuestras necesidades:

Nota: Aquí explicamos la configuración en un sistema Windows, sin embargo, los usuarios de otros sistemas lo pueden adaptar fácilmente.

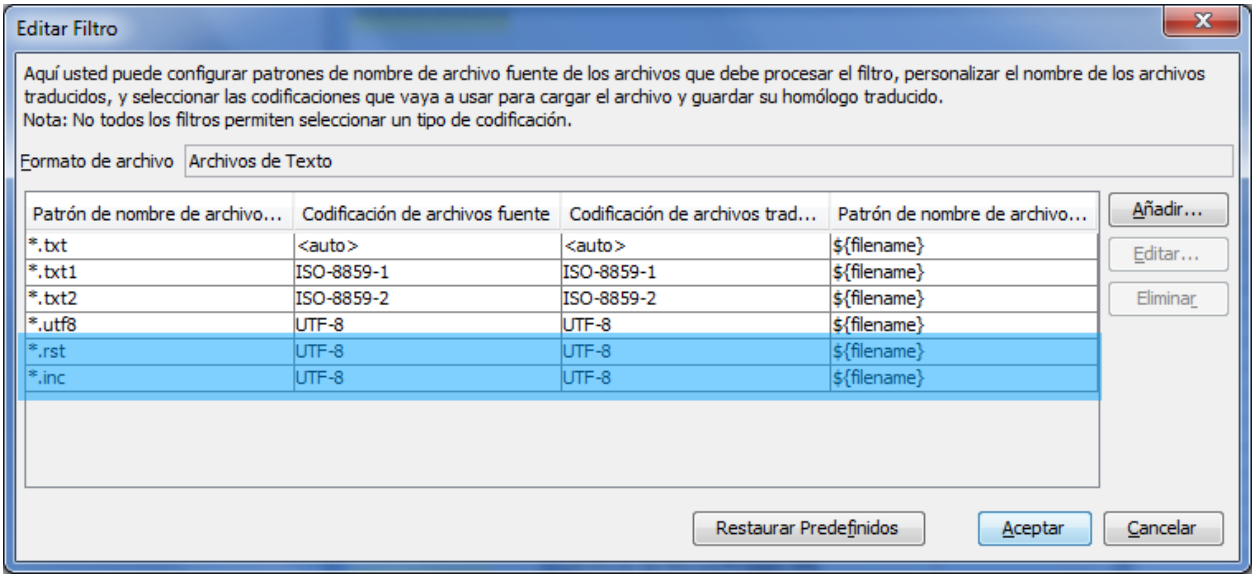
- En el menú, selecciona `OpcionesFiltros de archivo`.



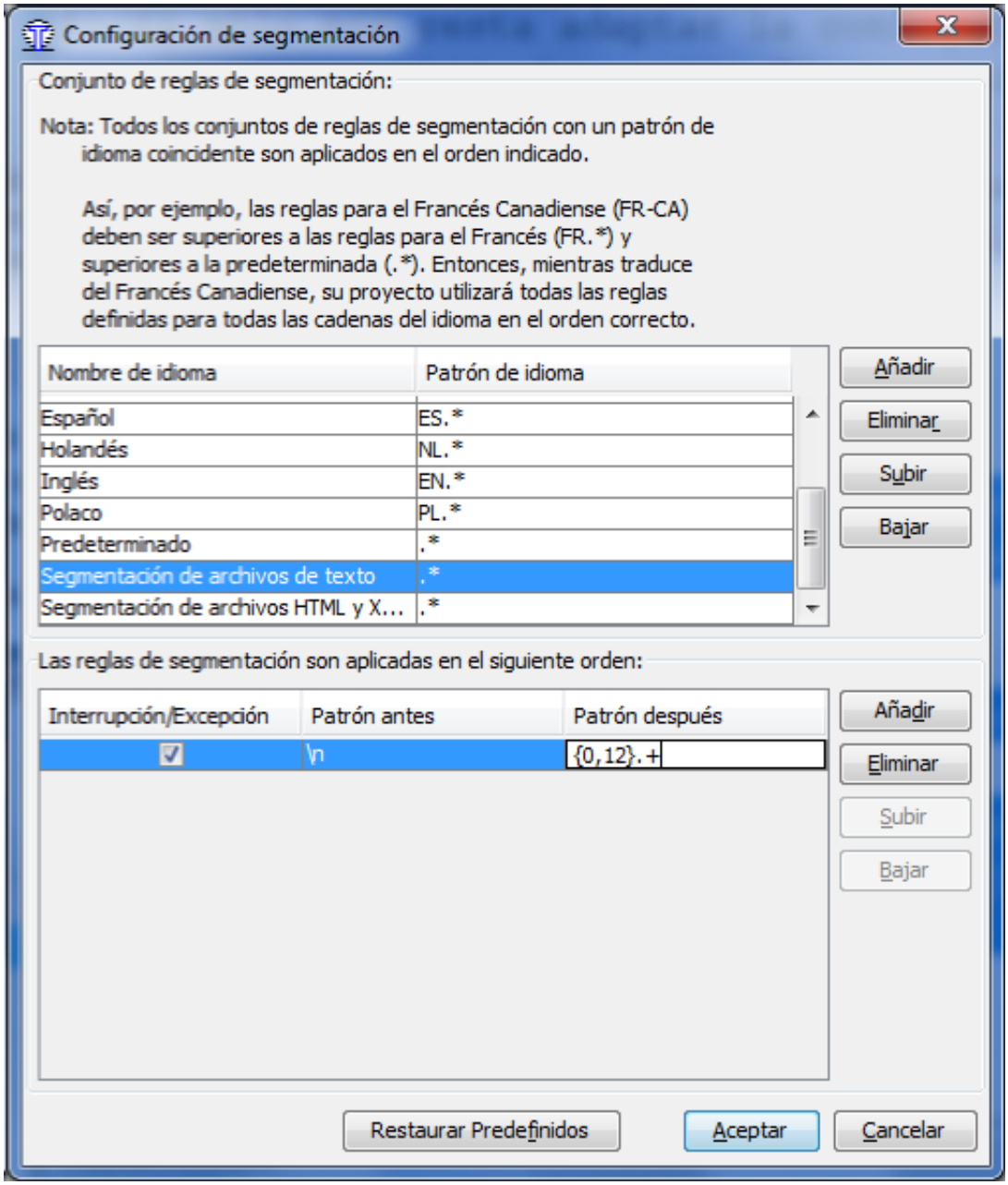
- En el cuadro de diálogo `Filtros de archivo` selecciona el filtro `Archivos de texto` y haz clic en `Editar`.



- A continuación, aparece un nuevo cuadro de diálogo, en este añade dos nuevos filtros con las extensiones `*.rst` e `*.inc`, selecciona `utf-8` como `Codificación de archivos fuente` y `traducidos` y haz clic en Aceptar para guardar los cambios.



Ahora sólo nos resta adaptar la configuración de una regla de segmentación, en el menú elije `OpcionesSegmentación`, te mostrará un cuadro de diálogo, en la lista superior selecciona `Segmentación de archivos de texto` y en la lista inferior cambia el `Patrón después` a **‘ {0,12}.+’** como muestra la siguiente imagen.

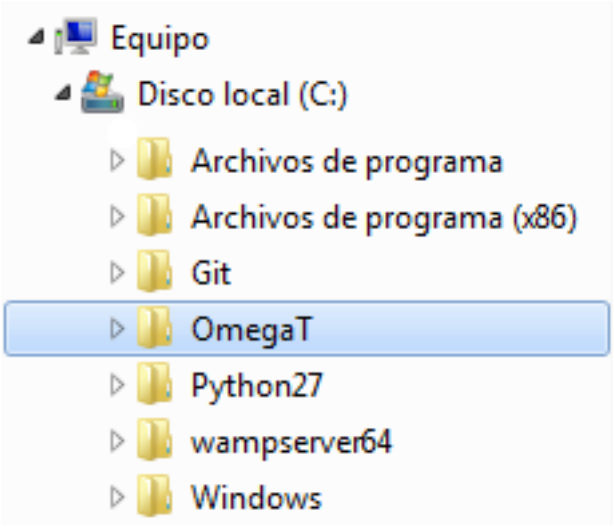


Nota: Puedes copiar y pegar el patrón, pero ten en cuenta que las comillas no forman parte de la expresión y que el primer carácter es un espacio en blanco.

¡Eso es todo! En cuanto respecta al filtro de archivos y segmentación, en el siguiente paso debemos conseguir los documentos originales (por supuesto en Inglés) como base para nuestra traducción.

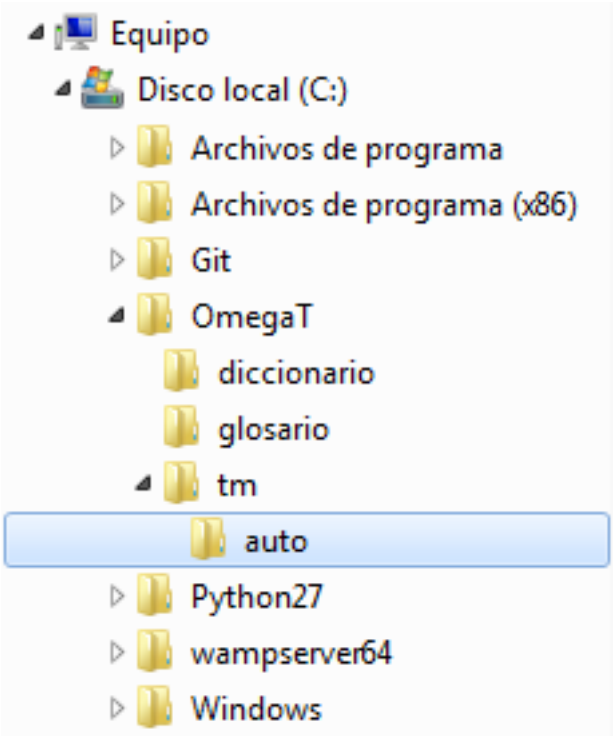
Creando el directorio del proyecto de traducción

Ahora vamos a crear un proyecto de traducción, a efecto de compartir las memorias de traducción con toda la documentación de *Symfony2* y paquetes adjuntos como *Twig* y *Doctrine*, vamos a crear un directorio que será la raíz de todos los proyectos de traducción, en este caso lo llamamos OmegaT:



Nota: Es muy importante que mantengas esta estructura de directorios, más adelante verás porqué.

Ahora, en el directorio raíz del proyecto de traducción, vamos a crear tres nuevos subdirectorios llamados `diccionario`, `glosario` y `tm`, además, dentro del subdirectorio `tm` vamos a crear un subdirectorio adicional, llamado `auto`, tal que así:



Cómo es de esperar de acuerdo a los nombres de los directorios, en cada uno de ellos colocaremos los archivos pertinentes, por ejemplo: en `glosario` colocaremos nuestros archivos de glosarios de términos, en `tm` las memorias de traducción y así sucesivamente.

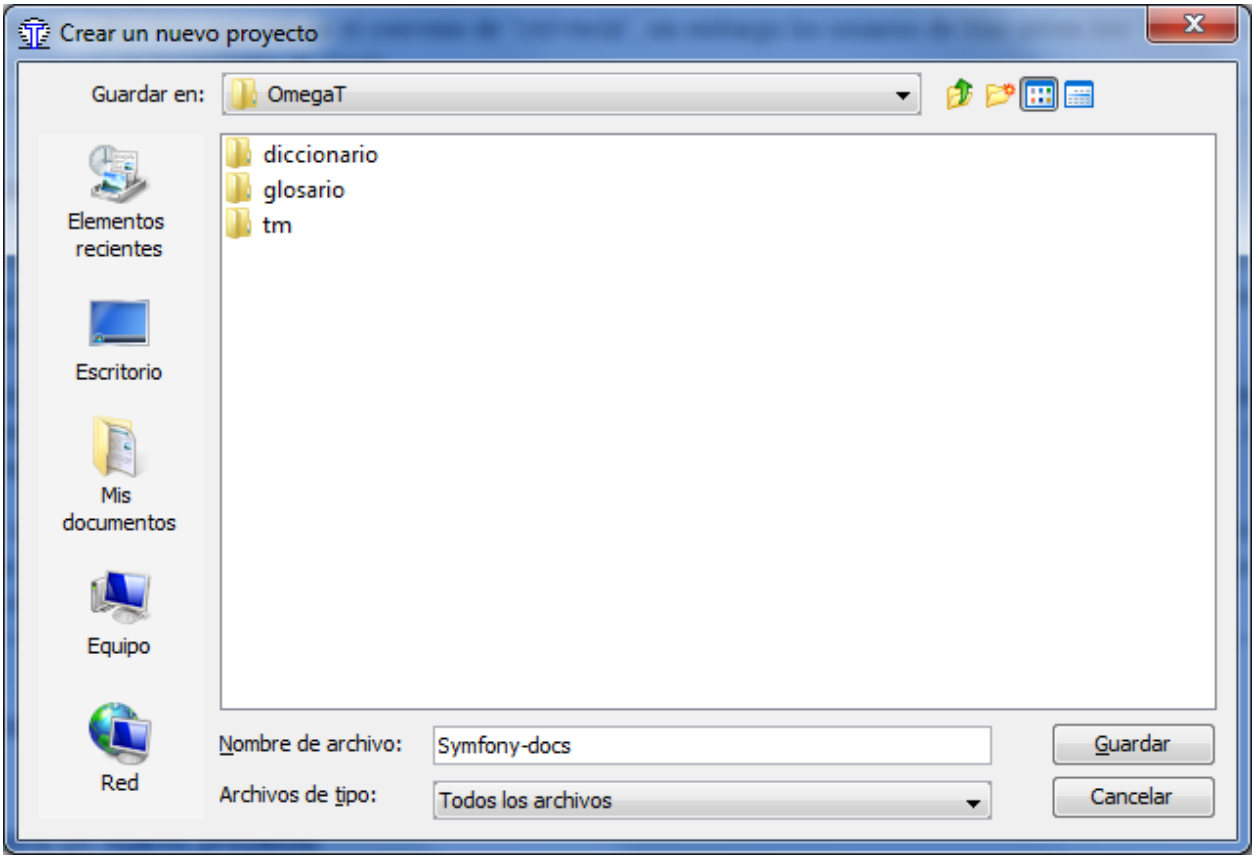
Sin embargo, cuando OmegaT encuentra una coincidencia en las memorias de traducción, automáticamente inserta un prefijo `[parcial]`, esta - en ocasiones - es una útil característica aunque a veces un tanto `molesta` porque tenemos que eliminar el prefijo manualmente, podemos eludir este comportamiento predeterminado, vamos a guardar nuestras

memorias de traducción en el subdirectorio `tm/auto/` con lo cual OmegaT no volverá a insertar el prefijo.

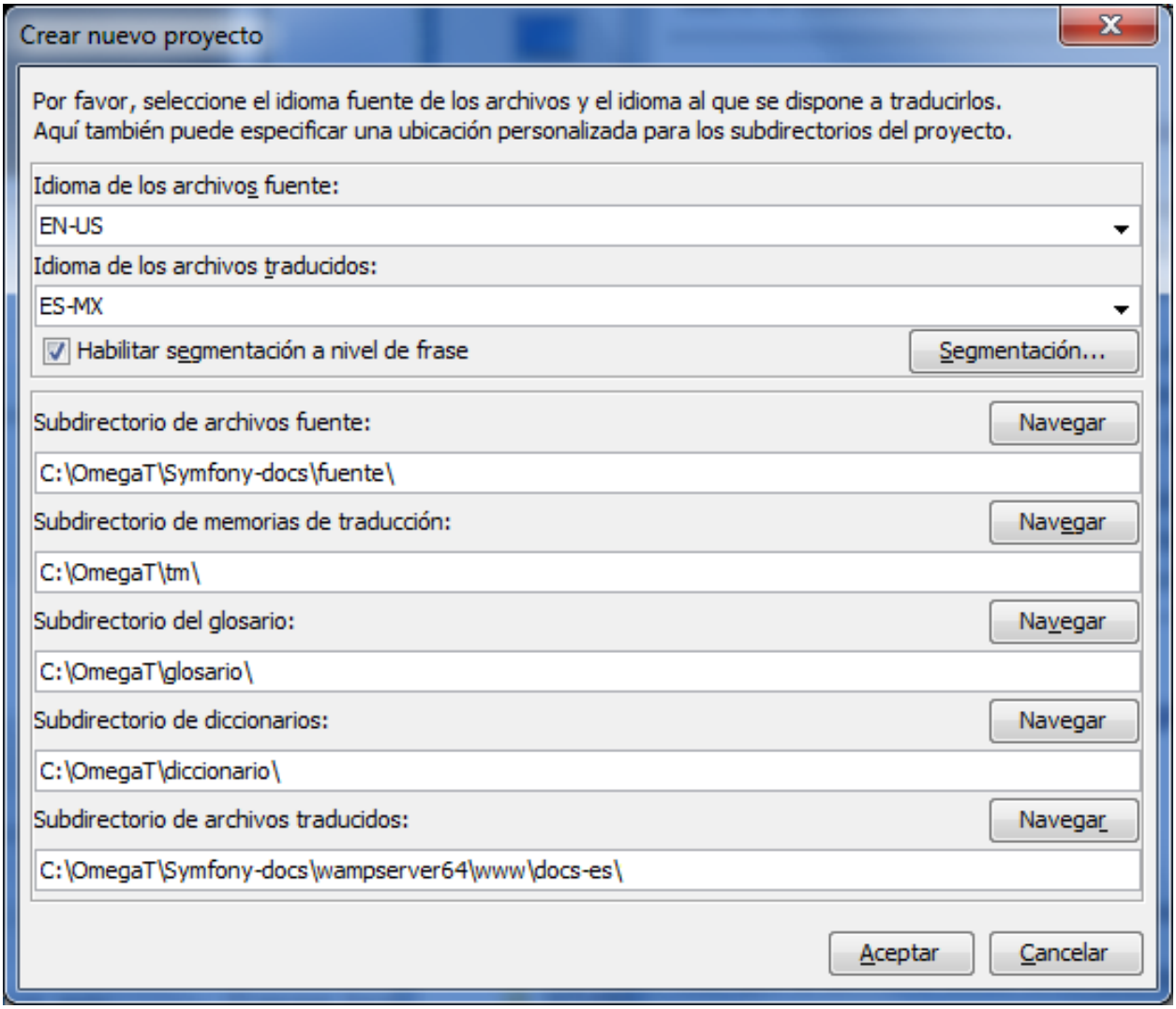
Truco: Otro modo de eliminar el prefijo sin colocar el cursor en esa posición para borrarlo manualmente, es con el acceso directo `CTRL+R` (Reemplazar con la coincidencia).

Creando el proyecto de traducción

Vamos a crear un nuevo proyecto de traducción en OmegaT, para ello ve al menú `ArchivoNuevo`, en el cuadro de diálogo navega hasta el directorio raíz de los proyectos de traducción y bautízalo como `symfony-docs` como muestra la figura:



Al pulsar en `Guardar` te mostrará un nuevo cuadro de diálogo para configurar los directorios del proyecto, ponlos de la siguiente manera:



Nota: Ten en cuenta que el *subdirectorio de archivos traducidos* lo especificamos fuera del directorio de traducción, específicamente dentro del servidor web (en este caso WampServer), en un subdirectorio en el cual también colocaremos los demás proyectos de traducción dónde generaremos la documentación en formato *HTML* para ir revisando nuestro avance.

A continuación, abrirá el nuevo proyecto y te mostrará un mensaje diciendo que aún no tienes archivos fuente para traducir, esto lo vamos a solucionar en la siguiente sección.

Consiguiendo los documentos originales

Llegó el momento de clonar los documentos originales, para ello necesitamos el sistema Git para el control de versiones, para familiarizarte con él te recomiendo el magnífico libro ProGit, es libre y está disponible en varios idiomas, para aprender cómo instalarlo puedes leer la sección Instalando Git del primer capítulo.

Clonando el repositorio `symfony-docs`

Puesto que no pretendemos modificar la documentación oficial (si quieres aportar algún artículo o guía necesitas seguir otro procedimiento), sino únicamente traducirla, necesitamos clonar el repositorio de la documentación de *Symfony* (en Inglés), en Windows necesitas lanzar `GitBash` (en otros sistemas abre una consola) y ejecuta las siguientes ordenes:

```
$ cd c:/OmegaT/symfony-docs/fuente/
$ git clone git://github.com/symfony/symfony-docs.git
```

Ahora, felizmente podrías comenzar a traducir en OmegaT. Pero... ¡Espera!, mucho del trabajo de la traducción ya se ha llevado a cabo, únicamente tenemos que descargar la memoria del proyecto de traducción y colocarla en el subdirectorio y archivo adecuado, el cual en este caso es `symfony-docs/omegat/project_save.tmx`, este archivo sí queremos compartirlo con los demás traductores, para ello vamos a crear nuestro repositorio, en este caso es el repositorio de la memoria de traducción del Proyecto de traducción al Español (necesitas buscar el de tu idioma).

Bifurcando el repositorio de traducción

Para bifurcar el repositorio de traducción debes crear una cuenta (si todavía no tienes una) en github, a continuación ve al repositorio de la memoria de traducción y haz clic en el botón `Fork`, ahora ya tienes una bifurcación con la cual puedes trabajar, sólo la tenemos que configurar para poder trabajar con ella en OmegaT. En `GitBash` (o en la misma consola, si utilizas otro sistema) ejecuta las siguientes ordenes, sustituyendo `TuNombre` y `TuCorreoE@ejemplo.com` con los valores reales:

```
$ cd c:/OmegaT/symfony-docs/omegat/
$ git config --global user.name "yourName"
$ git config --global user.correo yourEmail@example.com

$ git init
$ git clone git://github.com/gitnacho/omegat.git
```

A continuación, dentro de OmegaT, recarga el proyecto para que detecte tanto los archivos fuente recién descargados del repositorio oficial cómo la memoria, lo puedes conseguir por medio del menú `ArchivoVolver a cargar` o simplemente pulsando la tecla `F5`.

A partir de ahora puedes comenzar a traducir, corregir errores y solicitar la atracción de tus cambios al repositorio maestro; no obstante, debido a que estas herramientas son una enorme ayuda en el proceso de traducción, es buena idea llegar a un acuerdo para que cada colaborador sea el `administrador` de la memoria de traducción un determinado turno, digamos 2, 4 u 8 horas, incluso un día, algo que se pueda controlar por medio de algún sistema, por ejemplo la lista de correo de traductores o en su defecto administrar un proyecto completo, por ejemplo, *Doctrine* está subdividido, tenemos `DBAL`, `ORM`, etc.

Revisando el avance

Cuando escribes manualmente, en este caso durante la traducción, fácilmente puedes cometer errores como palabras dobles, intercambio en la posición de las letras, etc.

Mira, ¡qué interesante! Sgeun un etsduio de una uivenrsdiad ignlsea, no ipmotra el odren en el que etsan ersciats las lteras, la úicna csoa ipormtnate es que la pmrirea y la útima ltera etsén ecsritas en la psicioón cocrrtea. El rsteo peuden estar ttaolmntee mal y aun pordas lerelo sin pobrleams. Etso es pquore no lemeos cada ltera por sí msima snio la paalbra cmoo un tdoo.

¿No te parece increíble?

Este tipo de errores fácilmente los detecta el corrector ortográfico, pero, a pesar de que OmegaT cuenta con uno integrado, algunas veces olvidas comprobarlo y este tipo de error pasa desapercibido; también puedes leer físicamente el resultado pero, como acabas de ver, fácilmente puedes omitir ciertos errores tipográficos.

Una excelente ayuda en la revisión y corrección de estos errores son los sistemas de texto a voz y Balabolka es uno de ellos, es libre y está disponible en varios idiomas. Para utilizarlo sólo tienes que activar las teclas globales en la configuración del sistema para que al estar traduciendo te "repita" lo que has escrito.

Consiguiendo Balabolka

Lamentablemente, esta pieza de software sólo es para Windows, puedes descargar esta útil herramienta desde su página web y seguir las instrucciones de instalación, además, en la misma página están los enlaces para los archivos de voces en varios idiomas, los cuales se tienen que descargar e instalar por separado.

No dudo exista por ahí alguna utilidad de este tipo para Linux y OS X, si sabes de alguna te agradeceré me lo hagas saber para poner el enlace correspondiente.

Conclusión

Ahora sólo resta ponernos de acuerdo en algunos detalles, estaremos en contacto en la lista de correo de Symfony2 docs.

6.2.5 Licencia de la documentación *Symfony2*

La documentación de *Symfony2* está bajo una licencia Creative Commons Attribution-Share Alike 3.0 Unported Licencia.

Estás en libertad:

- para *Compartir* — copiar, distribuir y trasmitir públicamente la obra;
- para *Derivar* — adaptando la obra.

Bajo las siguientes condiciones:

- *Atribución* — Debes atribuir el trabajo de la manera especificada por el autor o licenciador (pero de ninguna manera que sugiera que apoya tu uso de la obra).
- *Compartir bajo la misma licencia* — Si alteras, transformas, o creas sobre esta obra, sólo podrás distribuir la obra resultante bajo una licencia idéntica o similar a esta.

En el entendido de:

- *Renuncia* — Cualquiera de estas condiciones puede no aplicarse si obtienes el permiso del titular de los derechos de autor;
- *Dominio Público* — Cuando la obra o cualquiera de sus elementos es del dominio público bajo la legislación aplicable, este estatus de ninguna manera es afectado por la licencia;
- *Otros Derechos* — De ninguna manera cualquiera de los siguientes derechos se ve afectado por la licencia:
  - Tu derecho leal o uso justo, u otros derechos de autor excepciones y limitaciones aplicables;
  - Los derechos morales del autor;
  - Otras personas pueden tener derechos ya sea en la propia obra o en la forma en que se utiliza la obra, como los derechos de publicidad o privacidad.
- *Atención* — Para cualquier reutilización o distribución, debes dejar claros los términos de la licencia de esta obra. La mejor manera de hacerlo es con un enlace a esta página web.

Este es un resumen humanamente legible del Texto legal (Licencia completa).

6.3 Aportando código

6.3.1 Reuniones IRC

El propósito de esta reunión es deliberar temas en tiempo real con muchos de los desarrolladores de *Symfony2*.

Cualquier persona puede proponer temas en la lista de correo symfony-dev hasta 24 horas antes de la reunión, idealmente incluyendo también información preparada pertinentemente a través de una *URL*. 24 horas antes de la reunión será publicado un enlace a doodle con una lista de todos los temas propuestos. Cualquier persona puede votar los temas hasta el comienzo de la reunión para definir su posición en el orden del día. Cada tema tendrá una duración fija de 15 minutos y la sesión dura una hora, dejando tiempo suficiente para por lo menos 4 temas.

Prudencia: Ten en cuenta que el objetivo de la reunión no es encontrar soluciones definitivas, sino más bien asegurarse de que existe un entendimiento común sobre el tema en cuestión y llevar adelante el debate en formas que son difíciles de conseguir con menos herramientas de comunicación en tiempo real.

Las reuniones son cada jueves a las 17:00 CET (+01:00) en el canal #symfony-dev del servidor Freenode IRC.

Los registros IRC se publicarán más tarde en el wiki de trac, el cual incluirá un breve resumen de cada uno de los temas. Puedes crear *tickets* para cualquiera de las tareas o problemas identificados durante la reunión y referidas en el resumen.

Algunas sencillas instrucciones y orientación para participar:

- Es posible cambiar los votos hasta el comienzo de la reunión, haciendo clic en "Editar entrada"
- El doodle cerrará la votación al comienzo de la reunión;
- El programa se define por los temas que tienen el mayor número de votos en doodle, o el que se propuso primero en caso de empate;
- Al comienzo de la reunión una persona se identifica a sí misma como el moderador;
- El moderador esencialmente es responsable de garantizar que se destinan 15 minutos por tema y garantizar que las tareas están claramente identificadas;
- Por lo general, el moderador se encargará de escribir el resumen y crear los *tickets* de trac a menos que alguien más los configure;
- Cualquier persona puede participar y expresamente **estás invitado** a participar;
- Lo ideal sería que uno se familiarizase con el tema propuesto antes de la reunión;
- Cuando se inicia un nuevo tema el proponente está invitado a empezar con una introducción en pocas palabras;
- Cualquier persona puede comentar conforme lo considere oportuno;
- Dependiendo de cuántas personas participen, uno potencialmente se debe retener a sí mismo de impulsar un argumento específico muy difícil;
- Recuerda que los registros IRC serán publicados más adelante, por lo tanto la gente tiene la oportunidad de revisar los comentarios más tarde, una vez más;
- Les animamos a levantar la mano para asumir las tareas definidas en la reunión.

Aquí está un ejemplo doodle.

6.3.2 Otros recursos

Con el fin de dar seguimiento a lo que está sucediendo en la comunidad pueden ser útiles estos recursos adicionales:

- Listas de peticiones de atracción abiertas
- Lista de commits recientes
- Lista abierta de errores y mejoras
- Lista de paquetes de fuente abierta
- **Código**:
  - *Errores* (Página 625) |
  - *Parches* (Página 625) |
  - *Seguridad* (Página 627) |
  - *Pruebas* (Página 628) |
  - *Estándares de codificación* (Página 629) |
  - *Convenciones de codificación* (Página 631) |
  - *Licencia* (Página 632)
- **Documentación**:
  - *Descripción* (Página 633) |
  - *Formato* (Página 634) |
  - *Traduciendo* (Página 636) |
  - *Licencia* (Página 647)
- **Comunidad**:
  - *Reuniones IRC* (Página 648) |
  - *Otros recursos* (Página 648)
- **Código**:
  - *Errores* (Página 625) |
  - *Parches* (Página 625) |
  - *Seguridad* (Página 627) |
  - *Pruebas* (Página 628) |
  - *Estándares de codificación* (Página 629) |
  - *Convenciones de codificación* (Página 631) |
  - *Licencia* (Página 632)
- **Documentación**:
  - *Descripción* (Página 633) |
  - *Formato* (Página 634) |
  - *Traduciendo* (Página 636) |
  - *Licencia* (Página 647)
- **Comunidad**:
  - *Reuniones IRC* (Página 648) |
  - *Otros recursos* (Página 648)