

Symfony

la guía definitiva

Fabien Potencier, François Zaninotto

Sobre este libro...

- Los contenidos de este libro están bajo una licencia Creative Commons Reconocimiento - No Comercial - Sin Obra Derivada 3.0 (<http://creativecommons.org/licenses/by-nc-nd/3.0/deed.es>)
- **Esta versión impresa se creó el 13 de julio de 2008 y todavía está incompleta.** La versión más actualizada de los contenidos de este libro se puede encontrar en http://www.librosweb.es/symfony_1_1
- Si quieres aportar sugerencias, comentarios, críticas o informar sobre errores, puedes enviarnos un mensaje a **contacto@librosweb.es**

Capítulo 1. Introducción a Symfony	7
1.1. Symfony en pocas palabras	7
1.2. Conceptos básicos	11
1.3. Resumen	17
Capítulo 2. Explorando el interior de Symfony	18
2.1. El patrón MVC.....	18
2.2. Organización del código	29
2.3. Herramientas comunes	34
2.4. Resumen	37
Capítulo 3. Ejecutar aplicaciones Symfony	38
3.1. Instalando el entorno de pruebas	38
3.2. Instalando las librerías de Symfony	40
3.3. Crear una aplicación web	42
3.4. Configurar el servidor web	44
3.5. Resolución de problemas	47
3.6. Versionado del código fuente.....	48
3.7. Resumen	50
Capítulo 4. Introducción a la creación de páginas	51
4.1. Crear el esqueleto del módulo	51
4.2. Enlazando a otra acción.....	56
4.3. Obteniendo información de la petición.....	58
4.4. Resumen	59
Capítulo 5. Configurar Symfony	61
5.1. El sistema de configuración	61
5.2. Un vistazo general a los archivos de configuración	66
5.3. Entornos	71
5.4. La cache de configuración	75
5.5. Accediendo a la configuración desde la aplicación	76
5.6. Trucos para los archivos de configuración.....	79
5.7. Resumen	81
Capítulo 6. El Controlador	82
6.1. El Controlador Frontal	82
6.2. Acciones.....	84
6.3. Accediendo a la petición.....	92
6.4. Sesiones de Usuario.....	94
6.5. Seguridad de la Acción.....	97
6.6. Métodos de Validación y Manejo de Errores	101
6.7. Filtros	103
6.8. Configuración del Módulo	109
6.9. Resumen	109
Capítulo 7. La Vista.....	111
7.1. Plantillas	111

7.2. Fragmentos de código	117
7.3. Configuración de la vista	124
7.4. Slots de componentes	133
7.5. Mecanismo de escape	135
7.6. Resumen	139
Capítulo 8. El modelo	140
8.1. ¿Por qué utilizar un ORM y una capa de abstracción?	140
8.2. Esquema de base de datos de Symfony	142
8.3. Las clases del modelo	144
8.4. Acceso a los datos.....	146
8.5. Conexiones con la base de datos.....	154
8.6. Extender el modelo	156
8.7. Sintaxis extendida del esquema	159
8.8. No crees el modelo dos veces	165
8.9. Resumen	167
Capítulo 9. Enlaces y sistema de enrutamiento.....	169
9.1. ¿Qué es el enrutamiento?	169
9.2. Reescritura de URL	173
9.3. Helpers de enlaces.....	175
9.4. Configuración del sistema de enrutamiento	180
9.5. Trabajando con rutas en las acciones.....	187
9.6. Resumen	188
Capítulo 10. Formularios	189
10.1. Helpers de formularios	189
10.2. Helpers de formularios para objetos	197
10.3. Validación de formularios.....	201
10.4. Validaciones complejas	213
10.5. Resumen	217
Capítulo 11. Integración con Ajax	218
11.1. Helpers básicos de JavaScript	218
11.2. Prototype.....	221
11.3. Helpers de Ajax.....	223
11.4. Parámetros para la ejecución remota	228
11.5. Creando efectos visuales	232
11.6. JSON.....	233
11.7. Interacciones complejas con Ajax.....	235
11.8. Resumen	240
Capítulo 12. Uso de la cache	241
12.1. Guardando la respuesta en la cache.....	241
12.2. Eliminando elementos de la cache	250
12.3. Probando y monitorizando la cache	254
12.4. HTTP 1.1 y la cache del lado del cliente.....	256

12.5. Resumen	259
Capítulo 13. Internacionalización y localización	260
13.1. Cultura del usuario	260
13.2. Estándares y formatos	263
13.3. Información textual en la base de datos	265
13.4. Traducción de la interfaz	267
13.5. Resumen	273
Capítulo 14. Generador de la parte de administración	274
14.1. Generación de código en función del modelo	274
14.2. Creando la parte de administración de las aplicaciones	275
14.3. Configuración del generador	279
14.4. Modificando el aspecto de los módulos generados	303
14.5. Resumen	306
Capítulo 15. Pruebas unitarias y funcionales	307
15.1. Automatización de pruebas.....	307
15.2. Pruebas unitarias	310
15.3. Pruebas funcionales.....	317
15.4. Recomendaciones sobre el nombre de las pruebas	327
15.5. Otras utilidades para pruebas	328
15.6. Resumen	332
Capítulo 16. Herramientas para la administración de aplicaciones	334
16.1. Logs.....	334
16.2. Depuración de aplicaciones.....	338
16.3. Utilizando Symfony fuera de la web	347
16.4. Cargando datos en una base de datos.....	350
16.5. Instalando aplicaciones	353
16.6. Resumen	357
Capítulo 17. Personalizar Symfony	359
17.1. Eventos	359
17.2. Factorías	367
17.3. Integrando componentes de otros frameworks.....	370
17.4. Plugins	372
17.5. Resumen	388
Capítulo 18. Rendimiento	389
18.1. Optimizando el servidor	389
18.2. Optimizando el modelo	390
18.3. Optimizando la vista	398
18.4. Optimizando la cache	400
18.5. Desactivando las características que no se utilizan	405
18.6. Optimizando el código fuente	406
18.7. Resumen	408
Capítulo 19. Configuración avanzada	409

19.1. Opciones de Symfony	409
19.2. Extendiendo la carga automática de clases.....	417
19.3. Estructura de archivos propia.....	419
19.4. Comprendiendo el funcionamiento de los manejadores de configuración	423
19.5. Resumen	426

Capítulo 1. Introducción a Symfony

¿Qué puedes hacer con Symfony? ¿Qué necesitas para utilizarlo? Este capítulo responde a todas estas preguntas.

1.1. Symfony en pocas palabras

Un framework simplifica el desarrollo de una aplicación mediante la automatización de algunos de los patrones utilizados para resolver las tareas comunes. Además, un framework proporciona estructura al código fuente, forzando al desarrollador a crear código más legible y más fácil de mantener. Por último, un framework facilita la programación de aplicaciones, ya que encapsula operaciones complejas en instrucciones sencillas.

Symfony es un completo framework diseñado para optimizar, gracias a sus características, el desarrollo de las aplicaciones web. Para empezar, separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web. Proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja. Además, automatiza las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación. El resultado de todas estas ventajas es que no se debe reinventar la rueda cada vez que se crea una nueva aplicación web.

Symfony está desarrollado completamente con PHP 5. Ha sido probado en numerosos proyectos reales y se utiliza en sitios web de comercio electrónico de primer nivel. Symfony es compatible con la mayoría de gestores de bases de datos, como MySQL, PostgreSQL, Oracle y SQL Server de Microsoft. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, etc.) como en plataformas Windows. A continuación se muestran algunas de sus características.

1.1.1. Características de Symfony

Symfony se diseñó para que se ajustara a los siguientes requisitos:

- Fácil de instalar y configurar en la mayoría de plataformas (y con la garantía de que funciona correctamente en los sistemas Windows y *nix estándares)
- Independiente del sistema gestor de bases de datos
- Sencillo de usar en la mayoría de casos, pero lo suficientemente flexible como para adaptarse a los casos más complejos
- Basado en la premisa de "*convenir en vez de configurar*", en la que el desarrollador solo debe configurar aquello que no es convencional
- Sigue la mayoría de *mejores prácticas* y patrones de diseño para la web
- Preparado para aplicaciones empresariales y adaptable a las políticas y arquitecturas propias de cada empresa, además de ser lo suficientemente estable como para desarrollar aplicaciones a largo plazo

- Código fácil de leer que incluye comentarios de phpDocumentor y que permite un mantenimiento muy sencillo
- Fácil de extender, lo que permite su integración con librerías desarrolladas por terceros

1.1.1.1. Automatización de características de proyectos web

Symfony automatiza la mayoría de elementos comunes de los proyectos web, como por ejemplo:

- La capa de internacionalización que incluye Symfony permite la traducción de los datos y de la interfaz, así como la adaptación local de los contenidos.
- La capa de presentación utiliza plantillas y *layouts* que pueden ser creados por diseñadores HTML sin ningún tipo de conocimiento del framework. Los *helpers* incluidos permiten minimizar el código utilizado en la presentación, ya que encapsulan grandes bloques de código en llamadas simples a funciones.
- Los formularios incluyen validación automatizada y relleno automático de datos ("*repopulation*"), lo que asegura la obtención de datos correctos y mejora la experiencia de usuario.
- Los datos incluyen mecanismos de escape que permiten una mejor protección contra los ataques producidos por datos corruptos.
- La gestión de la caché reduce el ancho de banda utilizado y la carga del servidor.
- La autenticación y la gestión de credenciales simplifican la creación de secciones restringidas y la gestión de la seguridad de usuario.
- El sistema de enrutamiento y las URL *limpias* permiten considerar a las direcciones de las páginas como parte de la interfaz, además de estar optimizadas para los buscadores.
- El soporte de e-mail incluido y la gestión de APIs permiten a las aplicaciones web interactuar más allá de los navegadores.
- Los listados son más fáciles de utilizar debido a la paginación automatizada, el filtrado y la ordenación de datos.
- Los plugins, las factorías (patrón de diseño "*Factory*") y los "*mixin*" permiten realizar extensiones a medida de Symfony.
- Las interacciones con Ajax son muy fáciles de implementar mediante los *helpers* que permiten encapsular los efectos JavaScript compatibles con todos los navegadores en una única línea de código.

1.1.1.2. Entorno de desarrollo y herramientas

Symfony puede ser completamente personalizado para cumplir con los requisitos de las empresas que disponen de sus propias políticas y reglas para la gestión de proyectos y la programación de aplicaciones. Por defecto incorpora varios entornos de desarrollo diferentes e incluye varias herramientas que permiten automatizar las tareas más comunes de la ingeniería del software:

- Las herramientas que generan automáticamente código han sido diseñadas para hacer prototipos de aplicaciones y para crear fácilmente la parte de gestión de las aplicaciones.
- El framework de desarrollo de pruebas unitarias y funcionales proporciona las herramientas ideales para el desarrollo basado en pruebas (*"test-driven development"*).
- La barra de depuración web simplifica la depuración de las aplicaciones, ya que muestra toda la información que los programadores necesitan sobre la página en la que están trabajando.
- La interfaz de línea de comandos automatiza la instalación de las aplicaciones entre servidores.
- Es posible realizar cambios *"en caliente"* de la configuración (sin necesidad de reiniciar el servidor).
- El completo sistema de log permite a los administradores acceder hasta el último detalle de las actividades que realiza la aplicación.

1.1.2. ¿Quién ha desarrollado Symfony y por qué motivo?

La primera versión de Symfony fue publicada en Octubre de 2005 por Fabien Potencier, fundador del proyecto y coautor de este libro. Fabien es el presidente de Sensio (<http://www.sensio.com/>), una empresa francesa de desarrollo de aplicaciones web conocida por sus innovaciones en este campo.

En el año 2003, Fabien realizó una investigación sobre las herramientas de software libre disponibles para el desarrollo de aplicaciones web con PHP. Fabien llegó a la conclusión de que no existía ninguna herramienta con esas características. Después del lanzamiento de la versión 5 de PHP, decidió que las herramientas disponibles habían alcanzado un grado de madurez suficiente como para integrarlas en un framework completo. Fabien empleó un año entero para desarrollar el núcleo de Symfony, basando su trabajo en el framework Mojavi (que también era un framework que seguía el funcionamiento MVC), en la herramienta Propel para el mapeo de objetos a bases de datos (conocido como ORM, de *"object-relational mapping"*) y en los *helpers* empleados por Ruby on Rails en sus plantillas.

Fabien desarrolló originalmente Symfony para utilizarlo en los proyectos de Sensio, ya que disponer de un framework efectivo es la mejor ayuda para el desarrollo eficiente y rápido de las aplicaciones. Además, el desarrollo web se hace más intuitivo y las aplicaciones resultantes son más robustas y más fáciles de mantener. El framework se utilizó por primera vez en el desarrollo de un sitio de comercio electrónico para un vendedor de lencería y posteriormente se utilizó en otros proyectos.

Después de utilizar Symfony en algunos proyectos, Fabien decidió publicarlo bajo una licencia de software libre. Sus razones para liberar el proyecto fueron para donar su trabajo a la comunidad, aprovechar la respuesta de los usuarios, mostrar la experiencia de Sensio y porque considera que es divertido hacerlo.

Nota ¿Por qué lo llamaron *"Symfony"* y no *"CualquierNombreFramework"*? Porque Fabien quería una nombre corto que tuviera una letra 's' (de Sensio) y una letra 'f' (de framework), que fuera

fácil de recordar y que no estuviera asociado a otra herramienta de desarrollo. Además, no le gustan las mayúsculas. "Symfony" era muy parecido a lo que estaba buscando, aunque no es una palabra correcta en el idioma inglés (la palabra correcta es "*symphony*"), y además estaba libre como nombre de proyecto. La otra alternativa era "*baguette*".

Para que Symfony fuera un proyecto de software libre exitoso, debía tener una documentación amplia y en inglés, para aumentar la incorporación de usuarios al proyecto. Fabien pidió a su compañero de trabajo François Zaninotto, el otro coautor de este libro, que investigara el código fuente del programa y escribiera un libro sobre Symfony. Aunque el proceso fue arduo, cuando el proyecto se lanzó públicamente, la documentación era suficiente como para atraer a muchos desarrolladores. El resto es historia.

1.1.3. La comunidad Symfony

En cuanto se abrió al público el sitio web de Symfony (<http://www.symfony-project.org/>) muchos desarrolladores de todo el mundo se descargaron e instalaron el framework, comenzaron a leer la documentación y construyeron sus primeras aplicaciones con Symfony, aumentando poco a poco la popularidad de Symfony.

En ese momento, los frameworks para el desarrollo de aplicaciones web estaban en pleno apogeo, y era muy necesario disponer de un completo framework realizado con PHP. Symfony proporcionaba una solución irresistible a esa carencia, debido a la calidad de su código fuente y a la gran cantidad de documentación disponible, dos ventajas muy importantes sobre otros frameworks disponibles. Los colaboradores aparecieron en seguida proponiendo parches y mejoras, detectando los errores de la documentación y realizando otras tareas muy importantes.

El repositorio público de código fuente y el sistema de notificación de errores y mejoras mediante tickets permite varias formas de contribuir al proyecto y todos los voluntarios son bienvenidos. Fabien continua siendo el mayor contribuidor de código al repositorio y se encarga de garantizar la calidad del código.

Actualmente, el foro de Symfony, las listas de correo y el IRC ofrecen otras alternativas válidas para el soporte del framework, con el que cada pregunta suele obtener una media de 4 respuestas. Cada día nuevos usuarios instalan Symfony y el wiki y la sección de fragmentos de código almacenan una gran cantidad de documentación generada por los usuarios. Cada semana el número de aplicaciones conocidas desarrolladas con Symfony se incrementa en 5 y el aumento continua.

La comunidad Symfony es el tercer pilar del framework y esperamos que tu también te unas a ella después de leer este libro.

1.1.4. ¿Es adecuado Symfony para mí?

Independientemente de que seas un experto programador de PHP 5 o un principiante en el desarrollo de aplicaciones web, podrás utilizar Symfony de forma sencilla. El principal argumento para decidir si deberías o no utilizar Symfony es el tamaño del proyecto.

Si tu proyecto consiste en desarrollar un sitio web sencillo con 5 o 10 páginas diferentes, acceso simple a bases de datos y no es importante asegurar un gran rendimiento o una documentación

adecuada, deberías realizar tu proyecto solo con PHP. En ese caso, no vas a obtener grandes ventajas por utilizar un framework de desarrollo de aplicaciones web, además de que utilizar objetos y el modelo MVC (*Modelo Vista Controlador*) solamente va a ralentizar el desarrollo de tu proyecto. Además, Symfony no está optimizado para ejecutarse de forma eficiente en un servidor compartido en el que los scripts de PHP se ejecutan solamente mediante CGI (*Common Gateway Interface*).

Por otra parte, si desarrollas aplicaciones web complejas con mucha lógica de negocio, no es recomendable utilizar solo PHP. Para asegurar el mantenimiento y las ampliaciones futuras de la aplicación, es necesario que el código sea ligero, legible y efectivo. Si quieres incorporar los últimos avances en interacción con usuarios (como por ejemplo Ajax), puedes acabar escribiendo cientos de líneas de JavaScript. Si quieres desarrollar aplicaciones de forma divertida y muy rápida, no es aconsejable utilizar solo PHP. En todos estos casos, deberías utilizar Symfony.

Si eres un desarrollador web profesional, ya conoces todas las ventajas de utilizar un framework de desarrollo de aplicaciones web y solo necesitas un framework que sea maduro, bien documentado y con una gran comunidad que lo apoye. En este caso, deberías dejar de buscar porque Symfony es lo que necesitas.

Sugerencia Si quieres ver una demostración visual de las posibilidades de Symfony, deberías ver los vídeos o *screencasts* que están disponibles en el sitio web de Symfony. En estas demostraciones se ve lo rápido y divertido que es desarrollar aplicaciones web con Symfony.

1.2. Conceptos básicos

Antes de empezar con Symfony, deberías conocer algunos conceptos básicos. Puedes saltarte esta sección si conoces el significado de OOP, ORM, RAD, DRY, KISS, TDD, YAML y PEAR.

1.2.1. PHP 5

Symfony está programado en PHP 5 (<http://www.php.net/>) y está enfocado al desarrollo de aplicaciones web en el mismo lenguaje de programación. Por este motivo, es obligatorio disponer de unos conocimientos avanzados de PHP 5 para sacar el máximo partido al framework. La versión mínima de PHP requerida para ejecutar Symfony es PHP 5.1.

Los programadores que conocen PHP 4 pero que no han trabajado con PHP 5 deberían centrarse en el nuevo modelo orientado a objetos de PHP.

1.2.2. Programación Orientada a Objetos (OOP)

La programación orientada a objetos (OOP, por sus siglas en inglés *Object-oriented programming*) no va a ser explicada en este capítulo, ya que se necesitaría un libro entero para ello. Como Symfony hace un uso continuo de los mecanismos orientados a objetos disponibles en PHP 5, es un requisito obligatorio el conocer la OOP antes de aprender Symfony.

En la Wikipedia se explica la OOP de la siguiente manera:

“ La idea de la programación orientada a objetos es que una aplicación se puede considerar como una colección de unidades individuales, llamadas objetos, que interactúan entre sí. Los programas tradicionales pueden considerarse como una colección de funciones o como una lista de instrucciones de programación.”

PHP 5 incluye los conceptos de clase, objeto, método, herencia y muchos otros propios de la programación orientada a objetos. Aquellos que no estén familiarizados con estos conceptos, deberían consultar la documentación oficial de PHP disponible en <http://www.php.net/manual/es/language.oop5.basic.php>.

1.2.3. Métodos mágicos

Uno de los puntos fuertes de los objetos de PHP es la utilización de los *"métodos mágicos"*. Este tipo de métodos permiten redefinir el comportamiento de las clases sin modificar el código externo. Con estos métodos es posible que la sintaxis de PHP sea más concisa y más fácil de extender. Además, estos métodos son fáciles de reconocer ya que sus nombres siempre empiezan con 2 guiones bajos seguidos (___).

Por ejemplo, al mostrar un objeto, PHP busca de forma implícita un método llamado `__toString()` en ese objeto y que permite comprobar si se ha creado una visualización personalizada para ese objeto:

```
$miObjeto = new miClase();
echo $miObjeto;

// Se busca el método mágico
echo $miObjeto->__toString();
```

Symfony utiliza los métodos mágicos de PHP, por lo que deberías conocer su funcionamiento. La documentación oficial de PHP también explica los métodos mágicos (<http://www.php.net/manual/es/language.oop5.magic.php>)

1.2.4. PEAR (PHP Extension and Application Repository)

PEAR es un *"framework y sistema de distribución para componentes PHP reutilizables"*. PEAR permite descargar, instalar, actualizar y desinstalar scripts de PHP. Si se utiliza un paquete de PEAR, no es necesario decidir donde guardar los scripts, cómo hacer que se puedan utilizar o cómo extender la línea de comandos (CLI).

PEAR es un proyecto creado por la comunidad de usuarios de PHP, está desarrollado con PHP y se incluye en las distribuciones estándar de PHP.

Sugerencia El sitio web de PEAR, <http://pear.php.net/>, incluye documentación y muchos paquetes agrupados en categorías.

PEAR es el método más profesional para instalar librerías externas en PHP. Symfony aconseja el uso de PEAR para disponer de una instalación única y centralizada que pueda ser utilizada en varios proyectos. Los plugins de Symfony son paquetes de PEAR con una configuración especial. El propio framework Symfony también está disponible como paquete de PEAR.

Afortunadamente, no es necesario conocer la sintaxis de PEAR para utilizar Symfony. Lo único necesario es entender su funcionamiento y tenerlo instalado. Para comprobar si PEAR está instalado en el sistema, se puede escribir lo siguiente en una línea de comandos:

```
| > pear info pear
```

El comando anterior muestra la versión de PEAR instalada en el sistema.

El proyecto Symfony dispone de su propio repositorio PEAR, también llamado canal. Los canales de PEAR solamente se pueden utilizar a partir de la versión 1.4.0, por lo que es necesario actualizar PEAR si se dispone de una versión anterior. Para actualizar PEAR, se debe ejecutar el siguiente comando:

```
| > pear upgrade PEAR
```

1.2.5. Mapeo de Objetos a Bases de datos (ORM)

Las bases de datos siguen una estructura relacional. PHP 5 y Symfony por el contrario son orientados a objetos. Por este motivo, para acceder a la base de datos como si fuera orientada a objetos, es necesario una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se denomina "*mapeo de objetos a bases de datos*" (ORM, de sus siglas en inglés "*object-relational mapping*").

Un ORM consiste en una serie de objetos que permiten acceder a los datos y que contienen en su interior cierta lógica de negocio.

Una de las ventajas de utilizar estas capas de abstracción de objetos/relacional es que evita utilizar una sintaxis específica de un sistema de bases de datos concreto. Esta capa transforma automáticamente las llamadas a los objetos en consultas SQL optimizadas para el sistema gestor de bases de datos que se está utilizando en cada momento.

De esta forma, es muy sencillo cambiar a otro sistema de bases de datos completamente diferente en mitad del desarrollo de un proyecto. Estas técnicas son útiles por ejemplo cuando se debe desarrollar un prototipo rápido de una aplicación y el cliente aun no ha decidido el sistema de bases de datos que más le conviene. El prototipo se puede realizar utilizando SQLite y después se puede cambiar fácilmente a MySQL, PostgreSQL u Oracle cuando el cliente se haya decidido. El cambio se puede realizar modificando solamente una línea en un archivo de configuración.

La capa de abstracción utilizada encapsula toda la lógica de los datos. El resto de la aplicación no tiene que preocuparse por las consultas SQL y el código SQL que se encarga del acceso a la base de datos es fácil de encontrar. Los desarrolladores especializados en la programación con bases de datos pueden localizar fácilmente el código.

Utilizar objetos en vez de registros y clases en vez de tablas tiene otra ventaja: se pueden definir nuevos métodos de acceso a las tablas. Por ejemplo, si se dispone de una tabla llamada `Cliente` con 2 campos, `Nombre` y `Apellido`, puede que sea necesario acceder directamente al nombre completo (`NombreCompleto`). Con la programación orientada a objetos, este problema se resuelve añadiendo un nuevo método de acceso a la clase `Cliente` de la siguiente forma:

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

Todas las funciones comunes de acceso a los datos y toda la lógica de negocio relacionada con los datos se puede mantener dentro de ese tipo de objetos. Por ejemplo, la siguiente clase `CarritoCompra` almacena los productos (que son objetos). Para obtener el precio total de los productos del carrito y así realizar el pago, se puede añadir un método llamado `getTotal()` de la siguiente forma:

```
public function getTotal()
{
    $total = 0;
    foreach ($this->getProductos() as $producto)
    {
        $total += $producto->getPrecio() * $item->getCantidad();
    }
    return $total;
}
```

Y eso es todo. Imagina cuanto te hubiera costado escribir una consulta SQL que hiciera lo mismo.

Propel, que también es un proyecto de software libre, es una de las mejores capas de abstracción de objetos/relacional disponibles en PHP 5. Propel está completamente integrado en Symfony e incluso es su ORM por defecto, por lo que la mayoría de las manipulaciones de datos realizadas en este libro siguen la sintaxis de Propel. En el libro se describe la utilización de los objetos de Propel, pero se puede encontrar una referencia más completa en el sitio web de Propel (<http://propel.phpdb.org/trac/>).

Nota A partir de Symfony 1.1, Propel se incluye en el *framework* en forma de plugin, lo que facilita el cambio a otro ORM. De hecho, si quieres utilizar Doctrine (<http://www.phpdoctrine.org/>) como ORM en tus proyectos, sólo tienes que instalar el plugin `sfDoctrinePlugin`.

1.2.6. Desarrollo rápido de aplicaciones (RAD)

Durante mucho tiempo, la programación de aplicaciones web fue un tarea tediosa y muy lenta. Siguiendo los ciclos habituales de la ingeniería del software (como los propuestos por el *Proceso Racional Unificado* o *Rational Unified Process*) el desarrollo de una aplicación web no puede comenzar hasta que se han establecido por escrito una serie de requisitos, se han creado los diagramas UML (*Unified Modeling Language*) y se ha producido abundante documentación sobre el proyecto. Este modelo se veía favorecido por la baja velocidad de desarrollo, la falta de versatilidad de los lenguajes de programación (antes de ejecutar el programa se debe construir, compilar y reiniciar) y sobre todo por el hecho de que los clientes no estaban dispuestos a adaptarse a otras metodologías.

Hoy en día, las empresas reaccionan más rápidamente y los clientes cambian de opinión constantemente durante el desarrollo de los proyectos. De este modo, los equipos de desarrollo deben adaptarse a esas necesidades y tienen que poder cambiar la estructura de una aplicación

de forma rápida. Afortunadamente, el uso de lenguajes de script como Perl y PHP permiten seguir otras estrategias de programación, como RAD (desarrollo rápido de aplicaciones) y el desarrollo ágil de software.

Una de las ideas centrales de esta metodología es que el desarrollo empiece lo antes posible para que el cliente pueda revisar un prototipo que funciona y pueda indicar el camino a seguir. A partir de ahí, la aplicación se desarrolla de forma iterativa, en la que cada nueva versión incorpora nuevas funcionalidades y se desarrolla en un breve espacio de tiempo.

Las consecuencias de estas metodologías para el desarrollador son numerosas. El programador no debe pensar acerca de las versiones futuras al incluir una nueva funcionalidad. Los métodos utilizados deben ser lo más sencillos y directos posibles. Estas ideas se resumen en el principio denominado KISS: ¡Haz las cosas sencillas, idiota! (*Keep It Simple, Stupid*)

Cuando se modifican los requisitos o cuando se añade una nueva funcionalidad, normalmente se debe reescribir parte del código existente. Este proceso se llama refactorización y sucede a menudo durante el desarrollo de una aplicación web. El código suele moverse a otros lugares en función de su naturaleza. Los bloques de código repetidos se refactorizan en un único lugar, aplicando el principio DRY: No te repitas (*Don't Repeat Yourself*).

Para asegurar que la aplicación sigue funcionando correctamente a pesar de los cambios constantes, se necesita una serie de pruebas unitarias que puedan ser automatizadas. Si están bien escritas, las pruebas unitarias permiten asegurar que nada ha dejado de funcionar después de haber refactorizado parte del código de la aplicación. Algunas metodologías de desarrollo de aplicaciones obligan a escribir las pruebas antes que el propio código, lo que se conoce como TDD: desarrollo basado en pruebas (*test-driven development*).

Nota Existen otros principios y hábitos relacionados con el desarrollo ágil de aplicaciones. Una de las metodologías más efectivas se conoce como XP: programación extrema (*Extreme Programming*). La documentación relacionada con XP puede enseñarte mucho sobre el desarrollo rápido y efectivo de las aplicaciones. Una buena forma de empezar con XP son los libros escritos por Kent Beck en la editorial Addison-Wesley.

Symfony es la herramienta ideal para el RAD. De hecho, el framework ha sido desarrollado por una empresa que aplica el RAD a sus propios proyectos. Por este motivo, aprender a utilizar Symfony no es como aprender un nuevo lenguaje de programación, sino que consiste en aprender a tomar las decisiones correctas para desarrollar las aplicaciones de forma más efectiva.

1.2.7. YAML

Según el sitio web oficial de YAML (<http://www.yaml.org/>), YAML es *"un formato para serializar datos que es fácil de procesar por las máquinas, fácil de leer para las personas y fácil de interactuar con los lenguajes de script"*. Dicho de otra forma, YAML es un lenguaje muy sencillo que permite describir los datos como en XML, pero con una sintaxis mucho más sencilla. YAML es un formato especialmente útil para describir datos que pueden ser transformados en arrays simples y asociativos, como por ejemplo:

```
$casa = array(
    'familia' => array(
```

```
'apellido' => 'García',
'padres' => array('Antonio', 'María'),
'hijos' => array('Jose', 'Manuel', 'Carmen')
),
'direccion' => array(
    'numero' => 34,
    'calle' => 'Gran Vía',
    'ciudad' => 'Cualquiera',
    'codigopostal' => '12345'
)
);
```

Este array de PHP se puede crear directamente procesando esta cadena de texto en formato YAML:

```
casa:
  familia:
    apellido: García
    padres:
      - Antonio
      - María
    hijos:
      - Jose
      - Manuel
      - Carmen
  direccion:
    numero: 34
    calle: Gran Vía
    ciudad: Cualquiera
    codigopostal: "12345"
```

YAML utiliza la tabulación para indicar su estructura, los elementos que forman una secuencia utilizan un guión medio y los pares clave/valor de los array asociativos se separan con dos puntos. YAML también dispone de una notación resumida para describir la misma estructura con menos líneas: los arrays simples se definen con `[]` y los arrays asociativos se definen con `{}`. Por tanto, los datos YAML anteriores se pueden escribir de forma abreviada de la siguiente manera:

```
casa:
  familia: { apellido: García, padres: [Antonio, María], hijos: [Jose, Manuel, Carmen] }
  direccion: { numero: 34, direccion: Gran Vía, ciudad: Cualquiera, codigopostal:
    "12345" }
```

YAML es el acrónimo de *"YAML Ain't Markup Language"* ("YAML No es un Lenguaje de Marcado") y se pronuncia *"yamel"*. El formato se lleva utilizando desde 2001 y existen utilidades para procesar YAML en una gran variedad de lenguajes de programación.

Sugerencia La especificación completa del formato YAML se puede encontrar en <http://www.yaml.org/>.

Como se ha visto, YAML es mucho más rápido de escribir que XML (ya que no hacen falta las etiquetas de cierre y el uso continuo de las comillas) y es mucho más poderoso que los tradicionales archivos `.ini` (ya que estos últimos no soportan la herencia y las estructuras complejas). Por este motivo, Symfony utiliza el formato YAML como el lenguaje preferido para

almacenar su configuración. Este libro contiene muchos archivos YAML, pero como es tan sencillo, probablemente no necesites aprender más detalles de este formato.

1.3. Resumen

Symfony es un framework para desarrollar aplicaciones web creado con PHP 5. Añade una nueva capa por encima de PHP y proporciona herramientas que simplifican el desarrollo de las aplicaciones web complejas. Este libro contiene todos los detalles del funcionamiento de Symfony y para entenderlo, solamente es necesario estar familiarizado con los conceptos básicos de la programación moderna, sobre todo la programación orientada a objetos (OOP), el mapeo de objetos a bases de datos (ORM) y el desarrollo rápido de aplicaciones (RAD). El único requisito técnico obligatorio es el conocimiento de PHP 5.

Capítulo 2. Explorando el interior de Symfony

La primera vez que se accede al código fuente de una aplicación realizada con Symfony, puede desanimar un poco a los nuevos desarrolladores. El código está dividido en muchos directorios y muchos scripts y los archivos son un conjunto de clases PHP, código HTML e incluso una mezcla de los dos. Además, existen referencias a clases que no se pueden encontrar dentro del directorio del proyecto y la anidación de directorios puede llegar hasta los seis niveles. Sin embargo, cuando comprendas las razones que están detrás de esta aparente complejidad, lo verás como algo completamente natural y no querrás cambiar la estructura de una aplicación Symfony por ninguna otra. En este capítulo se explica con detalle toda esa estructura.

2.1. El patrón MVC

Symfony está basado en un patrón clásico del diseño web conocido como arquitectura MVC, que está formado por tres niveles:

- El Modelo representa la información con la que trabaja la aplicación, es decir, su lógica de negocio.
- La Vista transforma el modelo en una página web que permite al usuario interactuar con ella.
- El Controlador se encarga de procesar las interacciones del usuario y realiza los cambios apropiados en el modelo o en la vista.

La Figura 2-1 ilustra el funcionamiento del patrón MVC.

La arquitectura MVC separa la lógica de negocio (el modelo) y la presentación (la vista) por lo que se consigue un mantenimiento más sencillo de las aplicaciones. Si por ejemplo una misma aplicación debe ejecutarse tanto en un navegador estándar como en un navegador de un dispositivo móvil, solamente es necesario crear una vista nueva para cada dispositivo; manteniendo el controlador y el modelo original. El controlador se encarga de aislar al modelo y a la vista de los detalles del protocolo utilizado para las peticiones (HTTP, consola de comandos, email, etc.). El modelo se encarga de la abstracción de la lógica relacionada con los datos, haciendo que la vista y las acciones sean independientes de, por ejemplo, el tipo de gestor de bases de datos utilizado por la aplicación.

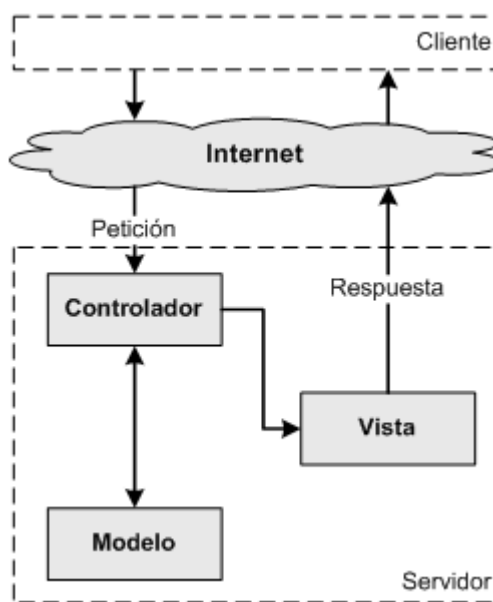


Figura 2.1. El patrón MVC

2.1.1. Las capas de la arquitectura MVC

Para poder entender las ventajas de utilizar el patrón MVC, se va a transformar una aplicación simple realizada con PHP en una aplicación que sigue la arquitectura MVC. Un buen ejemplo para ilustrar esta explicación es el de mostrar una lista con las últimas entradas o artículos de un blog.

2.1.1.1. Programación simple

Utilizando solamente PHP normal y corriente, el script necesario para mostrar los artículos almacenados en una base de datos se muestra en el siguiente listado:

Listado 2-1 - Un script simple

```
<?php

// Conectar con la base de datos y seleccionarla
$conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
mysql_select_db('blog_db', $conexion);

// Ejecutar la consulta SQL
$resultado = mysql_query('SELECT fecha, titulo FROM articulo', $conexion);

?>

<html>
  <head>
    <title>Listado de Artículos</title>
  </head>
  <body>
    <h1>Listado de Artículos</h1>
    <table>
      <tr><th>Fecha</th><th>Titulo</th></tr>
    </table>
  </body>
</html>
<?php
```

```
// Mostrar los resultados con HTML
while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC))
{
    echo "\t<tr>\n";
    printf("\t\t<td> %s </td>\n", $fila['fecha']);
    printf("\t\t<td> %s </td>\n", $fila['titulo']);
    echo "\t</tr>\n";
}
?>

    </table>
    </body>
</html>

<?php

// Cerrar la conexion
mysql_close($conexion);

?>
```

El script anterior es fácil de escribir y rápido de ejecutar, pero muy difícil de mantener y actualizar. Los principales problemas del código anterior son:

- No existe protección frente a errores (¿qué ocurre si falla la conexión con la base de datos?).
- El código HTML y el código PHP están mezclados en el mismo archivo e incluso en algunas partes están entrelazados.
- El código solo funciona si la base de datos es MySQL.

2.1.1.2. Separando la presentación

Las llamadas a `echo` y `printf` del listado 2-1 dificultan la lectura del código. De hecho, modificar el código HTML del script anterior para mejorar la presentación es un follón debido a cómo está programado. Así que el código va a ser dividido en dos partes. En primer lugar, el código PHP puro con toda la lógica de negocio se incluye en el script del controlador, como se muestra en el listado 2-2.

Listado 2-2 - La parte del controlador, en `index.php`

```
<?php

// Conectar con la base de datos y seleccionarla
$conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
mysql_select_db('blog_db', $conexion);

// Ejecutar la consulta SQL
$resultado = mysql_query('SELECT fecha, titulo FROM articulo', $conexion);

// Crear el array de elementos para la capa de la vista
$articulos = array();
while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC))
{
    $articulos[] = $fila;
```

```
}

// Cerrar la conexión
mysql_close($conexion);

// Incluir la lógica de la vista
require('vista.php');
```

El código HTML, que contiene cierto código PHP a modo de plantilla, se almacena en el script de la vista, como se muestra en el listado 2-3.

Listado 2-3 - La parte de la vista, en vista.php

```
<html>
  <head>
    <title>Listado de Artículos</title>
  </head>
  <body>
    <h1>Listado de Artículos</h1>
    <table>
      <tr><th>Fecha</th><th>Título</th></tr>
      <?php foreach ($articulos as $articulo): ?>
        <tr>
          <td><?php echo $articulo['fecha'] ?></td>
          <td><?php echo $articulo['titulo'] ?></td>
        </tr>
      <?php endforeach; ?>
    </table>
  </body>
</html>
```

Una buena regla general para determinar si la parte de la vista está suficientemente *limpia* de código es que debería contener una cantidad mínima de código PHP, la suficiente como para que un diseñador HTML sin conocimientos de PHP pueda entenderla. Las instrucciones más comunes en la parte de la vista suelen ser echo, if/endif, foreach/endforeach y poco más. Además, no se deben incluir instrucciones PHP que generen etiquetas HTML.

Toda la lógica se ha centralizado en el script del controlador, que solamente contiene código PHP y ningún tipo de HTML. De hecho, y como puedes imaginar, el mismo controlador se puede reutilizar para otros tipos de presentaciones completamente diferentes, como por ejemplo un archivo PDF o una estructura de tipo XML.

2.1.1.3. Separando la manipulación de los datos

La mayor parte del script del controlador se encarga de la manipulación de los datos. Pero, ¿qué ocurre si se necesita la lista de entradas del blog para otro controlador, por ejemplo uno que se dedica a generar el canal RSS de las entradas del blog? ¿Y si se quieren centralizar todas las consultas a la base de datos en un único sitio para evitar duplicidades? ¿Qué ocurre si cambia el modelo de datos y la tabla `articulo` pasa a llamarse `articulo_blog`? ¿Y si se quiere cambiar a PostgreSQL en vez de MySQL? Para poder hacer todo esto, es imprescindible eliminar del controlador todo el código que se encarga de la manipulación de los datos y ponerlo en otro script, llamado *el modelo*, tal y como se muestra en el listado 2-4.

Listado 2-4 - La parte del modelo, en modelo.php

```
<?php

function getTodosLosArticulos()
{
    // Conectar con la base de datos y seleccionarla
    $conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
    mysql_select_db('blog_db', $conexion);

    // Ejecutar la consulta SQL
    $resultado = mysql_query('SELECT fecha, titulo FROM articulo', $conexion);

    // Crear el array de elementos para la capa de la vista
    $articulos = array();
    while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC))
    {
        $articulos[] = $fila;
    }

    // Cerrar la conexión
    mysql_close($conexion);

    return $articulos;
}
```

El controlador modificado se puede ver en el listado 2-5.

Listado 2-5 - La parte del controlador, modificada, en index.php

```
<?php

// Incluir la lógica del modelo
require_once('modelo.php');

// Obtener la lista de artículos
$articulos = getTodosLosArticulos();

// Incluir la lógica de la vista
require('vista.php');
```

Ahora el controlador es mucho más fácil de leer. Su única tarea es la de obtener los datos del modelo y pasárselos a la vista. En las aplicaciones más complejas, el controlador se encarga además de procesar las peticiones, las sesiones de los usuarios, la autenticación, etc. El uso de nombres apropiados para las funciones del modelo hacen que sea innecesario añadir comentarios al código del controlador.

El script del modelo solamente se encarga del acceso a los datos y puede ser reorganizado a tal efecto. Todos los parámetros que no dependen de la capa de datos (como por ejemplo los parámetros de la petición del usuario) se deben obtener a través del controlador y por tanto, no se puede acceder a ellos directamente desde el modelo. Las funciones del modelo se pueden reutilizar fácilmente en otros controladores.

2.1.2. Separación en capas más allá del MVC

El principio más importante de la arquitectura MVC es la separación del código del programa en tres capas, dependiendo de su naturaleza. La lógica relacionada con los datos se incluye en el modelo, el código de la presentación en la vista y la lógica de la aplicación en el controlador.

La programación se puede simplificar si se utilizan otros patrones de diseño. De esta forma, las capas del modelo, la vista y el controlador se pueden subdividir en más capas.

2.1.2.1. Abstracción de la base de datos

La capa del modelo se puede dividir en la capa de acceso a los datos y en la capa de abstracción de la base de datos. De esta forma, las funciones que acceden a los datos no utilizan sentencias ni consultas que dependen de una base de datos, sino que utilizan otras funciones para realizar las consultas. Así, si se cambia de sistema gestor de bases de datos, solamente es necesario actualizar la capa de abstracción de la base de datos.

El listado 2-6 muestra un ejemplo de capa de abstracción de la base de datos y el listado 2-7 muestra una capa de acceso a datos específica para MySQL.

Listado 2-6 - La parte del modelo correspondiente a la abstracción de la base de datos

```
<?php

function crear_conexion($servidor, $usuario, $contrasena)
{
    return mysql_connect($servidor, $usuario, $contrasena);
}

function cerrar_conexion($conexion)
{
    mysql_close($conexion);
}

function consulta_base_de_datos($consulta, $base_datos, $conexion)
{
    mysql_select_db($base_datos, $conexion);

    return mysql_query($consulta, $conexion);
}

function obtener_resultados($resultado)
{
    return mysql_fetch_array($resultado, MYSQL_ASSOC);
}
```

Listado 2-7 - La parte del modelo correspondiente al acceso a los datos

```
function getTodosLosArticulos()
{
    // Conectar con la base de datos
    $conexion = crear_conexion('localhost', 'miusuario', 'micontrasena');

    // Ejecutar la consulta SQL
```

```

    $resultado = consulta_base_de_datos('SELECT fecha, titulo FROM articulo', 'blog_db',
    $conexion);

    // Crear el array de elementos para la capa de la vista
    $articulos = array();
    while ($fila = obtener_resultados($resultado))
    {
        $articulos[] = $fila;
    }

    // Cerrar la conexión
    cerrar_conexion($conexion);

    return $articulos;
}

```

Como se puede comprobar, la capa de acceso a datos no contiene funciones dependientes de ningún sistema gestor de bases de datos, por lo que es independiente de la base de datos utilizada. Además, las funciones creadas en la capa de abstracción de la base de datos se pueden reutilizar en otras funciones del modelo que necesiten acceder a la base de datos.

Nota Los ejemplos de los listados 2-6 y 2-7 no son completos, y todavía hace falta añadir algo de código para tener una completa abstracción de la base de datos (abstraer el código SQL mediante un constructor de consultas independiente de la base de datos, añadir todas las funciones a una clase, etc.) El propósito de este libro no es mostrar cómo se puede escribir todo ese código, ya que en el capítulo 8 se muestra cómo Symfony realiza de forma automática toda la abstracción.

2.1.2.2. Los elementos de la vista

La capa de la vista también puede aprovechar la separación de código. Las páginas web suelen contener elementos que se muestran de forma idéntica a lo largo de toda la aplicación: cabeceras de la página, el *layout* genérico, el pie de página y la navegación global. Normalmente sólo cambia el interior de la página. Por este motivo, la vista se separa en un *layout* y en una plantilla. Normalmente, el *layout* es global en toda la aplicación o al menos en un grupo de páginas. La plantilla sólo se encarga de visualizar las variables definidas en el controlador. Para que estos componentes interaccionen entre sí correctamente, es necesario añadir cierto código. Siguiendo estos principios, la parte de la vista del listado 2-3 se puede separar en tres partes, como se muestra en los listados 2-8, 2-9 y 2-10.

Listado 2-8 - La parte de la plantilla de la vista, en `miplantilla.php`

```

<h1>Listado de Artículos</h1>
<table>
<tr><th>Fecha</th><th>Título</th></tr>
<?php foreach ($articulos as $articulo): ?>
    <tr>
        <td><?php echo $articulo['fecha'] ?></td>
        <td><?php echo $articulo['titulo'] ?></td>
    </tr>
<?php endforeach; ?>
</table>

```


Listado 2-9 - La parte de la lógica de la vista

```
<?php

$titulo = 'Listado de Artículos';
$contenido = include('miplantilla.php');
```

Listado 2-10 - La parte del layout de la vista

```
<html>
  <head>
    <title><?php echo $titulo ?></title>
  </head>
  <body>
    <?php echo $contenido ?>
  </body>
</html>
```

2.1.2.3. Acciones y controlador frontal

En el ejemplo anterior, el controlador no se encargaba de realizar muchas tareas, pero en las aplicaciones web reales el controlador suele tener mucho trabajo. Una parte importante de su trabajo es común a todos los controladores de la aplicación. Entre las tareas comunes se encuentran el manejo de las peticiones del usuario, el manejo de la seguridad, cargar la configuración de la aplicación y otras tareas similares. Por este motivo, el controlador normalmente se divide en un controlador frontal, que es único para cada aplicación, y las acciones, que incluyen el código específico del controlador de cada página.

Una de las principales ventajas de utilizar un controlador frontal es que ofrece un punto de entrada único para toda la aplicación. Así, en caso de que sea necesario impedir el acceso a la aplicación, solamente es necesario editar el script correspondiente al controlador frontal. Si la aplicación no dispone de controlador frontal, se debería modificar cada uno de los controladores.

2.1.2.4. Orientación a objetos

Los ejemplos anteriores utilizan la programación procedimental. Las posibilidades que ofrecen los lenguajes de programación modernos para trabajar con objetos permiten simplificar la programación, ya que los objetos pueden encapsular la lógica, pueden heredar métodos y atributos entre diferentes objetos y proporcionan una serie de convenciones claras sobre la forma de nombrar a los objetos.

La implementación de una arquitectura MVC en un lenguaje de programación que no está orientado a objetos puede encontrarse con problemas de *namespaces* y código duplicado, dificultando la lectura del código de la aplicación.

La orientación a objetos permite a los desarrolladores trabajar con objetos de la vista, objetos del controlador y clases del modelo, transformando las funciones de los ejemplos anteriores en métodos. Se trata de un requisito obligatorio para las arquitecturas de tipo MVC.

Sugerencia Si quieres profundizar en el tema de los patrones de diseño para las aplicaciones web en el contexto de la orientación a objetos, puedes leer *"Patterns of Enterprise Application"*

Architecture" de Martin Fowler (Addison-Wesley, ISBN: 0-32112-742-0). El código de ejemplo del libro de Fowler está escrito en Java y en C#, pero es bastante fácil de leer para los programadores de PHP.

2.1.3. La implementación del MVC que realiza Symfony

Piensa por un momento cuántos componentes se necesitarían para realizar una página sencilla que muestre un listado de las entradas o artículos de un blog. Como se muestra en la figura 2-2, son necesarios los siguientes componentes:

- La capa del Modelo
 - Abstracción de la base de datos
 - Acceso a los datos
- La capa de la Vista
 - Vista
 - Plantilla
 - Layout
- La capa del Controlador
 - Controlador frontal
 - Acción

En total son siete scripts, lo que parecen muchos archivos para abrir y modificar cada vez que se crea una página. Afortunadamente, Symfony simplifica este proceso. Symfony toma lo mejor de la arquitectura MVC y la implementa de forma que el desarrollo de aplicaciones sea rápido y sencillo.

En primer lugar, el controlador frontal y el layout son comunes para todas las acciones de la aplicación. Se pueden tener varios controladores y varios layouts, pero solamente es obligatorio tener uno de cada. El controlador frontal es un componente que sólo tiene código relativo al MVC, por lo que no es necesario crear uno, ya que Symfony lo genera de forma automática.

La otra buena noticia es que las clases de la capa del modelo también se generan automáticamente, en función de la estructura de datos de la aplicación. La librería Propel se encarga de esta generación automática, ya que crea el *esqueleto* o estructura básica de las clases y genera automáticamente el código necesario. Cuando Propel encuentra restricciones de claves foráneas (o externas) o cuando encuentra datos de tipo fecha, crea métodos especiales para acceder y modificar esos datos, por lo que la manipulación de datos se convierte en un juego de niños. La abstracción de la base de datos es completamente invisible al programador, ya que la realiza otro componente específico llamado Creole. Así, si se cambia el sistema gestor de bases de datos en cualquier momento, no se debe reescribir ni una línea de código, ya que tan sólo es necesario modificar un parámetro en un archivo de configuración.

Por último, la lógica de la vista se puede transformar en un archivo de configuración sencillo, sin necesidad de programarla.

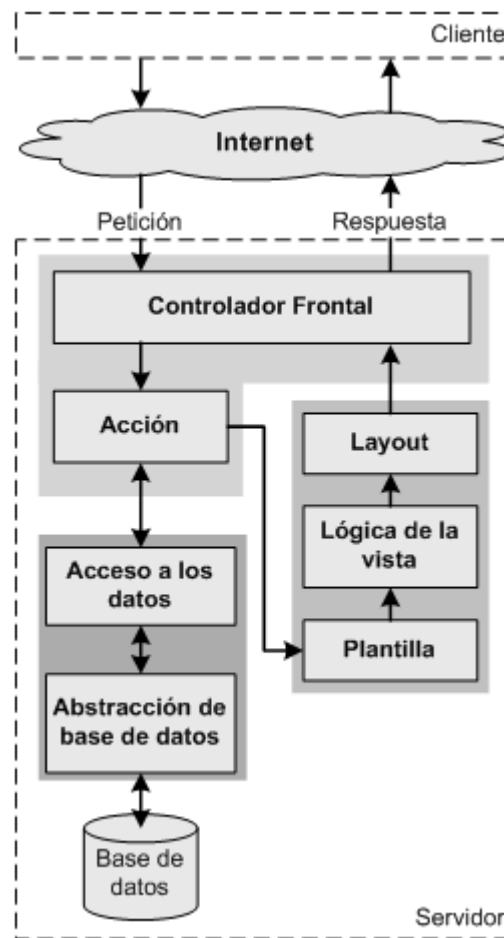


Figura 2.2. El flujo de trabajo de Symfony

Considerando todo lo anterior, el ejemplo de la página que muestra un listado con todas las entradas del blog solamente requiere de tres archivos en Symfony, que se muestran en los listados 2-11, 2-12 y 2-13.

Listado 2-11 - Acción listado, en `miproyecto/apps/miaplicacion/modules/weblog/actions/actions.class.php`

```
<?php

class weblogActions extends sfActions
{
    public function executeListado()
    {
        $this->articulos = ArtículoPeer::doSelect(new Criteria());
    }
}
```

Listado 2-12 - Plantilla listado, en `miproyecto/apps/miaplicacion/modules/weblog/templates/listadoSuccess.php`

```
<?php slot('titulo', 'Listado de Artículos') ?>

<h1>Listado de Artículos</h1>
<table>
```

```
<tr><th>Fecha</th><th>Título</th></tr>
<?php foreach ($articulos as $articulo): ?>
  <tr>
    <td><?php echo $articulo->getFecha() ?></td>
    <td><?php echo $articulo->getTitulo() ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

Además es necesario crear un layout como el del listado 2-13. Afortunadamente, el mismo layout se puede reutilizar muchas veces.

Listado 2-13 - Layout, en `miproyecto/apps/miaplicacion/templates/layout.php`

```
<html>
  <head>
    <title><?php include_slot('titulo') ?></title>
  </head>
  <body>
    <?php echo $sf_content ?>
  </body>
</html>
```

Estos scripts son todo lo que necesita la aplicación del ejemplo. El código mostrado es el necesario para crear la misma página que generaba el script simple del listado 2-1. Symfony se encarga del resto de tareas, como hacer que los componentes interactúen entre sí. Si se considera el número de líneas de código, el listado de entradas de blog creado según la arquitectura MVC no requiere más líneas ni más tiempo de programación que un script simple. Sin embargo, la arquitectura MVC proporciona grandes ventajas, como la organización del código, la reutilización, la flexibilidad y una programación mucho más entretenida. Por si fuera poco, crear la aplicación con Symfony permite crear páginas XHTML válidas, depurar fácilmente las aplicaciones, crear una configuración sencilla, abstracción de la base de datos utilizada, enrutamiento con URL *limpias*, varios entornos de desarrollo y muchas otras utilidades para el desarrollo de aplicaciones.

2.1.4. Las clases que forman el núcleo de Symfony

La implementación que realiza Symfony de la arquitectura MVC incluye varias clases que se mencionan una y otra vez a lo largo del libro:

- `sfController` es la clase del controlador. Se encarga de decodificar la petición y transferirla a la acción correspondiente.
- `sfRequest` almacena todos los elementos que forman la petición (parámetros, cookies, cabeceras, etc.)
- `sfResponse` contiene las cabeceras de la respuesta y los contenidos. El contenido de este objeto se transforma en la respuesta HTML que se envía al usuario.
- El contexto (que se obtiene mediante `sfContext::getInstance()`) almacena una referencia a todos los objetos que forman el núcleo de Symfony y puede ser accedido desde cualquier punto de la aplicación.

El capítulo 6 explica en detalle todos estos objetos.

Como se ha visto, todas las clases de Symfony utilizan el prefijo `sf`, como también hacen todas las variables principales de Symfony en las plantillas. De esta forma, se evitan las *colisiones* en los nombres de clases y variables de Symfony y los nombres de tus propias clases y variables, además de que las clases del framework son más fáciles de reconocer.

Nota Entre las normas seguidas por el código de Symfony, se encuentra el estándar *"UpperCamelCase"* para el nombre de las clases y variables. Solamente existen dos excepciones: las clases del núcleo de Symfony empiezan por `sf` (por tanto en minúsculas) y las variables utilizadas en las plantillas que utilizan la sintaxis de separar las palabras con guiones bajos.

Nota del traductor La notación *"CamelCase"* consiste en escribir frases o palabras compuestas eliminando los espacios intermedios y poniendo en mayúscula la primera letra de cada palabra. La variante *"UpperCamelCase"* también pone en mayúscula la primera letra de todas.

2.2. Organización del código

Ahora que ya conoces los componentes que forman una aplicación de Symfony, a lo mejor te estás preguntando sobre cómo están organizados. Symfony organiza el código fuente en una estructura de tipo proyecto y almacena los archivos del proyecto en una estructura estandarizada de tipo árbol.

2.2.1. Estructura del proyecto: Aplicaciones, Módulos y Acciones

Symfony considera un proyecto como *"un conjunto de servicios y operaciones disponibles bajo un determinado nombre de dominio y que comparten el mismo modelo de objetos"*.

Dentro de un proyecto, las operaciones se agrupan de forma lógica en aplicaciones. Normalmente, una aplicación se ejecuta de forma independiente respecto de otras aplicaciones del mismo proyecto. Lo habitual es que un proyecto contenga dos aplicaciones: una para la parte pública y otra para la parte de gestión, compartiendo ambas la misma base de datos. También es posible definir proyectos que estén formados por varios sitios web pequeños, cada uno de ellos considerado como una aplicación. En este caso, es importante tener en cuenta que los enlaces entre aplicaciones se deben indicar de forma absoluta.

Cada aplicación está formada por uno o más módulos. Un módulo normalmente representa a una página web o a un grupo de páginas con un propósito relacionado. Por ejemplo, una aplicación podría tener módulos como `home`, `articulos`, `ayuda`, `carritoCompra`, `cuenta`, etc.

Los módulos almacenan las acciones, que representan cada una de las operaciones que se puede realizar en un módulo. Por ejemplo el módulo `carritoCompra` puede definir acciones como `anadir`, `mostrar` y `actualizar`. Normalmente las acciones se describen mediante verbos. Trabajar con acciones es muy similar a trabajar con las páginas de una aplicación web tradicional, aunque en este caso dos acciones diferentes pueden acabar mostrando la misma página (como por ejemplo la acción de añadir un comentario a una entrada de un blog, que acaba volviendo a mostrar la página de la entrada con el nuevo comentario).

Nota Nota del traductor En el párrafo anterior, la acción del carrito se llama `anadir` y no `añadir`, ya que el nombre de una acción también se utiliza como parte del nombre de un fichero y como parte del nombre de una función, por lo que se recomienda utilizar exclusivamente caracteres ASCII, y por tanto, no debería utilizarse la letra ñ.

Sugerencia Si crees que todo esto es demasiado complicado para tu primer proyecto con Symfony, puedes agrupar todas las acciones en un único módulo, para simplificar la estructura de archivos. Cuando la aplicación se complique, puedes reorganizar las acciones en diferentes módulos. Como se comenta en el capítulo 1, la acción de reescribir el código para mejorar su estructura o hacerlo más sencillo (manteniendo siempre su comportamiento original) se llama refactorización, y es algo muy común cuando se aplican los principios del RAD ("*desarrollo rápido de aplicaciones*").

La figura 2-3 muestra un ejemplo de organización del código para un proyecto de un blog, siguiendo la estructura de proyecto / aplicación / módulo / acción. No obstante, la estructura de directorios real del proyecto es diferente al esquema mostrado por esa figura.

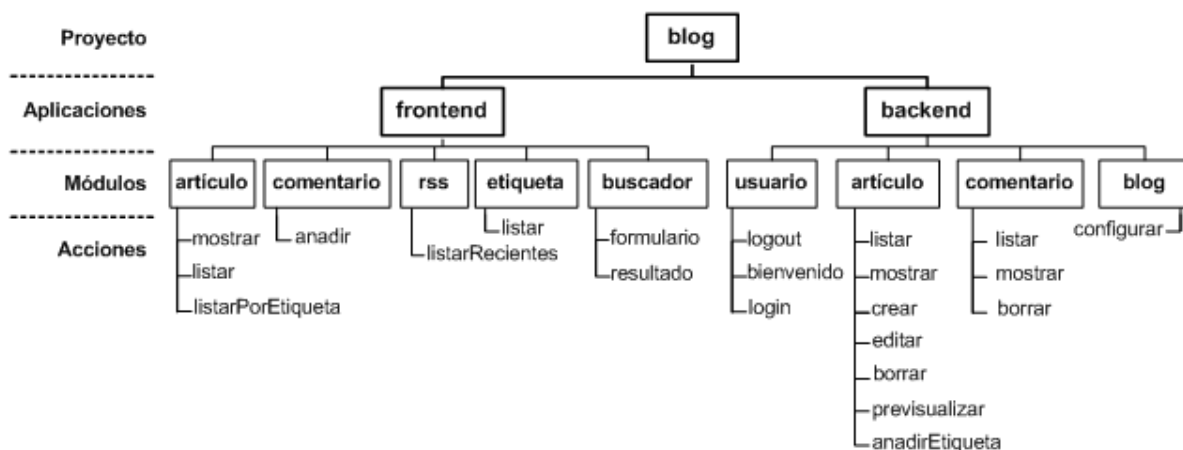


Figura 2.3. Ejemplo de organización del código

2.2.2. Estructura del árbol de archivos

Normalmente, todos los proyectos web comparten el mismo tipo de contenidos, como por ejemplo:

- Una base de datos, como MySQL o PostgreSQL
- Archivo estáticos (HTML, imágenes, archivos de JavaScript, hojas de estilos, etc.)
- Archivos subidos al sitio web por parte de los usuarios o los administradores
- Clases y librerías PHP
- Librerías externas (scripts desarrollados por terceros)
- Archivos que se ejecutan por lotes (*batch files*) que normalmente son scripts que se ejecutan vía línea de comandos o mediante cron
- Archivos de log (las trazas que generan las aplicaciones y/o el servidor)
- Archivos de configuración

Symfony proporciona una estructura en forma de árbol de archivos para organizar de forma lógica todos esos contenidos, además de ser consistente con la arquitectura MVC utilizada y con la agrupación proyecto / aplicación / módulo. Cada vez que se crea un nuevo proyecto, aplicación o módulo, se genera de forma automática la parte correspondiente de esa estructura. Además, la estructura se puede personalizar completamente, para reorganizar los archivos y directorios o para cumplir con las exigencias de organización de un cliente.

2.2.2.1. Estructura de la raíz del proyecto

La raíz de cualquier proyecto Symfony contiene los siguientes directorios:

```
apps/  
  frontend/  
  backend/  
cache/  
config/  
data/  
  sql/  
doc/  
lib/  
  model/  
log/  
plugins/  
test/  
  bootstrap/  
  unit/  
  functional/  
web/  
  css/  
  images/  
  js/  
  uploads/
```

La tabla 2-1 describe los contenidos de estos directorios

Tabla 2-1. Directorios en la raíz de los proyectos Symfony

Directorio	Descripción
apps/	Contiene un directorio por cada aplicación del proyecto (normalmente, frontend y backend para la parte pública y la parte de gestión respectivamente)
cache/	Contiene la versión cacheada de la configuración y (si está activada) la versión cacheada de las acciones y plantillas del proyecto. El mecanismo de cache (que se explica en el Capítulo 12) utiliza los archivos de este directorio para acelerar la respuesta a las peticiones web. Cada aplicación contiene un subdirectorio que guarda todos los archivos PHP y HTML preprocesados
config/	Almacena la configuración general del proyecto
data/	En este directorio se almacenan los archivos relacionados con los datos, como por ejemplo el esquema de una base de datos, el archivo que contiene las instrucciones SQL para crear las tablas e incluso un archivo de bases de datos de SQLite
doc/	Contiene la documentación del proyecto, formada por tus propios documentos y por la documentación generada por PHPdoc

lib/	Almacena las clases y librerías externas. Se suele guardar todo el código común a todas las aplicaciones del proyecto. El subdirectorio <code>model/</code> guarda el modelo de objetos del proyecto (como se describe en el Capítulo 8)
log/	Guarda todos los archivos de log generados por Symfony. También se puede utilizar para guardar los logs del servidor web, de la base de datos o de cualquier otro componente del proyecto. Symfony crea un archivo de log por cada aplicación y por cada entorno (los archivos de log se ven detalladamente en el Capítulo 16)
plugins/	Almacena los plugins instalados en la aplicación (el Capítulo 17 aborda el tema de los plugins)
test/	Contiene las pruebas unitarias y funcionales escritas en PHP y compatibles con el framework de pruebas de Symfony (que se explica en el capítulo 15). Cuando se crea un proyecto, Symfony crea algunos pruebas básicas
web/	La raíz del servidor web. Los únicos archivos accesibles desde Internet son los que se encuentran en este directorio

2.2.2.2. Estructura de cada aplicación

Todas las aplicaciones de Symfony tienen la misma estructura de archivos y directorios:

```
apps/
  [nombre aplicacion]/
    config/
    i18n/
    lib/
    modules/
    templates/
    layout.php
```

La tabla 2-2 describe los subdirectorios de una aplicación

Tabla 2-2. Subdirectorios de cada aplicación Symfony

Directorio	Descripción
config/	Contiene un montón de archivos de configuración creados con YAML. Aquí se almacena la mayor parte de la configuración de la aplicación, salvo los parámetros propios del framework. También es posible redefinir en este directorio los parámetros por defecto si es necesario. El Capítulo 5 contiene más detalles sobre la configuración de las aplicaciones
i18n/	Contiene todos los archivos utilizados para la internacionalización de la aplicación, sobre todo los archivos que traducen la interfaz (el Capítulo 13 detalla la internacionalización). La internacionalización también se puede realizar con una base de datos, en cuyo caso este directorio no se utilizaría
lib/	Contiene las clases y librerías utilizadas exclusivamente por la aplicación
modules/	Almacena los módulos que definen las características de la aplicación
templates/	Contiene las plantillas globales de la aplicación, es decir, las que utilizan todos los módulos. Por defecto contiene un archivo llamado <code>layout.php</code> , que es el layout principal con el que se muestran las plantillas de los módulos

Nota En las aplicaciones recién creadas, los directorios `i18n/`, `lib/` y `modules/` están vacíos.

Las clases de una aplicación no pueden acceder a los métodos o atributos de otras aplicaciones del mismo proyecto. Además, los enlaces entre 2 aplicaciones de un mismo proyecto se deben indicar de forma absoluta. Esta última restricción es importante durante la inicialización del proyecto, que es cuando debes elegir como dividir el proyecto en aplicaciones.

2.2.2.3. Estructura de cada módulo

Cada aplicación contiene uno o más módulos. Cada módulo tiene su propio subdirectorio dentro del directorio `modules` y el nombre del directorio es el que se elige durante la creación del módulo.

Esta es la estructura de directorios típica de un módulo:

```
apps/  
  [nombre aplicacion]/  
    modules/  
      [nombre modulo]/  
        actions/  
          actions.class.php  
        config/  
        lib/  
        templates/  
          indexSuccess.php
```

La tabla 2-3 describe los subdirectorios de un módulo.

Tabla 2-3. Subdirectorios de cada módulo

Directorio	Descripción
actions/	Normalmente contiene un único archivo llamado <code>actions.class.php</code> y que corresponde a la clase que almacena todas las acciones del módulo. También es posible crear un archivo diferente para cada acción del módulo
config/	Puede contener archivos de configuración adicionales con parámetros exclusivos del módulo
lib/	Almacena las clases y librerías utilizadas exclusivamente por el módulo
templates/	Contiene las plantillas correspondientes a las acciones del módulo. Cuando se crea un nuevo módulo, automáticamente se crea la plantilla llamada <code>indexSuccess.php</code>

Nota En los módulos recién creados, los directorios `config/` y `lib/` están vacíos.

2.2.2.4. Estructura del sitio web

Existen pocas restricciones sobre la estructura del directorio `web`, que es el directorio que contiene los archivos que se pueden acceder de forma pública. Si se utilizan algunas convenciones básicas en los nombres de los subdirectorios, se pueden simplificar las plantillas. La siguiente es una estructura típica del directorio `web`:

```
web/  
  css/  
  images/  
  js/  
  uploads/
```

Normalmente, los archivos estáticos se organizan según los directorios de la tabla 2-4.

Tabla 2-4. Subdirectorios habituales en la carpeta web

Directorio	Descripción
css/	Contiene los archivos de hojas de estilo creados con CSS (archivos con extensión .css)
images/	Contiene las imágenes del sitio con formato .jpg, .png o .gif
js/	Contiene los archivos de JavaScript con extensión .js
uploads/	Se pueden almacenar los archivos subidos por los usuarios. Aunque normalmente este directorio contiene imágenes, no se debe confundir con el directorio que almacena las imágenes del sitio (images/). Esta distinción permite sincronizar los servidores de desarrollo y de producción sin afectar a las imágenes subidas por los usuarios

Nota Aunque es muy recomendable mantener la estructura definida por defecto, es posible modificarla para adaptarse a las necesidades específicas de cada proyecto, como por ejemplo los proyectos que se ejecutan en servidores con sus propias estructuras de directorios definidas y con otras políticas para el desarrollo de las aplicaciones. El Capítulo 19 explica en detalle cómo modificar la estructura de directorios definida por Symfony.

2.3. Herramientas comunes

Algunas técnicas se utilizan una y otra vez en Symfony, por lo que es fácil encontrarse con ellas a lo largo de este libro y en el desarrollo de tus proyectos. Entre estas técnicas se encuentran los contenedores de parámetros (*parameter holders*), las constantes y la carga automática de clases.

2.3.1. Contenedores de parámetros

Muchas de las clases de Symfony contienen algún contenedor de parámetros. Se trata de una forma eficiente de encapsular los atributos y así poder utilizar métodos *getter* y *setter* sencillos. La clase `sfRequest` por ejemplo incluye un contenedor de parámetros que se puede obtener mediante el método `getParameterHolder()`. Todos los contenedores de parámetros almacenan sus datos de la misma forma, como se muestra en el listado 2-14.

Listado 2-14 - Uso del contenedor de parámetros de `sfRequest`

```
$petition->getParameterHolder()->set('parametro', 'valor');  
  
echo $petition->getParameterHolder()->get('parametro');  
=> 'valor'
```

La mayoría de clases que contienen contenedores de parámetros proporcionan métodos abreviados para las operaciones de tipo get/set. La clase `sfRequest` es una de esas clases, ya que el código abreviado del listado 2-15 obtiene el mismo resultado que el código original del listado 2-14.

Listado 2-15 - Uso de los métodos abreviados del contenedor de parámetros de `sfResponse`

```
$peticion->setParameter('parametro', 'valor');  
  
echo $peticion->getParameter('parametro');  
=> 'valor'
```

El método *getter* del contenedor de parámetros permite la definición de un segundo parámetro que actúa de valor por defecto. De esta manera, se obtiene una protección efectiva y sencilla frente a los errores. El listado 2-16 contiene un ejemplo de su uso.

Listado 2-16 - Uso de valores por defecto en las funciones de tipo *getter*

```
// El parámetro llamado 'parametro' no está definido, por lo que el getter devuelve un  
valor vacío  
echo $peticion->getParameter('parametro');  
=> null  
  
// El valor por defecto se puede obtener con sentencias condicionales  
if ($peticion->hasParameter('parametro'))  
{  
    echo $peticion->getParameter('parametro');  
}  
else  
{  
    echo 'valor_por_defecto';  
}  
=> 'valor_por_defecto'  
  
// El siguiente método es mucho más rápido  
echo $peticion->getParameter('parametro', 'valor_por_defecto');  
=> 'valor_por_defecto'
```

Algunas clases del núcleo de Symfony utilizan un contenedor de parámetros que permite el uso de *namespaces* (gracias a la clase `sfNamespacedParameterHolder`). Si se utiliza un tercer parámetro en un *getter* o en un *setter*, ese parámetro se utiliza como *namespace* del parámetro y por tanto, el parámetro sólo estará definido dentro de ese *namespace*. El listado 2-17 muestra un ejemplo.

Listado 2-17 - Uso de un *namespace* en el contenedor de parámetros de `sfUser`

```
$usuario->setAttribute('parametro', 'valor1');  
$usuario->setAttribute('parametro', 'valor2', 'mi/namespacespace');  
echo $usuario->getAttribute('parametro');  
=> 'valor1'  
echo $usuario->getAttribute('parametro', null, 'mi/namespacespace');  
=> 'valor2'
```

También es posible añadir contenedores de parámetros a tus propias clases, para aprovechar las ventajas de su sintaxis. El listado 2-18 muestra un ejemplo de cómo definir una clase con un contenedor de parámetros.

Listado 2-18 - Añadir un contenedor de parámetros a una clase

```
class MiClase  
{  
    protected $contenedorParametros = null;
```

```
public function initialize($parametros = array())
{
    $this->contenedorParametros = new sfParameterHolder();
    $this->contenedorParametros->add($parametros);
}

public function getContenedorParametros()
{
    return $this->contenedorParametros;
}
}
```

2.3.2. Constantes

Aunque pueda parecer sorprendente, el código de Symfony no incluye ninguna constante. La razón es que las constantes de PHP tienen un inconveniente: no se puede modificar su valor una vez definidas. Por este motivo, Symfony utiliza su propio objeto para almacenar la configuración, llamado `sfConfig`, y que reemplaza a las constantes. Este objeto proporciona métodos estáticos para poder acceder a los parámetros desde cualquier punto de la aplicación. El listado 2-19 muestra el uso de los métodos de la clase `sfConfig`.

Listado 2-19 - Uso de los métodos de la clase `sfConfig` en vez de constantes

```
// En vez de constantes de PHP...
define('MI_CONSTANTE', 'valor');
echo MI_CONSTANTE;

// ...Symfony utiliza el objeto sfConfig
sfConfig::set('mi_constante', 'valor');
echo sfConfig::get('mi_constante');
```

Los métodos de `sfConfig` permiten definir valores por defecto y se puede invocar el método `sfConfig::set()` más de una vez sobre el mismo parámetro para modificar su valor. El capítulo 5 detalla el uso de los métodos de `sfConfig`.

2.3.3. Carga automática de clases

Normalmente, cuando se utiliza un método de una clase o cuando se crea un objeto en PHP, se debe incluir antes la definición de esa clase.

```
include 'clases/MiClase.php';
$miObjeto = new MiClase();
```

Sin embargo, en los proyectos complejos con muchas clases y una estructura de directorios con muchos niveles, requiere mucho trabajo incluir todas las clases necesarias indicando correctamente la ruta de cada clase. Symfony incluye una función `spl_autoload_register()` para evitar la necesidad de los `include` y así poder escribir directamente:

```
$miObjeto = new MiClase();
```

En este caso, Symfony busca la definición de la clase `MiClase` en todos los archivos con extensión `.php` que se encuentran en alguno de los directorios `lib/` del proyecto. Si se encuentra la definición de la clase, se incluye de forma automática.

De esta forma, si se guardan todas las clases en los directorios `lib/`, no es necesario incluir las clases de forma explícita. Por este motivo, los proyectos de Symfony no suelen incluir instrucciones de tipo `include` o `require`.

Nota Para mejorar el rendimiento, la carga automática de clases de Symfony busca durante la primera petición en una serie de directorios (que se definen en un archivo interno de configuración). Una vez realizada la búsqueda en los directorios, se guarda el nombre de todas las clases encontradas y su ruta de acceso en un array asociativo de PHP. Así, las siguientes peticiones no tienen que volver a mirar todos los directorios en busca de las clases. Este comportamiento implica que se debe borrar la cache de Symfony cada vez que se añade o se mueve una clase del proyecto (salvo en el entorno de desarrollo, donde no es necesario). El comando utilizado para borrar la cache es `symfony cache:clear`, salvo en el entorno de desarrollo, donde Symfony borra automáticamente la caché una vez cuando no encuentra una clase. El Capítulo 12 explica con detalle el mecanismo de cache y la configuración de la carga automática de clases se muestra en el capítulo 19.

2.4. Resumen

El uso de un framework que utiliza MVC obliga a dividir y organizar el código de acuerdo a las convenciones establecidas por el framework. El código de la presentación se guarda en la vista, el código de manipulación de datos se guarda en el modelo y la lógica de procesamiento de las peticiones constituye el controlador. Aplicar el patrón MVC a una aplicación resulta bastante útil además de restrictivo.

Symfony es un framework de tipo MVC escrito en PHP 5. Su estructura interna se ha diseñado para obtener lo mejor del patrón MVC y la mayor facilidad de uso. Gracias a su versatilidad y sus posibilidades de configuración, Symfony es un framework adecuado para cualquier proyecto de aplicación web.

Ahora que ya has aprendido la teoría que está detrás de Symfony, estas casi preparado para desarrollar tu primera aplicación. Pero antes de eso, necesitas tener instalado Symfony en tu servidor de desarrollo.

?

Capítulo 3. Ejecutar aplicaciones Symfony

Como se ha visto en los capítulos anteriores, el framework Symfony está formado por un conjunto de archivos escritos en PHP. Los proyectos realizados con Symfony utilizan estos archivos, por lo que la instalación de Symfony consiste en obtener esos archivos y hacer que estén disponibles para los proyectos.

La versión mínima de PHP requerida para ejecutar Symfony 1.1 es PHP 5.1. Por tanto, es necesario asegurarse de que se encuentra instalado, para lo cual se puede ejecutar el siguiente comando en la línea de comandos del sistema operativo:

```
> php -v  
  
PHP 5.2.5 (cli) (built: Nov 20 2007 16:55:40)  
Copyright (c) 1997-2007 The PHP Group  
Zend Engine v2.2.0, Copyright (c) 1998-2007 Zend Technologies
```

Si el número de la versión que se muestra es 5.1 o superior, ya es posible realizar la instalación de Symfony que se describe en este capítulo.

3.1. Instalando el entorno de pruebas

Si lo único que quieres es comprobar lo que puede dar de sí Symfony, lo mejor es que te decantes por la instalación rápida. En este caso, se utiliza el *"entorno de pruebas"* o *sandbox*.

El entorno de pruebas está formado por un conjunto de archivos. Contiene un proyecto vacío de Symfony e incluye todas las librerías necesarias (Symfony, Lime, Creole, Propel y Phing), una aplicación de prueba y la configuración básica. No es necesario realizar ninguna configuración en el servidor ni instalar ningún paquete adicional para que funcione correctamente.

Para instalar el entorno de pruebas, se debe descargar su archivo comprimido desde http://www.symfony-project.org/get/sf_sandbox_1_1.tgz. Una vez descargado el archivo, es esencial asegurarse que tiene la extensión `.tgz`, ya que de otro modo no se descomprimirá correctamente. La extensión `.tgz` no es muy común en sistemas operativos tipo Windows, pero programas como WinRAR o 7-Zip lo pueden descomprimir sin problemas. A continuación, se descomprime su contenido en el directorio raíz del servidor web, que normalmente es `web/` o `www/`. Para asegurar cierta uniformidad en la documentación, en este capítulo se supone que se ha descomprimido el entorno de pruebas en el directorio `sf_sandbox/`.

Cuidado Para hacer pruebas en un servidor local, se pueden colocar todos los archivos en la raíz del servidor web. Sin embargo, se trata de una mala práctica para los servidores de producción, ya que los usuarios pueden ver el funcionamiento interno de la aplicación.

Se puede comprobar si se ha realizado correctamente la instalación del entorno de pruebas mediante los comandos proporcionados por Symfony. Entra en el directorio `sf_sandbox/` y ejecuta el siguiente comando:

```
> php symfony -V
```

El resultado del comando debería mostrar la versión del entorno de pruebas:

```
symfony version 1.1.0 (/ruta/hasta/el/directorio/lib/dir/utilizado/por/el/entorno/de/pruebas)
```

A continuación, se prueba si el servidor web puede acceder al entorno de pruebas mediante la siguiente URL:

```
http://localhost/sf_sandbox/web/frontend_dev.php/
```

Si todo ha ido bien, deberías ver una página de bienvenida como la que se muestra en la figura 3-1, con lo que la instalación rápida se puede dar por concluida. Si no se muestra esa página, se mostrará un mensaje de error que te indica los cambios necesarios en la configuración. También puedes consultar la sección "Resolución de problemas" que se encuentra más adelante en este capítulo.



Figura 3.1. Página de bienvenida del entorno de pruebas

El entorno de pruebas está pensado para que practiques con Symfony en un servidor local, no para desarrollar aplicaciones complejas que acaban siendo publicadas en la web. No obstante, la versión de Symfony que está incluida en el entorno de pruebas es completamente funcional y equivalente a la que se instala vía PEAR.

Para desinstalar el entorno de pruebas, borra el directorio `sf_sandbox/` de la carpeta `web/` de tu servidor.

3.2. Instalando las librerías de Symfony

Al desarrollar aplicaciones con Symfony, es probable que tengas que instalarlo dos veces: una para el entorno de desarrollo y otra para el servidor de producción (a no ser que el servicio de *hosting* que utilices tenga Symfony preinstalado). En cada uno de los servidores lo lógico es evitar duplicidades juntando todos los archivos de Symfony en un único directorio, independientemente de que desarrolles una o varias aplicaciones.

Como el desarrollo de Symfony evoluciona rápidamente, es posible que esté disponible una nueva versión estable del framework unos días después de la primera instalación. La actualización del framework es algo a tener muy en cuenta, por lo que se trata de otra razón de peso para juntar en un único directorio todas las librerías de Symfony.

Existen dos alternativas para instalar las librerías necesarias para el desarrollo de las aplicaciones:

- La instalación que utiliza PEAR es la recomendada para la mayoría de usuarios. Con este método, la instalación es bastante sencilla, además de ser fácil de compartir y de actualizar.
- La instalación que utiliza Subversion (SVN) solamente se recomienda para los programadores de PHP más avanzados y es el método con el que pueden obtener los últimos parches, pueden añadir sus propias características al framework y pueden colaborar con el proyecto Symfony.

Symfony integra algunos paquetes externos:

- `lime` es una utilidad para las pruebas unitarias.
- `Creole` es un sistema de abstracción de la base de datos. Se trata de un sistema similar a los *PHP Data Objects* (PDO) y proporciona una interfaz entre el código PHP y el código SQL de la base de datos, permitiendo cambiar fácilmente de sistema gestor de bases de datos.
- `Propel` se utiliza para el ORM. Proporciona persistencia para los objetos y un servicio de consultas.
- `Phing` es una utilidad que emplea `Propel` para generar las clases del modelo.

`Lime` ha sido desarrollado por el equipo de Symfony. `Creole`, `Propel` y `Phing` han sido creados por otros equipos de desarrollo y se publican bajo la licencia *GNU Lesser Public General License* (LGPL). Todos estos paquetes están incluidos en Symfony.

Sugerencia El framework Symfony dispone de una licencia de tipo MIT. Todos los avisos de copyright del software externo que incluye Symfony se encuentran en el archivo COPYRIGHT y todas sus licencias se encuentran en el directorio `licenses/`

3.2.1. Instalando Symfony con PEAR

El paquete PEAR de Symfony incluye las librerías propias de Symfony y todas sus dependencias. Además, también contiene un script que permite extender la línea de comandos del sistema para que funcione el comando `symfony`.

Para instalar Symfony de esta manera, en primer lugar se debe añadir el *canal Symfony* a PEAR mediante este comando:

```
| > pear channel-discover pear.symfony-project.com
```

Para comprobar las librerías disponibles en ese canal, se puede ejecutar lo siguiente:

```
| > pear remote-list -c symfony
```

Una vez añadido el canal, ya es posible instalar la última versión estable de Symfony mediante el siguiente comando:

```
| > pear install symfony/symfony
|
| downloading symfony-1.1.0.tgz ...
| Starting to download symfony-1.1.0.tgz (1,283,270 bytes)
| .....
| .....
| .....done: 1,283,270 bytes
| install ok: channel://pear.symfony-project.com/symfony-1.1.0
```

Y la instalación ya ha terminado. Los archivos y las utilidades de línea de comandos de Symfony ya se han instalado. Para asegurarte de que se ha instalado correctamente, prueba a ejecutar el comando `symfony` para que te muestre la versión de Symfony que se encuentra instalada:

```
| > symfony -V
|
| symfony version 1.1.0 (/ruta/hasta/el/directorio/lib/dir/de/Symfony/en/PEAR)
```

Después de la instalación, las librerías de Symfony se encuentran en los siguientes directorios:

- `$php_dir/symfony/` contiene las principales librerías.
- `$data_dir/symfony/` contiene los archivos web que utilizan por defecto los módulos de Symfony.
- `$doc_dir/symfony/` contiene la documentación.
- `$test_dir/symfony/` contiene las pruebas unitarias y funcionales de Symfony.

Las variables que acaban en `_dir` se definen en la configuración de PEAR. Para ver sus valores, puedes ejecutar el siguiente comando:

```
| > pear config-show
```

3.2.2. Obtener Symfony mediante el repositorio SVN

En los servidores de producción, o cuando no es posible utilizar PEAR, se puede descargar la última versión de las librerías Symfony directamente desde el repositorio Subversion que utiliza Symfony:

```
> mkdir /ruta/a/symfony
> cd /ruta/a/symfony
> svn checkout http://svn.symfony-project.com/tags/RELEASE_1_1_0/ .
```

El comando `symfony`, que solamente está disponible en las instalaciones PEAR, en realidad es una llamada al script que se encuentra en `/ruta/a/symfony/data/bin/symfony`. Por tanto, en una instalación realizada con SVN, el comando `symfony -V` es equivalente a:

```
> php /ruta/a/symfony/data/bin/symfony -V

symfony version 1.1.0 (/ruta/hasta/el/directorio/lib/dir/de/Symfony/en/SVN)
```

Si instalas Symfony mediante Subversion, es posible que ya tuvieras creado un proyecto con Symfony. Para que ese proyecto utilice la nueva versión de Symfony es necesario modificar el valor de la ruta definida en el archivo `lib/ProjectConfiguration.class.php` del proyecto, como se muestra a continuación:

```
<?php

require_once '/ruta/hasta/lib/autoload/sfCoreAutoload.class.php';
sfCoreAutoload::register();

class ProjectConfiguration extends sfProjectConfiguration
{
    // ...
}
```

El Capítulo 19 muestra otras opciones para enlazar un proyecto con una instalación de Symfony, incluyendo el uso de enlaces simbólicos y rutas relativas.

Sugerencia Otra forma de instalar Symfony es bajar directamente el paquete de PEAR (<http://pear.symfony-project.com/get/symfony-1.1.0.tgz>) y descomprimirlo en algún directorio. El resultado de esta instalación es el mismo que si se instala mediante el repositorio de Subversion.

3.3. Crear una aplicación web

Como se vio en el Capítulo 2, Symfony agrupa las aplicaciones relacionadas en proyectos. Todas las aplicaciones de un proyecto comparten la misma base de datos. Por tanto, para crear una aplicación web en primer lugar se debe crear un proyecto.

3.3.1. Crear el Proyecto

Los proyectos de Symfony siguen una estructura de directorios predefinida. Los comandos que proporciona Symfony permiten automatizar la creación de nuevos proyectos, ya que se encargan de crear la estructura de directorios básica del proyecto y con los permisos adecuados. Por

tanto, para crear un proyecto se debe crear un directorio y decirle a Symfony que cree un proyecto en su interior.

Si has utilizado la instalación con PEAR, ejecuta los siguientes comandos:

```
> mkdir ~/miproyecto
> cd ~/miproyecto
> symfony generate:project miproyecto
```

Si has instalado Symfony mediante SVN, puedes crear un proyecto con los siguientes comandos:

```
> mkdir ~/miproyecto
> cd ~/miproyecto
> php /ruta/hasta/data/bin/symfony generate:project miproyecto
```

La estructura de directorios creada por Symfony se muestra a continuación:

```
apps/
cache/
config/
data/
doc/
lib/
log/
plugins/
test/
web/
```

Sugerencia La tarea `generate:project` añade un script llamado `symfony` en el directorio raíz del proyecto. Este script es idéntico al comando `symfony` que instala PEAR, por lo que se puede utilizar la instrucción `php symfony` en vez del comando `symfony` cuando no se dispone de las utilidades de la línea de comandos (lo que sucede cuando se instala Symfony mediante Subversion).

3.3.2. Crear la aplicación

El proyecto recién creado está incompleto, ya que requiere por lo menos de una aplicación. Para crear la aplicación, se utiliza el comando `symfony generate:app`, al que se le tiene que pasar como argumento el nombre de la nueva aplicación:

```
> php symfony generate:app frontend
```

El comando anterior crea un directorio llamado `frontend/` dentro del directorio `apps/` que se encuentra en la raíz del proyecto. Por defecto se crea una configuración básica de la aplicación y una serie de directorios:

```
apps/
  frontend/
    config/
    i18n/
    lib/
    modules/
    templates/
```

En el directorio `web` del proyecto también se crean algunos archivos PHP correspondientes a los controladores frontales de cada uno de los entornos de ejecución de la aplicación:

```
web/  
  index.php  
  frontend_dev.php
```

El archivo `index.php` es el controlador frontal de producción de la nueva aplicación. Como se trata de la primera aplicación, Symfony crea un archivo llamado `index.php` en vez de `frontend.php` (si después se crea una nueva aplicación llamada por ejemplo `backend`, el controlador frontal del entorno de producción que se crea se llamará `backend.php`). Para ejecutar la aplicación en el entorno de desarrollo, se debe ejecutar el controlador frontal llamado `frontend_dev.php`. Por razones de seguridad el controlador frontal de desarrollo sólo se puede ejecutar por defecto desde `localhost`. El Capítulo 5 explica en detalle los distintos entornos de ejecución.

El comando `symfony` siempre se ejecuta en el directorio raíz del proyecto (`miproyecto/` en los ejemplos anteriores) porque todas las tareas que ejecuta este comando dependen del proyecto.

3.4. Configurar el servidor web

Los scripts que se encuentran en el directorio `web/` son los únicos puntos de entrada a la aplicación. Por este motivo, debe configurarse el servidor web para que puedan ser accedidos desde Internet. En el servidor de desarrollo y en los servicios de *hosting* profesionales, se suele tener acceso a la configuración completa de Apache para poder configurar servidores virtuales (*virtual host*). En los servicios de alojamiento compartido, lo normal es tener acceso solamente a los archivos `.htaccess`.

3.4.1. Configurar los servidores virtuales

El listado 3-1 muestra un ejemplo de la configuración necesaria para crear un nuevo servidor virtual en Apache mediante la modificación del archivo `httpd.conf`.

Listado 3-1 - Ejemplo de configuración de Apache, en `apache/conf/httpd.conf`

```
<VirtualHost *:80>  
  ServerName miaplicacion.ejemplo.com  
  DocumentRoot "/home/juan/miproyecto/web"  
  DirectoryIndex index.php  
  Alias /sf /$sf_symfony_data_dir/web/sf  
  <Directory "/$sf_symfony_data_dir/web/sf">  
    AllowOverride All  
    Allow from All  
  </Directory>  
  <Directory "/home/steve/miproyecto/web">  
    AllowOverride All  
    Allow from All  
  </Directory>  
</VirtualHost>
```

En la configuración del listado 3-1, se debe sustituir la variable `$sf_symfony_data_dir` por la ruta real del directorio de datos de Symfony. Por ejemplo, la ruta en un sistema *nix en el que se ha instalado Symfony mediante PEAR sería:

```
| Alias /sf /usr/local/lib/php/data/symfony/web/sf
```

Nota No es obligatorio el alias al directorio `web/sf/`. La finalidad del alias es permitir que Apache pueda encontrar las imágenes, hojas de estilos y archivos de JavaScript utilizados en la barra de depuración, en el generador automático de aplicaciones de gestión, en las páginas propias de Symfony y en las utilidades de Ajax. La alternativa a crear este alias podría ser la de crear un enlace simbólico (symlink) o copiar directamente los contenidos del directorio `/sf_symfony_data_dir/web/sf/` al directorio `miproyecto/web/sf/`.

No te olvides reiniciar Apache para que los cambios surtan efecto. La aplicación recién creada ya se puede acceder con cualquier navegador en esta dirección:

```
| http://localhost/frontend_dev.php/
```

Al acceder a la aplicación, se debería mostrar una imagen similar a la mostrada en la figura 3-1.

Symfony utiliza la reescritura de URL para mostrar "URL limpias" en la aplicación, es decir, URL con mucho sentido, optimizadas para buscadores y que ocultan a los usuarios los detalles técnicos internos de la aplicación. El Capítulo 9 explica en detalle el sistema de enrutamiento utilizado por Symfony y su implicación en las URL de las aplicaciones.

Para que funcione correctamente la reescritura de URL, es necesario que Apache esté compilado con el módulo `mod_rewrite` o al menos que esté instalado el módulo `mod_rewrite` como módulo DSO. En este último caso, la configuración de Apache debe contener las siguientes líneas en el archivo `httpd.conf`:

```
| AddModule mod_rewrite.c
| LoadModule rewrite_module modules/mod_rewrite.so
```

Para los servidores web IIS (Internet Information Services) es necesario disponer de `isapi/rewrite` instalado y activado. El sitio web del proyecto Symfony dispone de un tutorial sobre cómo instalar Symfony en servidores IIS (http://www.symfony-project.org/cookbook/1_1/en/web_server_iis).

3.4.2. Configurar un servidor compartido

En los servidores de alojamiento compartido es un poco más complicado instalar las aplicaciones creadas con Symfony, ya que los servidores suelen tener una estructura de directorios que no se puede modificar.

Cuidado No es recomendable hacer las pruebas y el desarrollo directamente en un servidor compartido. Una de las razones es que la aplicación es pública incluso cuando no está terminada, pudiendo mostrar su funcionamiento interno y pudiendo provocar problemas de seguridad. El otro motivo es que el rendimiento de los servidores compartidos habituales no es suficiente como para depurar la aplicación con las utilidades de Symfony. Por este motivo, no se recomienda comenzar el desarrollo de una aplicación en un servidor compartido, sino que debería desarrollarse en un servidor local y subirla al servidor compartido una vez terminada la

aplicación. En el Capítulo 16 se muestran técnicas y herramientas para la instalación de las aplicaciones.

Imaginemos que el servidor compartido llama a la carpeta web `www/` en vez de `web/` y que no es posible modificar el archivo de configuración `httpd.conf`, sino que solo es posible acceder a un archivo de tipo `.htaccess` en ese directorio.

Los proyectos creados con Symfony permiten configurar cada ruta de cada directorio. En el Capítulo 19 se detalla la configuración de los directorios, pero mientras tanto, se va a renombrar el directorio `web` a `www` y se va a modificar la configuración de la aplicación para que lo tenga en cuenta. El listado 3-2 muestra los cambios que es preciso añadir al final del archivo `lib/ProjectConfiguration.class.php`.

Listado 3-2 - Modificación de la estructura de directorios por defecto, en `lib/ProjectConfiguration.class.php`

```
class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
        $this->setWebDir($this->getRootDir().'/www');
    }
}
```

La carpeta web de la raíz del servidor contiene por defecto un archivo de tipo `.htaccess`. El listado 3-3 muestra su contenido, que debe ser modificado de acuerdo a los requerimientos de tu servidor compartido.

Listado 3-3 - Configuración por defecto de `.htaccess`, ahora guardado en `miproyecto/www/.htaccess`

```
Options +FollowSymLinks +ExecCGI

<IfModule mod_rewrite.c>
    RewriteEngine On

    # uncomment the following line, if you are having trouble
    # getting no_script_name to work
    #RewriteBase /

    # we skip all files with .something
    #RewriteCond %{REQUEST_URI} \.+
    #RewriteCond %{REQUEST_URI} !\.html$
    #RewriteRule .* - [L]

    # we check if the .html version is here (caching)
    RewriteRule ^$ index.html [QSA]
    RewriteRule ^([^.]+)$ $1.html [QSA]
    RewriteCond %{REQUEST_FILENAME} !-f

    # no, so we redirect to our front web controller
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

Después de realizar los cambios, ya debería ser posible acceder a la aplicación. Comprueba si se muestra la página de bienvenida accediendo a esta dirección:

```
| http://www.ejemplo.com/frontend_dev.php/
```

Symfony permite realizar otras configuraciones de servidor. Por ejemplo se puede acceder a las aplicaciones Symfony utilizando alias en vez de servidores virtuales. También es posible ejecutar las aplicaciones Symfony en servidores IIS. Existen tantas técnicas como posibles configuraciones, aunque el propósito de este libro no es explicarlas todas.

Para encontrar ayuda sobre las distintas configuraciones de servidor, puedes consultar el wiki del proyecto Symfony (<http://trac.symfony-project.com/>) en el que existen varios tutoriales con explicaciones detalladas paso a paso.

3.5. Resolución de problemas

Si se producen errores durante la instalación, lo mejor es intentar mostrar los mensajes de error en el navegador o en la consola de comandos. Normalmente los errores muestran pistas sobre su posible causa y hasta pueden contener enlaces a algunos recursos disponibles en Internet sobre ese problema.

3.5.1. Problemas típicos

Si continúan los problemas con Symfony, puedes comprobar los siguientes errores comunes:

- Algunas instalaciones de PHP incluyen tanto PHP 4 como PHP 5. En este caso, suele ser habitual que el comando de PHP 5 sea `php5`, por lo que se debe ejecutar la instrucción `php5 symfony` en vez de `symfony`. Puede que también sea necesario añadir la directiva `SetEnv PHP_VER 5` en el archivo de configuración `.htaccess` e incluso puede que tengas que renombrar los scripts del directorio `web/` para que tengan una extensión `.php5` en vez de la tradicional extensión `.php`. Cuando se intenta ejecutar Symfony con PHP 4, el error que se muestra es similar al siguiente:

```
| Parse error, unexpected ',', expecting '(' in .../symfony.php on line 19.
```

- El límite de memoria utilizado por PHP se define en el archivo de configuración `php.ini` y debería valer por lo menos 32M (equivalente a 32 MB). El síntoma común de este problema es cuando se muestra un mensaje de error al instalar Symfony mediante PEAR o cuando se utiliza la línea de comandos:

```
| Allowed memory size of 8388608 bytes exhausted
```

- La directiva `zend.ze1_compatibility_mode` del archivo de configuración de PHP (`php.ini`) debe tener un valor igual a `off`. Si no es así, cuando se intenta acceder a cualquier script, se muestra el siguiente mensaje de error:

```
| Strict Standards: Implicit cloning object of class 'sfTimer' because of  
| 'zend.ze1_compatibility_mode'
```

- Los directorios `log/` y `cache/` del proyecto deben tener permiso de escritura para el servidor web. Si se ejecuta una aplicación sin estos permisos, se muestra la siguiente excepción:

```
| sfCacheException [message] Unable to write cache file"/usr/miproyecto/cache/frontend/  
| prod/config/config_config_handlers.yml.php"
```

- La ruta del sistema debe incluir la ruta al comando php, y la directiva `include_path` del archivo de configuración de PHP (`php.ini`) debe contener una ruta a PEAR (en el caso de que se utilice PEAR).
- En ocasiones, existe más de un archivo `php.ini` en el sistema (por ejemplo cuando se instala PHP mediante el paquete WAMP). En estos casos, se puede realizar una llamada a la función `phpinfo()` de PHP para saber la ruta exacta del archivo `php.ini` que está utilizando la aplicación.

Nota Aunque no es obligatorio, es muy recomendable por motivos de rendimiento establecer el valor `off` a las directivas `magic_quotes_gpc` y `register_globals` del archivo de configuración de PHP (`php.ini`).

3.5.2. Recursos relacionados con Symfony

Existen varias formas de encontrar soluciones a los problemas típicos y que ya les han ocurrido a otros usuarios:

- El foro de instalación de Symfony (<http://www.symfony-project.org/forum/>) contiene muchas preguntas sobre configuraciones en diferentes plataformas, entornos y servidores.
- La lista de correo de usuarios de Symfony (<http://groups.google.es/group/symfony-es>) permite buscar en sus archivos de mensajes, por lo que es posible que encuentres a otros usuarios que han pasado por los mismos problemas.
- El wiki de Symfony (<http://trac.symfony-project.com/#Installingsymfony>) contiene tutoriales detallados paso a paso sobre la instalación de Symfony que han sido creados por otros usuarios.

Si no encuentras la respuesta en esos recursos, puedes preguntar a la comunidad de Symfony. Las preguntas puedes hacerlas en el foro, en la lista de correo e incluso en el canal IRC de Symfony (`#symfony`) para obtener la respuesta de los miembros más activos de la comunidad.

3.6. Versionado del código fuente

Una vez creada la aplicación, se recomienda empezar con el versionado del código fuente (también llamado *control de versiones*). El versionado almacena todas las modificaciones realizadas en el código, permite acceder a las versiones anteriores de cualquier archivo, simplifica la creación de parches y permite trabajar en equipo de forma eficiente. Symfony soporta de forma nativa el uso de CVS, aunque recomienda el uso de Subversion (<http://subversion.tigris.org/>). Los ejemplos que se muestran a continuación utilizan comandos de Subversion y presuponen que existe un servidor de Subversion instalado y que se va a crear un nuevo repositorio para el proyecto. Para los usuarios de Windows, se recomienda utilizar TortoiseSVN (<http://tortoisetsvn.tigris.org/>) como cliente de Subversion. La documentación oficial de Subversion es un buen recurso para ampliar los conocimientos sobre el versionado del código y sobre los comandos que utilizan los siguientes ejemplos.

Los siguientes ejemplos requieren que exista una variable de entorno llamada `$SVNREP_DIR` y cuyo valor es la ruta completa al repositorio. Si no es posible definir la variable de entorno, en los siguientes comandos se debe escribir la ruta completa al repositorio en vez de `$SVNREP_DIR`.

En primer lugar se crea un nuevo repositorio para el proyecto `miproyecto`:

```
| > svnadmin create $SVNREP_DIR/miproyecto
```

Después se crea el layout o estructura básica del repositorio mediante los directorios `trunk`, `tags` y `branches`. El comando necesario es bastante largo:

```
| > svn mkdir -m "Creacion del layout" file:/// $SVNREP_DIR/miproyecto/trunk  
file:/// $SVNREP_DIR/miproyecto/tags file:/// $SVNREP_DIR/miproyecto/branches
```

A continuación se realiza la primera versión, para lo que es necesario importar todos los archivos del proyecto salvo los archivos temporales de `cache/` y `log/`:

```
| > cd ~/miproyecto  
| > rm -rf cache/*  
| > rm -rf log/*  
| > svn import -m "Primera importacion" . file:/// $SVNREP_DIR/miproyecto/trunk
```

El siguiente comando permite comprobar si se han subido correctamente los archivos:

```
| > svn ls file:/// $SVNREP_DIR/miproyecto/trunk/
```

Por el momento todo va bien, ya que ahora el repositorio SVN contiene una versión de referencia de todos los archivos del proyecto. De esta forma, los archivos del directorio `miproyecto/` deben hacer referencia a los que almacena el repositorio. Para ello, renombra el directorio `miproyecto/` (si todo funciona correctamente lo podrás borrar) y descarga los contenidos del repositorio en un nuevo directorio:

```
| > cd ~  
| > mv miproyecto miproyecto.original  
| > svn co file:/// $SVNREP_DIR/miproyecto/trunk miproyecto  
| > ls miproyecto
```

Y eso es todo. Ahora ya es posible trabajar con los archivos que se encuentran en el directorio `miproyecto/` y subir todos los cambios al repositorio. Puedes borrar el directorio `miproyecto.original/` porque ya no se utiliza.

Solamente es necesario realizar una última configuración. Si se suben todos los archivos del directorio al repositorio, se van a copiar algunos archivos innecesarios, como los que se encuentran en los directorios `cache/` y `log/`. Subversion permite establecer una lista de archivos que se ignoran al subir los contenidos al repositorio. Además, es preciso establecer de nuevo los permisos correctos a los directorios `cache/` y `log/`:

```
| > cd ~/miproyecto  
| > chmod 777 cache  
| > chmod 777 log  
| > svn propedit svn:ignore log  
| > svn propedit svn:ignore cache
```

Al ejecutar los comandos anteriores, Subversion muestra el editor de textos configurado por defecto. Si no se muestra nada, configura el editor de textos que utiliza Subversion por defecto mediante el siguiente comando:

```
> export SVN_EDITOR=<nombre_del_editor_de_textos>  
> svn propedit svn:ignore log  
> svn propedit svn:ignore cache
```

Para incluir todos los archivos de los directorios, se debe escribir lo siguiente cuando se muestre el editor de textos:

```
| *
```

Para finalizar, guarda los cambios y cierra el editor.

3.7. Resumen

Para probar y *jugar* con Symfony en un servidor local, la mejor opción es instalar el entorno de pruebas o *sandbox*, que contiene un entorno de ejecución preconfigurado para Symfony.

Para desarrollar aplicaciones web reales o para instalarlo en un servidor de producción, se puede optar por la instalación via PEAR o mediante el repositorio de Subversion. Estos métodos instalan las librerías de Symfony, pero se deben crear manualmente los proyectos y las aplicaciones. El último paso de la configuración de las aplicaciones es la configuración del servidor web, que puede realizarse de muchas formas. Symfony funciona muy bien con los servidores virtuales y de hecho es el método recomendado.

Si se producen errores durante la instalación, existen muchos tutoriales y *preguntas frecuentes* en el sitio web de Symfony. Incluso es posible trasladar tu problema a la comunidad Symfony para obtener una respuesta en general rápida y efectiva.

Después de crear el proyecto, se recomienda empezar con el versionado del código fuente para realizar el control de versiones.

Una vez que ya se puede utilizar Symfony, es un buen momento para desarrollar la primera aplicación web básica.

Capítulo 4. Introducción a la creación de páginas

Curiosamente, el primer tutorial que utilizan los programadores para aprender cualquier lenguaje de programación o framework es el que muestra por pantalla el mensaje "¡Hola Mundo!" (del inglés *Hello, world!*). Resulta extraño creer que un ordenador pueda ser capaz de saludar a todo el mundo, ya que todos los intentos que ha habido hasta ahora en el campo de la inteligencia artificial han resultado en unos sistemas artificiales de conversación bastante pobres. No obstante, Symfony no es más tonto que cualquier otro framework, y la prueba es que se puede crear una página que muestre el mensaje "Hola, <tu_nombre>".

En este capítulo se muestra como crear un módulo, que es el elemento que agrupa a las páginas. También se aprende cómo crear una página, que a su vez se divide en una acción y una plantilla, siguiendo la arquitectura MVC. Las interacciones básicas con las páginas se realizan mediante enlaces y formularios, por lo que también se muestra como incluirlos en las plantillas y como manejarlos en las acciones.

4.1. Crear el esqueleto del módulo

Como se vio en el Capítulo 2, Symfony agrupa las páginas en módulos. Por tanto, antes de crear una página es necesario crear un módulo, que inicialmente no es más que una estructura vacía de directorios y archivos que Symfony puede reconocer.

La línea de comandos de Symfony automatiza la creación de los módulos. Sólo se necesita llamar a la tarea `generate:module` indicando como argumentos el nombre de la aplicación y el nombre del nuevo módulo. En el capítulo anterior se creó una aplicación llamada `frontend`. Para añadirle un módulo llamado `contenido`, se deben ejecutar los siguientes comandos:

```
> cd ~/miproyecto
> php symfony generate:module frontend contenido

>> dir+      ~/miproyecto/apps/frontend/modules/contenido/actions
>> file+     ~/miproyecto/apps/frontend/modules/contenido/actions/actions.class.php
>> dir+      ~/miproyecto/apps/frontend/modules/contenido/templates
>> file+     ~/miproyecto/apps/frontend/modules/contenido/templates/indexSuccess.php
>> file+     ~/miproyecto/test/functional/frontend/contenidoActionsTest.php
>> tokens    ~/miproyecto/test/functional/frontend/contenidoActionsTest.php
>> tokens    ~/miproyecto/apps/frontend/modules/contenido/actions/actions.class.php
>> tokens    ~/miproyecto/apps/frontend/modules/contenido/templates/indexSuccess.php
```

Además de los directorios `actions/` y `templates/` este comando crea tres archivos. El archivo que se encuentra en el directorio `test/` está relacionado con las pruebas funcionales, que se ven en el Capítulo 15. El archivo `actions.class.php` (que se muestra en el listado 4-1) redirige la acción a la página de bienvenida del módulo por defecto. Por último, el archivo `templates/indexSuccess.php` está vacío.

Listado 4-1 - La acción generada por defecto, en actions/actions.class.php

```
<?php

class contenidoActions extends sfActions
{
    public function executeIndex()
    {
        $this->forward('default', 'module');
    }
}
```

Nota Si se abre el archivo `actions.class.php` generado realmente, su contenido es mucho mayor que las pocas líneas mostradas anteriormente, incluyendo un montón de comentarios. Symfony recomienda utilizar comentarios de PHP para documentar el proyecto y por tanto añade a cada archivo de cada clase comentarios que son compatibles con el formato de la herramienta phpDocumentor (<http://www.phpdoc.org/>).

En cada nuevo módulo, Symfony crea una acción por defecto llamada `index`. La acción completa se compone del método `executeIndex` de la acción y del archivo de su plantilla llamada `indexSuccess.php`. El significado del prefijo `execute` y del sufijo `Success` se explican detalladamente en los Capítulos 6 y 7 respectivamente. Por el momento se puede considerar que esta forma de nombrar a los archivos y métodos es una convención que sigue Symfony. Para visualizar la página generada (que se muestra en la figura 4-1) se debe acceder a la siguiente dirección en un navegador:

| http://localhost/frontend_dev.php/contenido/index

En este capítulo no se utiliza la acción `index`, por lo que se puede borrar el método `executeIndex()` del archivo `actions.class.php` y también se puede borrar el archivo `indexSuccess.php` del directorio `templates/`.

Nota Symfony permite crear los módulos sin necesidad de utilizar la línea de comandos. Uno de esos métodos es crear manualmente todos los directorios y archivos necesarios. En otras ocasiones, las acciones y las plantillas de un módulo se emplean para manipular los datos de una tabla de la base de datos. Como el código necesario para crear, obtener, actualizar y borrar los datos casi siempre es el mismo, Symfony incorpora un mecanismo que permite generar de forma automática todo el código PHP de un módulo de este tipo. El Capítulo 14 contiene los detalles de esta técnica.



Figura 4.1. La página de índice generada automáticamente

4.1.1. Añadir una página

En Symfony la lógica o código de las páginas se define en la acción y la presentación se define en las plantillas. Las páginas estáticas que no requieren de ninguna lógica necesitan definir una acción vacía.

4.1.1.1. Añadir una acción

En este ejemplo, la página que muestra el mensaje "¡Hola Mundo!" se puede acceder mediante una acción llamada ver. Para crearla, solamente es necesario añadir el método executeVer en la clase contenidoActions, como muestra el Listado 4-2.

Listado 4-2 - Añadir una acción es equivalente a añadir un método de tipo *execute* en la clase de la acción

```
<?php
class contenidoActions extends sfActions
{
```

```
public function executeVer()  
{  
}  
}
```

El nombre del método de la acción siempre es `execute + xxxxxxx + ()`, donde la segunda parte del nombre es el nombre de la acción con la primera letra en mayúsculas.

Por tanto, si ahora se accede a la siguiente dirección:

```
| http://localhost/frontend_dev.php/contenido/ver
```

Symfony mostrará un mensaje de error indicando que la plantilla `verSuccess.php` no existe. Se trata de un error normal por el momento, ya que las páginas siempre están formadas por una acción y una plantilla.

Cuidado Las URL (no los dominios) distinguen mayúsculas y minúsculas, y por tanto también las distingue Symfony, (aunque el nombre de los métodos en PHP no distingue mayúsculas de minúsculas). Por tanto, si se añade un método llamado `executever()` o `executeVER()`, y se intenta acceder desde el navegador a `ver`, Symfony muestra un mensaje de error de tipo 404 (Página no encontrada).

Symfony incluye un sistema de enrutamiento que permite separar completamente el nombre real de la acción y la forma de la URL que se utiliza para llamar a la acción. De esta forma, es posible personalizar las URL como si fueran una parte más de la respuesta. La estructura de directorios del servidor o los parámetros de la petición ya no son obstáculos para construir URL con cualquier formato; la URL de una acción puede construirse siguiendo cualquier formato. Por ejemplo, la URL típica de la acción `index` de un módulo llamado `articulo` suele tener el siguiente aspecto:

```
| http://localhost/frontend_dev.php/articulo/index?id=123
```

Esta URL se emplea para obtener un artículo almacenado en la base de datos. En el ejemplo anterior, se obtiene un artículo cuyo identificador es 123, que pertenece a la sección de artículos de Europa y que trata sobre la economía en Francia. Con un simple cambio en el archivo `routing.yml`, la URL anterior se puede construir de la siguiente manera:

```
| http://localhost/articulos/europa/francia/economia.html
```

La URL que se obtiene no solo es mejor desde el punto de vista de los buscadores, sino que es mucho más significativa para el usuario medio, que incluso puede utilizar la barra de direcciones como si fuera una especie de línea de comandos para realizar consultas a medida, como por ejemplo la siguiente URL:

```
| http://localhost/articulos/etiquetas/economia+francia+euro
```

Symfony es capaz de procesar y generar este tipo de URL inteligentes. El sistema de enrutamiento es capaz de extraer automáticamente los parámetros de la petición y ponerlos a disposición de la acción. También es capaz de formatear los enlaces incluidos en la respuesta para que también sean enlaces de tipo *inteligente*. El Capítulo 9 explica en detalle el sistema de enrutamiento.

En resumen, el nombrado de las acciones no se debe realizar teniendo en cuenta la URL que se utilizará para acceder a ellas, sino que se deberían nombrar según la función de la acción dentro de la aplicación. El nombre de la acción explica su funcionalidad, por lo que suele ser un verbo en su forma de infinitivo (como por ejemplo `ver`, `listar`, `modificar`, etc.). El nombre de las acciones se puede ocultar a los usuarios, por lo que si es necesario, se pueden utilizar nombres muy explícitos para las acciones (como por ejemplo `listarPorNombre` o `verConComentarios`). Con este tipo de nombres, no son necesarios demasiados comentarios para explicar la funcionalidad de la acción y el código fuente resultante es mucho más fácil de comprender.

4.1.1.2. Añadir una plantilla

La acción espera una plantilla para mostrarse en pantalla. Una plantilla es un archivo que está ubicado en el directorio `templates/` de un módulo, y su nombre está compuesto por el nombre de la acción y el resultado de la misma. El resultado por defecto es `success` (exitoso), por lo que el archivo de plantilla que se crea para la acción `ver` se llamará `verSuccess.php`.

Se supone que las plantillas sólo deben contener código de presentación, así que procura mantener la menor cantidad de código PHP en ellas como sea posible. De hecho, una página que muestre "¡Hola, mundo!" puede tener una plantilla tan simple como la del Listado 4-3.

Listado 4-3 - La plantilla contenido/templates/verSuccess.php

```
| <p>¡Hola, mundo!</p>
```

Si necesitas ejecutar algún código PHP en la plantilla, es mejor evitar la sintaxis usual de PHP, como se muestra en el Listado 4-4. En su lugar, es preferible escribir las plantillas utilizando la sintaxis alternativa de PHP, mostrada en el Listado 4-5, para mantener el código entendible para personas sin conocimientos de PHP. De esta forma, no sólo el código final estará correctamente indentado, sino que además ayudará a mantener el código complejo de PHP en la acción, dado que sólo las estructuras de control (`if`, `foreach`, `while` y demás) poseen una sintaxis alternativa.

Listado 4-4 - La sintaxis tradicional de PHP, buena para las acciones, pero mala para las plantillas

```
| <p>¡Hola, mundo!</p>
| <?php
|
| if ($prueba)
| {
|     echo "<p>".time()."</p>";
| }
|
| ?>
```

Listado 4-5 - La sintaxis alternativa de PHP, buena para las plantillas

```
| <p>¡Hola, mundo!</p>
| <?php if ($prueba): ?>
|     <p><?php echo time(); ?></p>
| <?php endif; ?>
```

Sugerencia Una buena regla para comprobar si la sintaxis de la plantilla es lo suficientemente legible, es que el archivo no debe contener código HTML generado por PHP mediante la función

echo, ni tampoco llaves. Y en la mayoría de los casos, al abrir una etiqueta `<?php`, debería cerrarse con `?>` en la misma línea.

4.1.2. Transfiriendo información de la acción a la plantilla

La tarea de la acción es realizar los cálculos complejos, obtener datos, realizar comprobaciones, y crear o inicializar las variables necesarias para que se presenten o se utilicen en la plantilla. Symfony hace que los atributos de la clase de la acción (disponibles vía `$this->nombreDeVariable` en la acción), estén directamente accesibles en la plantilla en el ámbito global (vía `$nombreVariable`). Los listados 4-6 y 4-7 muestran cómo pasar información de la acción a la plantilla.

Listado 4-6 - Configurando un atributo de acción dentro de ella para hacerlo disponible para la plantilla

```
<?php

class contenidoActions extends sfActions
{
    public function executeVer()
    {
        $hoy = getdate();
        $this->hora = $hoy['hours'];
    }
}
```

Listado 4-7 - La plantilla tiene acceso directo a los atributos de la acción

```
<p>␣Hola, Mundo!</p>
<?php if ($hora >= 18): ?>
    <p>Quizás debería decir buenas tardes. Ya son las <?php echo $hora ?>.</p>
<?php endif; ?>
```

El uso de etiquetas cortas de apertura (`<?=>`, equivalente a `<?php echo`) no se recomienda para aplicaciones web profesionales, debido a que el servidor web de producción puede ser capaz de entender más de un lenguaje de script, y por tanto, confundirse. Además, las etiquetas cortas de apertura no funcionan con la configuración por defecto de PHP y necesitan de ajustes en el servidor para ser activadas. Por último, a la hora de lidiar con XML y la validación, fallará inmediatamente porque `<?` tiene un significado especial en XML.

Nota La plantilla es capaz de acceder a algunos datos sin necesidad de definir variables en la acción. Cada plantilla puede invocar métodos de los objetos `$sf_context`, `$sf_request`, `$sf_params` y `$sf_user`. Esos métodos contienen datos relacionados con el contexto actual, la petición y sus parámetros, y la sesión. Más adelante se muestra cómo utilizarlos de manera eficiente.

4.2. Enlazando a otra acción

Ya se ha comentado que existe una independencia total entre el nombre de la acción y la URL utilizada para llamarla, por lo que si se crea un enlace a actualizar en una plantilla como en el

Listado 4-10, sólo funcionará con el enrutamiento establecido por defecto. Si más tarde se decide modificar la manera de mostrar las URL, entonces será necesario verificar todas las plantillas para modificar los enlaces o hipervínculos.

Listado 4-10 - Forma clásica de incluir los enlaces

```
<a href="/frontend_dev.php/contenido/actualizar?nombre=anonimo">
  Nunca digo mi nombre
</a>
```

Para evitar este inconveniente, es necesario siempre utilizar el *helper* `link_to()` para crear enlaces a las acciones de la aplicación. Si lo único que quieres es la URL del enlace, entonces debes utilizar el *helper* `url_for()`.

Un *helper* es una función PHP definida por Symfony y que está pensada para ser utilizada en las plantillas. Los *helpers* generan código HTML y normalmente resultan más eficientes que escribir a mano ese mismo código HTML. El Listado 4-11 muestra el uso de los *helpers* para enlaces.

Listado 4-11 - Los *helpers* `link_to()` y `url_for()`

```
<p>¡Hola, Mundo!</p>
<?php if ($hora >= 18): ?>
<p>Quizás debería decir buenas tardes. Ya son las <?php echo $hora ?>.</p>
<?php endif; ?>
<form method="post" action="<?php echo url_for('contenido/actualizar') ?>">
  <label for="nombre">¿Cómo te llamas?</label>
  <input type="text" name="nombre" id="nombre" value="" />
  <input type="submit" value="Ok" />
  <?php echo link_to('Nunca digo mi nombre','contenido/actualizar?nombre=anonymous') ?>
</form>
```

El código HTML resultante es el mismo que el anterior, pero en este caso, si se modifican las reglas de enrutamiento, todas las plantillas siguen funcionando correctamente ya que se actualizan las URL de forma automática.

Los formularios merecen un capítulo completo para ellos, ya que Symfony provee muchas herramientas, sobre todo *helpers*, para facilitar tu trabajo. En el capítulo 10 aprenderás todo sobre estos *helpers*.

El *helper* `link_to()`, al igual que muchos otros, acepta un argumento para opciones especiales y atributos de etiqueta adicionales. El Listado 4-12 muestra un ejemplo de un argumento `option` y su código HTML resultante. El argumento `option` puede ser tanto un array asociativo como una simple cadena de texto mostrando pares de `clave=valor` separados por espacios.

Listado 4-12 - La mayoría de los *helpers* aceptan un argumento `option`

```
// Argumento "option" como un array asociativo
<?php echo link_to('Nunca digo mi nombre', 'contenido/actualizar?nombre=anonimo',
  array(
    'class' => 'enlace_especial',
    'confirm' => '¿Estás seguro?',
    'absolute' => true
  )) ?>
```

```
// Argumento "option" como una cadena de texto
<?php echo link_to('Nunca digo mi nombre', 'contenido/actualizar?nombre=anonimo',
    'class=enlace_especial confirm=¿Estás seguro? absolute=true') ?>

// Las dos funciones generan el mismo resultado
=> <a class="enlace_especial" onclick="return confirm('¿Estás seguro?');"
    href="http://localhost/frontend_dev.php/contenido/actualizar/nombre/anonimo">
    Nunca digo mi nombre</a>
```

Siempre que se utiliza un *helper* de Symfony que devuelve una etiqueta HTML, es posible insertar atributos de etiqueta adicionales (como el atributo `class` en el ejemplo del Listado 4-12) en el argumento `option`. Incluso es posible escribir estos atributos a la vieja usanza que utiliza HTML 4.0 (sin comillas dobles), y Symfony se encargará de mostrarlos correctamente formateados en XHTML. Esta es otra razón por la que los *helpers* son más rápidos de escribir que el HTML puro.

Nota Dado que requiere un procesamiento y transformación adicional, la sintaxis de cadena de texto es un poco más lenta que la sintaxis en forma de array.

Al igual que el resto de *helpers* de Symfony, los *helpers* de enlaces son muy numerosos y tienen muchas opciones. En el capítulo 9 se explican todos estos *helpers* con más detalle.

4.3. Obteniendo información de la petición

El método `getParameter()` del objeto `sfRequest` permite recuperar desde la acción los datos relacionados con la información que envía el usuario a través de un formulario (normalmente en una petición POST) o a través de la URL (mediante una petición GET). El Listado 4-13 muestra cómo es posible obtener el valor del parámetro `nombre` en la acción `actualizar`.

Listado 4-13 - Recuperando datos de la petición dentro de una acción

```
<?php

class contenidoActions extends sfActions
{
    // ...

    public function executeActualizar($petition)
    {
        $this->nombre = $petition->getParameter('nombre');
    }
}
```

Para que el código sea más sencillo, a todos los métodos de tipo `executeXxx()` se les pasa como primer argumento el objeto que representa al `sfRequest` actual.

Si la manipulación de datos es simple, ni siquiera es necesario utilizar la acción para recuperar los parámetros de la petición. La plantilla tiene acceso a un objeto llamado `$sf_params` que ofrece un método `get()` para recuperar los parámetros de la petición, tal y como hace el método `getParameter()` en la acción.

Si la acción `executeActualizar` estuviera vacía, el listado 4-14 muestra cómo se puede obtener el valor del parámetro `nombre` desde la plantilla `actualizarSuccess.php`.

Listado 4-14 - Obteniendo datos de la petición directamente en la plantilla

```
| <p>Hola, <?php echo $sf_params->get('nombre') ?>|</p>
```

Nota ¿Y por qué no utilizar en cambio las variables `$_POST`, `$_GET`, or `$_REQUEST`? Porque entonces las URL serían formateadas de manera diferente (como en `http://localhost/articulos/europa/francia/economia.html`, sin `?` ni `=`), las variables comunes de PHP ya no funcionarían, y sólo el sistema de enrutamiento sería capaz de recuperar los parámetros de petición. Además, seguramente quieras agregar un filtro a los datos de la petición para prevenir código malicioso, lo cual sólo es posible si se mantienen todos los parámetros de petición en un contenedor de parámetros.

El objeto `$sf_params` es más potente que simplemente añadir una especie de *getter* a un array. Por ejemplo, si sólo se desea probar la existencia de un parámetro de petición, se puede utilizar simplemente el método `$sf_parameter->has()`, en lugar de comprobar el valor en cuestión con `get()`, tal como en el Listado 4-15.

Listado 4-15 - Comprobando la existencia de un parámetro de petición en la plantilla

```
| <?php if ($sf_params->has('nombre')): ?>
|   <p>␣Hola, <?php echo $sf_params->get('nombre') ?>|</p>
| <?php else: ?>
|   <p>␣Hola, Juan Pérez!</p>
| <?php endif; ?>
```

Como puede que hayas adivinado, el código anterior puede escribirse en una sola línea. Al igual que la mayoría de los métodos *getter* de Symfony, tanto el método `$peticion->getParameter()` en la acción, como el método `$sf_params->get()` en la plantilla (que por cierto llama al mismo método del mismo objeto), aceptan un segundo argumento: el valor por defecto a utilizar si dicho parámetro de petición no está presente.

```
| <p>␣Hola, <?php echo $sf_params->get('nombre', 'Juan Pérez') ?>|</p>
```

4.4. Resumen

En Symfony, las páginas están compuestas por una acción (un método del archivo `actions/actions.class.php` con el prefijo `execute`) y una plantilla (un archivo en el directorio `templates/`, normalmente terminado en `Success.php`). Las páginas se agrupan en módulos, de acuerdo a su función en la aplicación. Escribir plantillas es muy sencillo con la ayuda de los *helpers*, funciones provistas por Symfony para generar código HTML. Además es necesario pensar que la URL es parte de la respuesta, por lo que se puede formatear de cualquier forma que se necesite, sólo debes abstenerte de utilizar cualquier referencia directa a la URL en el nombre de la acción o al recuperar un parámetro de petición.

Una vez aprendidos estos principios básicos, es posible escribir una aplicación web completa con Symfony. Pero te costaría mucho tiempo, dado que casi cualquier tarea a completar durante

el transcurso del desarrollo de la aplicación, se simplifica de una forma u otra por alguna funcionalidad de Symfony...motivo por el que este libro aún no termina.

Capítulo 5. Configurar Symfony

Para simplificar su uso, Symfony define una serie de convenciones o normas que se ajustan a los requisitos habituales de las aplicaciones web estándar. De todas formas, los archivos de configuración, a pesar de ser tan sencillos de utilizar, son lo suficientemente potentes como para personalizar cualquier aspecto del framework y la forma en que interactúan las aplicaciones. También es posible con estos archivos de configuración añadir parámetros específicos para las aplicaciones.

En este capítulo se explica cómo funciona el mecanismo de configuración:

- La configuración de Symfony se guarda en archivos escritos con YAML, aunque se puede utilizar otro formato.
- En la estructura de directorios del proyecto, existen archivos de configuración a nivel de proyecto, de aplicación y de módulo.
- También es posible definir conjuntos de opciones de configuración. En Symfony, un conjunto de opciones de configuración se llama *entorno*.
- Desde cualquier punto del código de la aplicación se puede acceder a los valores establecidos en los archivos de configuración.
- Además, Symfony permite utilizar código PHP dentro de los archivos YAML y algún que otro truco más para hacer más flexible el sistema de configuración.

5.1. El sistema de configuración

La mayoría de aplicaciones web comparten una serie de características, independientemente de su finalidad. Por ejemplo, es habitual restringir algunas partes de la aplicación a una serie de usuarios, utilizar un layout común para mostrar todas las páginas, los formularios deben volver a mostrar los datos que ha introducido el usuario después de una validación errónea. El framework define el comportamiento básico de estas características y el programador puede adaptar cada una mediante las opciones de configuración. Esta forma de trabajar ahorra mucho tiempo de desarrollo, ya que muchos cambios importantes no necesitan modificar ni siquiera una línea de código, aunque estos cambios impliquen muchos cambios internos. Además se trata de una forma mucho más eficiente, ya que permite asegurar que toda la configuración se encuentra en un lugar único y fácilmente localizable.

No obstante, este método también tiene dos inconvenientes muy importantes:

- Los programadores acaban escribiendo archivos XML complejos y muy largos.
- En una arquitectura basada en PHP, cada petición consumiría mucho más tiempo de proceso.

Considerando todas estas desventajas, Symfony utiliza solamente lo mejor de los archivos de configuración. De hecho, el objetivo del sistema de configuración de Symfony es ser:

- **Potente:** todo lo que puede ser gestionado con archivos de configuración, se gestiona con archivos de configuración.
- **Simple:** muchas de las características de la configuración no se utilizan habitualmente, por lo que las aplicaciones normales no tienen que tratar con ellas.
- **Sencillo:** los archivos de configuración son sencillos de leer, de modificar y de crear por parte de los desarrolladores.
- **Personalizable:** el lenguaje que se utiliza por defecto en los archivos de configuración es YAML, pero se puede cambiar por archivos INI, XML o cualquier otro formato que prefiera el programador.
- **Rápido:** la aplicación nunca procesa los archivos de configuración, sino que se encarga de ello el sistema de configuración, que compila todos los archivos de configuración en trozos de código PHP que se pueden procesar muy rápidamente.

5.1.1. Sintaxis YAML y convenciones de Symfony

Symfony utiliza por defecto el formato YAML para la configuración, en vez de los tradicionales formatos INI y XML. El formato YAML indica su estructura mediante la tabulación y es muy rápido de escribir. El Capítulo 1 ya describe algunas de sus ventajas y las reglas más básicas. No obstante, se deben tener presentes algunas convenciones al escribir archivos YAML. En esta sección se mencionan las convenciones o normas más importantes. El sitio web de YAML (<http://www.yaml.org/>) contiene la lista completa de normas del formato.

En primer lugar, **no se deben utilizar tabuladores en los archivos YAML**, sino que siempre se deben utilizar espacios en blanco. Los sistemas que procesan YAML no son capaces de tratar con los tabuladores, por lo que la tabulación de los archivos se debe crear con espacios en blanco como se muestra en el listado 5-1 (en YAML un tabulador se indica mediante 2 espacios en blanco seguidos).

Listado 5-1 - Los archivos YAML no permiten los tabuladores

```
# No utilices tabuladores
all:
  -> mail:
  -> -> webmaster:  webmaster@ejemplo.com

# Utiliza espacios en blanco
all:
  mail:
    webmaster: webmaster@ejemplo.com
```

Si los parámetros son cadenas de texto que contienen espacios en blanco al principio o al final, se debe encerrar la cadena entera entre comillas simples. Si la cadena de texto contiene *caracteres especiales*, también se encierran con comillas simples, como se muestra en el listado 5-2.

Listado 5-2 - Las cadenas de texto *especiales* deben encerrarse entre comillas simples

```
error1: Este campo es obligatorio
error2: ' Este campo es obligatorio '
```

```
# Las comillas simples que aparecen dentro de las cadenas de
# texto, se deben escribir dos veces
error3: 'Este [nowiki]''campo''[/nowiki] es obligatorio'
```

Se pueden escribir cadenas de texto muy largas en varias líneas, además de juntar cadenas escritas en varias líneas. En este último caso, se debe utilizar un carácter especial para indicar que se van a escribir varias líneas (se puede utilizar `>` o `|`) y se debe añadir una pequeña tabulación (dos espacios en blanco) a cada línea del grupo de cadenas de texto. El listado 5-3 muestra este caso.

Listado 5-3 - Definir cadenas de texto muy largas y cadenas de texto multi-línea

```
# Las cadenas de texto muy largas se pueden escribir en
# varias líneas utilizando el carácter >
# Posteriormente, cada nueva línea se transforma en un
# espacio en blanco para formar la cadena de texto original.
# De esta forma, el archivo YAML es más fácil de leer
frase_para_recordar: >
  Vive como si fueras a morir mañana y
  aprende como si fueras a vivir para siempre.

# Si un texto está formado por varias líneas, se utiliza
# el carácter | para separar cada nueva línea. Los espacios
# en blanco utilizados para tabular las líneas no se tienen
# en cuenta.
direccion: |
  Mi calle, número X
  Nombre de mi ciudad
  CP XXXXX
```

Los arrays se definen mediante corchetes que encierran a los elementos o mediante la sintaxis expandida que utiliza guiones medios para cada elemento del array, como se muestra en el listado 5-4.

Listado 5-4 - Sintaxis de YAML para incluir arrays

```
# Sintaxis abreviada para los arrays
idiomas: [ Alemán, Francés, Inglés, Italiano ]

# Sintaxis expandida para los arrays
idiomas:
  - Alemán
  - Francés
  - Inglés
  - Italiano
```

Para definir arrays asociativos, se deben encerrar los elementos mediante llaves (`{}` y `}`) y siempre se debe insertar un espacio en blanco entre la clave y el valor de cada par `clave: valor`. También existe una sintaxis expandida que requiere indicar cada par `clave: valor` en una nueva línea y con una tabulación (es decir, con 2 espacios en blanco delante) como se muestra en el listado 5-5.

Listado 5-5 - Sintaxis de YAML para incluir arrays asociativos

```
# Sintaxis incorrecta, falta un espacio después de los 2 puntos
mail: {webmaster:webmaster@ejemplo.com,contacto:contacto@ejemplo.com}

# Sintaxis abreviada correcta para los array asociativos
mail: { webmaster: webmaster@ejemplo.com, contacto: contacto@ejemplo.com }

# Sintaxis expandida para los arrays asociativos
mail:
  webmaster: webmaster@ejemplo.com
  contacto:  contacto@ejemplo.com
```

Para los parámetros booleanos, se pueden utilizar los valores on, 1 o true para los valores verdaderos y off, 0 o false para los valores falsos. El listado 5-6 muestra los posibles valores booleanos.

Listado 5-6 - Sintaxis de YAML para los valores booleanos

```
valores_verdaderos:  [ on, 1, true ]
valores_falsos:      [ off, 0, false ]
```

Es recomendable añadir comentarios (que se definen mediante el carácter #) y todos los espacios en blanco adicionales que hagan falta para hacer más fáciles de leer los archivos YAML, como se muestra en el listado 5-7.

Listado 5-7 - Comentarios en YAML y espacios adicionales para alinear valores

```
# Esta línea es un comentario
mail:
  webmaster: webmaster@ejemplo.com
  contacto:  contacto@ejemplo.com
  admin:     admin@ejemplo.com   # se añaden espacios en blanco para alinear los valores
```

En algunos archivos de configuración de Symfony, se ven líneas que empiezan por # (y por tanto se consideran comentarios y se ignoran) pero que parecen opciones de configuración correctas. Se trata de una de las convenciones de Symfony: la configuración por defecto, heredada de los archivos YAML del núcleo de Symfony, se repite en forma de líneas comentadas a lo largo de los archivos de configuración de cada aplicación, con el único objetivo de informar al desarrollador. De esta forma, para modificar esa opción de configuración, solamente es necesario eliminar el carácter de los comentarios y establecer su nuevo valor. El listado 5-8 muestra un ejemplo.

Listado 5-8 - La configuración por defecto se muestra en forma de comentarios

```
# Por defecto la cache está desactivada
settings:
# cache: off

# Para modificar esta opción, se debe descomentar la línea
settings:
  cache: on
```

En ocasiones, Symfony agrupa varias opciones de configuración en categorías. Todas las opciones que pertenecen a una categoría se muestran tabuladas y bajo el nombre de esa categoría. La configuración es más sencilla de leer si se agrupan las listas largas de pares clave:

valor. Los nombres de las categorías comienzan siempre con un punto (.) y el listado 5-19 muestra un ejemplo de uso de categorías.

Listado 5-9 - Los nombres de categorías son como los nombres de las clave, pero empiezan con un punto

```
all:
  .general:
    impuestos: 19.6

mail:
  webmaster: webmaster@ejemplo.com
```

En el ejemplo anterior, `mail` es una clave y `general` sólo es el nombre de la categoría. En realidad, el archivo YAML se procesa como si no existiera el nombre de la categoría, es decir, como se muestra en el listado 5-10. El parámetro `impuestos` realmente es descendiente directo de la clave `all`. No obstante, el uso de nombres de categorías facilita a Symfony el manejo de los arrays que se encuentran bajo la clave `all`.

Listado 5-10 - Los nombres de categorías solo se utilizan para hacer más fácil de leer los archivos YAML y la aplicación los ignora

```
all:
  impuestos: 19.6

mail:
  webmaster: webmaster@ejemplo.com
```

YAML solamente es una interfaz para definir las opciones que utiliza el código PHP, por lo que la configuración definida mediante YAML se transforma en código PHP. Si ya has accedido al menos una vez a la aplicación, comprueba la cache de su configuración (por ejemplo en `cache/frontend/dev/config/`). En ese directorio se encuentran los archivos PHP generados por la configuración YAML. Más adelante se detalla la cache de la configuración.

Lo mejor de todo es que si no quieres utilizar archivos YAML, puedes realizar la misma configuración a mano o mediante otros formatos (XML, INI, etc.) Más adelante en este libro se comentan otras formas alternativas a YAML para realizar la configuración e incluso se muestra como modificar las funciones de Symfony que se encargan de procesar la configuración (en el Capítulo 19). Utilizando estas técnicas, es posible evitar los archivos de configuración e incluso definir tu propio formato de configuración.

5.1.2. ¡Ay, Socorro, los archivos YAML han roto la aplicación!

Los archivos YAML se procesan y se transforman en array asociativos y arrays normales de PHP. Después, estos valores transformados son los que se utilizan en la aplicación para modificar el comportamiento de la vista, el modelo y el controlador. Por este motivo, cuando existe un error en un archivo YAML, normalmente no se detecta hasta que se utiliza ese valor específico. Para complicar las cosas, el error o la excepción que se muestra no siempre indica de forma clara que puede tratarse de un error en los archivos YAML de configuración.

Si la aplicación deja de funcionar después de un cambio en la configuración, se debe comprobar que no se ha cometido alguno de los errores típicos de los desarrolladores principiantes con YAML:

- No existe un espacio en blanco entre la clave y su valor:

```
| clave1:valor1      # Falta un espacio después del :
```

- Alguna clave de una secuencia de valores está mal tabulada:

```
| all:
|   clave1: valor1
|   clave2: valor2 # Su tabulación no es igual que los otros miembros de la secuencia
|   clave3: valor3
```

- Alguna clave o valor contiene caracteres reservados por YAML que no han sido encerrados por las comillas simples:

```
| mensaje: dile lo siguiente: hola    # :, [, ], { y } están reservados por YAML
| mensaje: 'dile lo siguiente: hola'  # Sintaxis correcta
```

- La línea que se modifica está comentada:

```
| # clave: valor      # No se tiene en cuenta porque empieza por #
```

- Existen 2 claves iguales con diferentes valores dentro del mismo nivel:

```
| clave1: valor1
| clave2: valor2
| clave1: valor3      # clave1 está definida 2 veces, solo se tiene en cuenta su último
| valor
```

- Todos los valores se consideran cadenas de texto, salvo que se convierta de forma explícita su valor:

```
| ingresos: 12,345    # El valor es una cadena de texto y no un número, salvo que se
| convierta
```

5.2. Un vistazo general a los archivos de configuración

La configuración de las aplicaciones realizadas con Symfony se distribuye en varios archivos según su propósito. Los archivos contienen definiciones de parámetros, normalmente llamadas opciones de configuración. Algunos parámetros se pueden redefinir en varios niveles de la aplicación web (proyecto, aplicación y módulo) y otros parámetros son exclusivos de algún nivel. En los siguientes capítulos se muestran los diversos archivos de configuración relacionados con el tema de cada capítulo. En el Capítulo 19 se explica la configuración avanzada.

5.2.1. Configuración del Proyecto

Symfony crea por defecto algunos archivos de configuración relacionados con el proyecto. El directorio `miproyecto/config/` contiene los siguientes archivos:

- `ProjectConfiguration.class.php`: se trata del primer archivo que se incluye con cada petición o comando. Contiene la ruta a los archivos del framework y se puede modificar

para realizar una instalación personalizada. El Capítulo 19 muestra el uso más avanzado de este archivo.

- `databases.yml`: contiene la definición de los accesos a bases de datos y las opciones de conexión de cada acceso (servidor, nombre de usuario, contraseña, nombre de base de datos, etc.) El Capítulo 8 lo explica en detalle. Sus parámetros se pueden redefinir en el nivel de la aplicación.
- `properties.ini`: contiene algunos parámetros que utiliza la herramienta de línea de comandos, como son el nombre del proyecto y las opciones para conectar con servidores remotos. El Capítulo 16 muestra las opciones de este archivo.
- `rsync_exclude.txt`: indica los directorios que se excluyen durante la sincronización entre servidores. El Capítulo 16 también incluye una explicación de este archivo.
- `schema.yml` y `propel.ini`: son los archivos de configuración que utiliza Propel para el acceso a los datos (recuerda que Propel es el sistema ORM que incorpora Symfony). Se utilizan para que las librerías de Propel puedan interactuar con las clases de Symfony y con los datos de la aplicación. `schema.yml` contiene la representación del modelo de datos relacional del proyecto. `propel.ini` se genera de forma automática y es muy probable que no necesites modificarlo. Si no se utiliza Propel, estos dos archivos son innecesarios. El Capítulo 8 explica en detalle el uso de estos dos archivos.

La mayoría de estos archivos los utilizan componentes externos o la línea de comandos e incluso algunos son procesados antes de que se inicie la herramienta que procesa archivos en formato YAML. Por este motivo, algunos de estos archivos no utilizan el formato YAML.

5.2.2. Configuración de la Aplicación

La configuración de la aplicación es la parte más importante de toda la configuración. La configuración se distribuye entre el controlador frontal (que se encuentra en el directorio `web/`) que contiene la configuración principal, el directorio `config/` de la aplicación que contiene diversos archivos en formato YAML, los archivos de internacionalización que se encuentran en el directorio `i18n/` y también existen otros archivos del framework que contienen opciones de configuración ocultas pero útiles para la configuración de la aplicación.

5.2.2.1. Configuración del Controlador Frontal

La primera configuración de la aplicación se encuentra en su controlador frontal, que es el primer script que se ejecuta con cada petición. El listado 5-11 muestra el código por defecto del controlador frontal generado automáticamente:

Listado 5-11 - El controlador frontal de producción generado automáticamente

```
<?php

require_once(dirname(__FILE__).'../config/ProjectConfiguration.class.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend', 'prod',
false);
sfContext::createInstance($configuration)->dispatch();
```

Después de definir el nombre de la aplicación (*frontend*) y el entorno en el que se ejecuta la aplicación (*prod*), se carga el archivo general de configuración, se crea un contexto y se despacha la petición (*dispatching*). En la clase de configuración de la aplicación se definen algunos métodos importantes:

- `$configuration->getRootDir()`: directorio raíz del proyecto (en general no hay que modificar su valor, salvo que se cambie la estructura de archivos del proyecto).
- `$configuration->getApplication()`: el nombre de la aplicación, que se utiliza para determinar las rutas de los archivos.
- `$configuration->getEnvironment()`: nombre del entorno en el que se ejecuta la aplicación (*prod*, *dev* o cualquier otro valor que se haya definido). Se utiliza para determinar las opciones de configuración que se utilizan. Al final de este capítulo se explican los entornos de ejecución.
- `$configuration->isDebug()`: activa el modo de depuración de la aplicación (el capítulo 16 contiene los detalles).

Cuando se quiere cambiar alguno de estos valores, normalmente se crea un nuevo controlador frontal. El siguiente capítulo explica su funcionamiento y cómo crear nuevos controladores.

5.2.2.2. Configuración principal de la aplicación

La configuración más importante de la aplicación se encuentra en el directorio `miproyecto/apps/frontend/config/`:

- `app.yml`: contiene la configuración específica de la aplicación; por ejemplo se pueden almacenar variables globales que se utilizan en la lógica de negocio de la aplicación y que no se almacenan en una base de datos. Los ejemplos habituales de estas variables son los porcentajes de los impuestos como el IVA, los gastos de envío, direcciones de email de contacto, etc. Por defecto el archivo está vacío.
- `frontendConfiguration.class.php`: esta clase inicia la ejecución de la aplicación, ya que realiza todas las inicializaciones necesarias para que la aplicación se pueda ejecutar. En este archivo se puede personalizar la estructura de directorios de la aplicación y se pueden añadir constantes que manejan las aplicaciones (el Capítulo 19 lo explica con más detalle). Esta clase hereda de la clase `ProjectConfiguration`.
- `factories.yml`: Symfony incluye sus propias clases para el manejo de la vista, de las peticiones, de las respuestas, de la sesión, etc. No obstante, es posible definir otras clases propias para realizar estas tareas. El Capítulo 17 lo explica más detalladamente.
- `filters.yml`: los filtros son trozos de código que se ejecutan con cada petición. En este archivo se definen los filtros que se van a procesar y cada módulo puede redefinir los filtros que se procesan. El Capítulo 6 explica en detalle los filtros.
- `routing.yml`: almacena las reglas de enrutamiento, que permiten transformar las URL habituales de las aplicaciones web en URL *limpias* y sencillas de recordar. Cada vez que se

crea una aplicación se crean unas cuantas reglas básicas por defecto. El Capítulo 9 está dedicado a los enlaces y el sistema de enrutamiento.

- `settings.yml`: contiene las principales opciones de configuración de una aplicación Symfony. Entre otras, permite especificar si la aplicación utiliza la internacionalización, el idioma por defecto de la aplicación, el tiempo de expiración de las peticiones y si se activa o no la cache. Un cambio en una única línea de configuración de este archivo permite detener el acceso a la aplicación para realizar tareas de mantenimiento o para actualizar alguno de sus componentes. Las opciones más habituales y su uso se describen en el Capítulo 19.
- `view.yml`: establece la estructura de la vista por defecto: el nombre del layout, las hojas de estilos CSS y los archivos JavaScript que se incluyen, el Content-Type, etc. El capítulo 7 explica detalladamente todas sus posibilidades. Cada módulo puede redefinir el valor de todas estas opciones.

5.2.2.3. Configuración de la Internacionalización

Las aplicaciones con soporte de internacionalización son las que pueden mostrar una misma página en diferentes idiomas. Para conseguirlo, es necesario realizar una configuración específica. Los dos sitios donde se configura la internacionalización en Symfony son:

- Archivo `factories.yml` del directorio `config/` de la aplicación: en este archivo se define la factoría encargada de la internacionalización y otras opciones comunes para la traducción de páginas, como por ejemplo el idioma por defecto, si las traducciones se guardan en archivos o bases de datos y su formato.
- Los archivos de traducción en el directorio `i18n/` de la aplicación: se trata de una especie de diccionarios que indican la traducción de cada frase que utilizan las plantillas de la aplicación de forma que cuando el usuario cambie de idioma sea posible mostrar las páginas en ese idioma.

Para activar el mecanismo de `i18n`, se debe modificar el archivo `settings.yml`. El Capítulo 13 profundiza en todas las características relacionadas con la `i18n`.

Nota Nota del traductor El término `i18n` es el más utilizado para referirse a la "internacionalización". Su origen proviene de las 18 letras que existen entre la letra "i" y la letra "n" en la palabra "internacionalización". Otras palabras siguen la misma técnica y así es habitual decir `l10n` para hablar de la "localization" o adaptación local de los contenidos.

5.2.2.4. Configuración adicional de la Aplicación

Algunos archivos de configuración se encuentran en el directorio de instalación de Symfony (en `$sf_symfony_lib_dir/config/config/`) y por tanto no aparecen en los directorios de configuración de las aplicaciones. Las opciones que se encuentran en esos archivos son opciones para las que raramente se modifica su valor o que son globales a todos los proyectos. De todas formas, si necesitas modificar alguna de estas opciones, crea un archivo vacío con el mismo nombre en el directorio `miproyecto/apps/frontend/config/` y redefine todas las opciones que quieras modificar. Las opciones definidas en una aplicación siempre tienen preferencia respecto

a las opciones definidas por el framework. Los archivos de configuración que se encuentran en el directorio `config/` de la instalación de Symfony son los siguientes:

- `autoload.yml`: contiene las opciones relativas a la carga automática de clases. Esta opción permite utilizar clases propias sin necesidad de incluirlas previamente en el script que las utiliza, siempre que esas clases se encuentren en algunos directorios determinados. El Capítulo 19 lo describe en detalle.
- `core_compile.yml` y `bootstrap_compile.yml`: define la lista de clases que se incluyen al iniciar la aplicación (en `bootstrap_compile.yml`) y las que se incluyen al procesar una petición (en `core_compile.yml`). Todas estas clases se concatenan en un único archivo PHP optimizado en el que se eliminan los comentarios y que acelera mucho la ejecución de la aplicación (ya que se reduce el número de archivos que se acceden a uno solo desde los más de 40 archivos necesarios originalmente para cada petición). Esta característica es muy necesaria cuando no se utiliza ningún acelerador de PHP. El Capítulo 18 incluye diversas técnicas para optimizar el rendimiento de las aplicaciones.
- `config_handlers.yml`: permite añadir o modificar los manejadores de archivos de configuración. El Capítulo 19 contiene todos los detalles.

5.2.3. Configuración de los Módulos

Inicialmente los módulos no tienen ninguna configuración propia. En cualquier caso, es posible modificar las opciones de la aplicación en cualquier módulo que así lo requiera. Un ejemplo de configuración específica es aquel que permite incluir un archivo JavaScript concreto en todas las acciones de un módulo. También se pueden añadir nuevos parámetros exclusivamente para un módulo concreto.

Como se puede suponer, los archivos de configuración de los módulos se encuentran en el directorio `miproyecto/apps/frontend/modules/mimodulo/config/`. Los archivos disponibles son los siguientes:

- `generator.yml`: se utiliza para los módulos generados automáticamente a partir de una tabla de la base de datos, es decir, para los módulos utilizados en el *scaffolding* y para las partes de administración creadas de forma automática. Contiene las opciones que definen como se muestran las filas y los registros en las páginas generadas y también define las interacciones con el usuario: filtros, ordenación, botones, etc. El Capítulo 14 explica todas estas características.
- `module.yml`: contiene la configuración de la acción y otros parámetros específicos del módulo (es un archivo equivalente al archivo `app.yml` de la aplicación). El Capítulo 6 lo explica en detalle.
- `security.yml`: permite restringir el acceso a determinadas acciones del módulo. En este archivo se configura que una página solamente pueda ser accedida por los usuarios registrados o por un grupo de usuarios registrados con permisos especiales. En el Capítulo 6 se detalla su funcionamiento.

- `view.yml`: permite configurar las vistas de una o de todas las acciones del módulo. Redefine las opciones del archivo `view.yml` de la aplicación y su funcionamiento se describe en el Capítulo 7.

Casi todos los archivos de configuración de los módulos permiten definir parámetros para todas las vistas y/o acciones del módulo o solo para una serie de vistas y/o acciones.

Seguramente estás un poco abrumado por la cantidad de archivos de configuración que tiene la aplicación. Pero debes tener en cuenta que:

Muy pocas veces tendrás que modificar la configuración, ya que las convenciones y normas establecidas por Symfony suelen coincidir con los requerimientos habituales de las aplicaciones. Cada archivo de configuración se utiliza para alguna característica concreta, que se detallarán una a una en los siguientes capítulos. Cuando se estudia individualmente uno de los archivos, es muy fácil comprender su estructura y su finalidad. Para las aplicaciones más profesionales, es habitual modificar la configuración por defecto. Los archivos de configuración permiten modificar fácilmente el funcionamiento de Symfony sin necesidad de añadir o modificar código a la aplicación. Imagina la cantidad de código PHP que se necesitaría para obtener un control similar al de Symfony. Si toda la configuración estuviera centralizada en un único archivo, además de ser un archivo enorme y muy difícil de leer, no sería posible redefinir el valor de las opciones en los diferentes niveles (como se ve más tarde en este capítulo en la sección "Configuraciones en cascada").

El mecanismo de configuración de Symfony es uno de sus puntos fuertes, ya que permite que el framework se pueda utilizar para crear cualquier tipo de aplicación y no solamente aquellas para las que se diseñó originalmente.

5.3. Entornos

Cuando se desarrolla una aplicación, es habitual disponer de varias configuraciones distintas pero relacionadas. Por ejemplo durante el desarrollo se tiene un archivo de configuración con los datos de conexión a la base de datos de pruebas, mientras que en el servidor de producción los datos de conexión necesarios son los de la base de datos de producción. Symfony permite definir diferentes entornos de ejecución para poder manejar de forma sencilla las diferentes configuraciones.

5.3.1. ¿Qué es un entorno?

Las aplicaciones de Symfony se pueden ejecutar en diferentes entornos. Todos los entornos comparten el mismo código PHP (salvo el código del controlador frontal) pero pueden tener configuraciones completamente diferentes. Cuando se crea una aplicación, Symfony crea por defecto 3 entornos: producción (prod), pruebas (test) y desarrollo (dev). También es posible añadir cualquier nuevo entorno que se considere necesario.

En cierta forma, un entorno y una configuración son sinónimos. Por ejemplo el entorno de pruebas registra las alertas y los errores en el archivo de log, mientras que el entorno de producción solamente registra los errores. En el entorno de desarrollo se suele desactivar la cache, pero está activa en los entornos de pruebas y de producción. Los entornos de pruebas y

desarrollo normalmente trabajan con una base de datos que contiene datos de prueba, mientras que el servidor de producción trabaja con la base de datos *buena*. En este caso, la configuración de la base de datos varía en los diferentes entornos. Todos los entornos pueden ejecutarse en una misma máquina, aunque en los servidores de producción normalmente solo se instala el entorno de producción.

El entorno de desarrollo tiene activadas las opciones de log y de depuración de aplicaciones, ya que es más importante el mantenimiento de la aplicación que su rendimiento. Sin embargo, en el entorno de producción se ajustan las opciones de configuración para obtener el máximo rendimiento, por lo que muchas características están desactivadas por defecto. Una buena regla general suele ser la de utilizar el entorno de desarrollo hasta que consideres que la funcionalidad de la aplicación en la que estás trabajando se encuentra terminada y después pasarse al entorno de producción para comprobar su rendimiento.

El entorno de pruebas varía respecto del de desarrollo y el de producción. La única forma de interactuar con este entorno es mediante la línea de comandos para realizar pruebas funcionales y ejecutar scripts. De esta forma, el entorno de pruebas es parecido al de producción, pero no se accede a través de un navegador. De todas formas, simula el uso de *cookies* y otros componentes específicos de HTTP.

Para ejecutar la aplicación en otro entorno, solamente es necesario cambiar de controlador frontal. Hasta ahora, en todos los ejemplos se accedía al entorno de desarrollo, ya que las URL utilizadas llamaban al controlador frontal de desarrollo:

```
| http://localhost/frontend_dev.php/mimodulo/index
```

Sin embargo, si se quiere acceder a la aplicación en el entorno de producción, es necesario modificar la URL para llamar al controlador frontal de producción:

```
| http://localhost/index.php/mimodulo/index
```

Si el servidor web tiene habilitado el `mod_rewrite`, es posible utilizar las reglas de reescritura de URL de Symfony, que se encuentran en `web/.htaccess`. Estas reglas definen que el controlador frontal de producción es el script que se ejecuta por defecto en las peticiones, por lo que se pueden utilizar URL como la siguiente:

```
| http://localhost/mimodulo/index
```

No se deben confundir los entornos con los servidores. En Symfony, un entorno diferente es en realidad una configuración diferente, que se corresponde con un controlador frontal determinado (que es el script que se encarga de procesar la petición). Sin embargo, un servidor diferente se corresponde con un nombre de dominio diferente en la dirección.

```
| http://localhost/frontend_dev.php/mimodulo/index  
  
| Servidor = localhost  
| Entorno = frontend_dev.php (es decir, entorno de desarrollo)
```

Normalmente, los desarrolladores programan y prueban sus aplicaciones en servidores de desarrollo, que no son accesibles desde Internet y donde se puede modificar cualquier configuración de PHP o del propio servidor. Cuando la aplicación se va a lanzar de forma pública,

se transfieren los archivos de la aplicación al servidor de producción y se permite el acceso a los usuarios.

Por tanto, en cada servidor existen varios entornos de ejecución. Se puede ejecutar por ejemplo la aplicación en el entorno de producción aunque el servidor sea el de desarrollo. No obstante, suele ser habitual que en el servidor de producción solamente estén disponibles los entornos de ejecución de producción, para evitar que los usuarios puedan ver la configuración del servidor o puedan comprometer la seguridad del sistema.

Para evitar la exposición accidental en producción de los controladores frontales que no son de producción, Symfony añade una sencilla comprobación de la IP desde la que se accede a estos controladores frontales, de forma que sólo se pueda acceder a ellos desde localhost. Si quieres acceder a ellos desde cualquier otra IP, puedes eliminar esta comprobación, pero es un riesgo que puede comprometer la seguridad de tu proyecto y que por tanto está completamente desaconsejado. De hecho, como el nombre más común de las aplicaciones es frontend, cualquier usuario malintencionado tratará de acceder a `frontend_dev.php` para obtener la mayor información posible sobre tu sistema.

Para crear un nuevo entorno de ejecución, no es necesario recurrir a la línea de comandos o crear nuevos directorios. Lo único que hay que hacer es crear un nuevo archivo de tipo controlador frontal (puedes copiar uno de los existentes) y modificar el nombre de su entorno de ejecución. Este nuevo entorno hereda todas las configuraciones por defecto del framework y todas las opciones comunes a todos los entornos. En el siguiente capítulo se detalla como realizar esta operación.

5.3.2. Configuration en cascada

Una misma opción de configuración puede estar definida más de una vez en archivos diferentes. De esta forma es posible por ejemplo definir que el tipo MIME de las páginas de la aplicación sea `text/html`, pero que las páginas creadas con el módulo que se encarga del RSS tengan un tipo MIME igual a `text/xml`. Symfony permite establecer el primer valor en `frontend/config/view.yml` y el segundo en `frontend/modules/rss/config/view.yml`. El sistema de configuración se encarga de que una opción establecida a nivel de módulo tenga más prioridad que la opción definida a nivel de aplicación.

De hecho, Symfony define varios niveles de configuración:

- Niveles de granularidad:
 - Configuración por defecto establecida por el framework
 - Configuración global del proyecto (en `miproyecto/config/`)
 - Configuración local de cada aplicación (en `miproyecto/apps/frontend/config/`)
 - Configuración local de cada módulo (en `miproyecto/apps/frontend/modules/mimodulo/config/`)
- Niveles de entornos de ejecución:
 - Específico para un solo entorno

- Para todos los entornos

Muchas de las opciones que se pueden establecer dependen del entorno de ejecución. Por este motivo, los archivos de configuración YAML están divididos por entornos, además de incluir una sección que se aplica a todos los entornos. De esta forma, un archivo de configuración típico de Symfony se parece al que se muestra en el listado 5-12.

Listado 5-12 - La estructura típica de los archivos de configuración de Symfony

```
# Opciones para el entorno de producción
prod:
  ...

# Opciones para el entorno de desarrollo
dev:
  ...

# Opciones para el entorno de pruebas
test:
  ...

# Opciones para un entorno creado a medida
mientorno:
  ...

# Opciones para todos los entornos
all:
  ...
```

Además de estas opciones, el propio framework define otros valores por defecto en archivos que no se encuentran en la estructura de directorios del proyecto, sino que se encuentran en el directorio `$sf_symfony_lib_dir/config/config/` de la instalación de Symfony. El listado 5-13 muestra la configuración por defecto de estos archivos. Todas las aplicaciones heredan estas opciones.

Listado 5-13 - La configuración por defecto, en `$sf_symfony_lib_dir/config/config/settings.yml`

```
# Opciones por defecto:
default:
  default_module:      default
  default_action:      index
  ...
```

Las opciones de estos archivos se incluyen como opciones comentadas en los archivos de configuración del proyecto, la aplicación y los módulos, como se muestra en el listado 5-14. De esta forma, se puede conocer el valor por defecto de algunas opciones y modificarlo si es necesario.

Listado 5-14 - La configuración por defecto, repetida en varios archivos para conocer fácilmente su valor, en `frontend/config/settings.yml`

```
#all:
# default_module:      default
```

```
# default_action:      index
#...
```

El resultado final es que una misma opción puede ser definida varias veces y el valor que se considera en cada momento se genera mediante la configuración en cascada. Una opción definida en un entorno de ejecución específico tiene más prioridad sobre la misma opción definida para todos los entornos, que también tiene preferencia sobre las opciones definidas por defecto. Las opciones definidas a nivel de módulo tienen preferencia sobre las mismas opciones definidas a nivel de aplicación, que a su vez tienen preferencia sobre las definidas a nivel de proyecto. Todas estas prioridades se resumen en la siguiente lista de prioridades, en el que el primer valor es el más prioritario de todos:

1. Módulo
2. Aplicación
3. Proyecto
4. Entorno específico
5. Todos los entornos
6. Opciones por defecto

5.4. La cache de configuración

Si cada nueva petición tuviera que procesar todos los archivos YAML de configuración y tuviera que aplicar la configuración en cascada, se produciría una gran penalización en el rendimiento de la aplicación. Symfony incluye un mecanismo de cache de configuración para aumentar la velocidad de respuesta de las peticiones.

Unas clases especiales, llamadas manejadores, procesan todos los archivos de configuración originales y los transforman en código PHP que se puede procesar de forma muy rápida. En el entorno de desarrollo se prima la interactividad y no el rendimiento, por lo que en cada petición se comprueba si se ha modificado la configuración. Como se procesan siempre los archivos modificados, cualquier cambio de un archivo YAML se refleja de forma inmediata. Sin embargo, en el entorno de producción solamente se procesa la configuración una vez durante la primera petición y se almacena en una cache el código PHP generado, para que lo utilicen las siguientes peticiones. El rendimiento en el entorno producción no se resiente, ya que las peticiones solamente ejecutan código PHP optimizado.

Si por ejemplo el archivo `app.yml` contiene lo siguiente:

```
all:                                # Opciones para todos los entornos
  mail:
    webmaster:                      webmaster@ejemplo.com
```

La carpeta `cache/` del proyecto contendrá un archivo llamado `config_app.yml.php` y con el siguiente contenido:

```
<?php
sfConfig::add(array(
  'app_mail_webmaster' => 'webmaster@ejemplo.com',
));
```

La consecuencia es que los archivos YAML raramente son procesados por el framework, ya que se utiliza la cache de la configuración siempre que sea posible. Sin embargo, en el entorno de desarrollo cada nueva petición obliga a Symfony a comparar las fechas de modificación de los archivos YAML y las de los archivos almacenados en la cache. Solamente se vuelven a procesar aquellos archivos que hayan sido modificados desde la petición anterior.

Este mecanismo supone una gran ventaja respecto de otros frameworks de PHP, en los que se compilan los archivos de configuración en cada petición, incluso en producción. Al contrario de lo que sucede con Java, PHP no define un contexto de ejecución común a todas las peticiones. En otros frameworks de PHP, se produce una pérdida de rendimiento importante al procesar toda la configuración con cada petición. Gracias al sistema de cache, Symfony no sufre esta penalización ya que la pérdida de rendimiento provocada por la configuración es muy baja.

La cache de la configuración implica una consecuencia muy importante. Si se modifica la configuración en el entorno de producción, se debe forzar a Symfony a que vuelva a procesar los archivos de configuración para que se tengan en cuenta los cambios. Para ello, solo es necesario borrar la cache, borrando todos los contenidos del directorio cache/ o utilizando una tarea específica proporcionada por Symfony:

```
| > php symfony cache:clear
```

5.5. Accediendo a la configuración desde la aplicación

Los archivos de configuración se transforman en código PHP y la mayoría de sus opciones solamente son utilizadas por el framework. Sin embargo, en ocasiones es necesario acceder a los archivos de configuración desde el código de la aplicación (en las acciones, plantillas, clases propias, etc.) Se puede acceder a todas las opciones definidas en los archivos `settings.yml`, `app.yml` y `module.yml` mediante una clase especial llamada `sfConfig`.

5.5.1. La clase `sfConfig`

Desde cualquier punto del código de la aplicación se puede acceder a las opciones de configuración mediante la clase `sfConfig`. Se trata de un registro de opciones de configuración que proporciona un método *getter* que puede ser utilizado en cualquier parte del código:

```
| // Obtiene una opción  
| opcion = sfConfig::get('nombre_opcion', $valor_por_defecto);
```

También se pueden crear o redefinir opciones desde el código de la aplicación:

```
| // Crear una nueva opción  
| sfConfig::set('nombre_opcion', $valor);
```

El nombre de la opción se construye concatenando varios elementos y separándolos con guiones bajos en este orden:

- Un prefijo relacionado con el nombre del archivo de configuración (`sf_` para `settings.yml`, `app_` para `app.yml`, `mod_` para `module.yml`)
- Si existen, todas las claves ascendentes de la opción (y en minúsculas)

- El nombre de la clave, en minúsculas

No es necesario incluir el nombre del entorno de ejecución, ya que el código PHP solo tiene acceso a los valores definidos para el entorno en el que se está ejecutando.

El listado 5-16 muestra el código necesario para acceder a los valores de las opciones definidas en el archivo `app.yml` mostrado en el listado 5-15.

Listado 5-15 - Ejemplo de configuración del archivo `app.yml`

```
all:
  .general:
    impuestos: 19.6
    usuario_por_defecto:
      nombre: Juan Pérez
      email:
        webmaster: webmaster@ejemplo.com
        contacto: contacto@ejemplo.com
  dev:
    email:
      webmaster: otro@ejemplo.com
      contacto: otro@ejemplo.com
```

Listado 5-16 - Acceso a las opciones de configuración desde el entorno de desarrollo

```
echo sfConfig::get('app_impuestos'); // Recuerda que se ignora el nombre de la
categoría                               // Es decir, no es necesario incluir 'general'

=> '19.6'
echo sfConfig::get('app_usuario_por_defecto_nombre');
=> 'Juan Pérez'
echo sfConfig::get('app_email_webmaster');
=> 'otro@ejemplo.com'
echo sfConfig::get('app_email_contacto');
=> 'otro@ejemplo.com'
```

Las opciones de configuración de Symfony tienen todas las ventajas de las constantes PHP, pero sin sus desventajas, ya que se puede modificar su valor durante la ejecución de la aplicación.

Considerando el funcionamiento que se ha mostrado, el archivo `settings.yml` que se utiliza para establecer las opciones del framework en cada aplicación, es equivalente a realizar varias llamadas a la función `sfConfig::set()`. Así que el listado 5-17 se interpreta de la misma forma que el listado 5-18.

Listado 5-17 - Extracto del archivo de configuración `settings.yml`

```
all:
  .settings:
    available: on
    path_info_array: SERVER
    path_info_key: PATH_INFO
    url_format: PATH
```

Listado 5-18 - Forma en la que Symfony procesa el archivo `settings.yml`

```
sfConfig::add(array(
    'sf_available' => true,
    'sf_path_info_array' => 'SERVER',
    'sf_path_info_key' => 'PATH_INFO',
    'sf_url_format' => 'PATH',
));
```

El Capítulo 19 explica el significado de las opciones de configuración del archivo `settings.yml`.

5.5.2. El archivo `app.yml` y la configuración propia de la aplicación

El archivo `app.yml`, que se encuentra en el directorio `miproyecto/apps/frontend/config/`, contiene la mayoría de las opciones de configuración relacionadas con la aplicación. Por defecto el archivo está vacío y sus opciones se configuran para cada entorno de ejecución. En este archivo se deben incluir todas las opciones que necesiten modificarse rápidamente y se utiliza la clase `sfConfig` para acceder a sus valores desde el código de la aplicación. El listado 5-19 muestra un ejemplo.

Listado 5-19 - Archivo `app.yml` que define los tipos de tarjeta de crédito aceptados en un sitio

```
all:
  tarjetascredito:
    falsa:      off
    visa:       on
    americanexpress: on

dev:
  tarjetascredito:
    falsa:      on
```

Para saber si las tarjetas de crédito *falsas* se aceptan en el entorno de ejecución de la aplicación, se debe utilizar la siguiente instrucción:

```
| sfConfig::get('app_tarjetascredito_falsa');
```

Nota Si quieres definir un array de elementos bajo la clave `all`, es necesario que utilices el nombre de una categoría, ya que de otro modo Symfony trata cada uno de los valores de forma independiente, tal y como se muestra en el ejemplo anterior.

```
all:
  .array:
    tarjetascredito:
      falsa:      off
      visa:       on
      americanexpress: on
print_r(sfConfig::get('app_tarjetascredito'));

Array(
    [falsa] => false
    [visa] => true
    [americanexpress] => true
)
```

Sugerencia Cuando vayas a definir una constante o una opción dentro de un script, piensa si no sería mejor incluir esa opción en el archivo `app.yml`. Se trata del lugar más apropiado para guardar todas las opciones de la configuración.

Los requerimientos de algunas aplicaciones complejas pueden dificultar el uso del archivo `app.yml`. En este caso, se puede almacenar la configuración en cualquier otro archivo, con el formato y la sintaxis que se prefiera y que sea procesado por un manejador realizado completamente a medida. El Capítulo 19 explica en detalle el funcionamiento de los manejadores de configuraciones.

5.6. Trucos para los archivos de configuración

Antes de empezar a crear los primeros archivos YAML, existen algunos trucos muy útiles que es conveniente aprender. Estos trucos permiten evitar la duplicidad de la configuración y permiten personalizar el formato YAML.

5.6.1. Uso de constantes en los archivos de configuración YAML

Algunas opciones de configuración dependen del valor de otras opciones. Para evitar escribir 2 veces el mismo valor, Symfony permite definir constantes dentro de los archivos YAML. Si el manejador de los archivos se encuentra con un nombre de opción todo en mayúsculas y encerrado entre los símbolos `%` y `%`, lo reemplaza por el valor que tenga en ese momento. El listado 5-20 muestra un ejemplo.

Listado 5-20 - Uso de constantes en los archivos YAML, ejemplo extraído del archivo `autoload.yml`

```
autoload:
  symfony:
    name:          symfony
    path:          %SF_SYMFONY_LIB_DIR%
    recursive:     on
    exclude:       [vendor]
```

El valor de la opción `path` es el que devuelve en ese momento la llamada a `sfConfig::get('sf_symfony_lib_dir')`. Si un archivo de configuración depende de otro archivo, es necesario que el archivo del que se depende sea procesado antes (en el código de Symfony se puede observar el orden en el que se procesan los archivos de configuración). El archivo `app.yml` es uno de los últimos que se procesan, por lo que sus opciones pueden depender de las opciones de otros archivos de configuración.

5.6.2. Uso de programación en los archivos de configuración

Puede ocurrir que los archivos de configuración dependan de parámetros externos (como por ejemplo una base de datos u otro archivo de configuración). Para poder procesar este tipo de casos, Symfony procesa los archivos de configuración como si fueran archivos de PHP antes de procesarlos como archivos de tipo YAML. De esta forma, como se muestra en el listado 5-21, es posible incluir código PHP dentro de un archivo YAML:

Listado 5-21 - Los archivos YAML puede contener código PHP

```
all:
  traduccion:
    formato: <?php echo (sfConfig::get('sf_i18n') == true ? 'xliff' : 'none')."\\n" ?>
```

El único inconveniente es que la configuración se procesa al principio de la ejecución de la petición del usuario, por lo que no están disponibles ninguno de los métodos y funciones de Symfony.

Además, como la instrucción `echo` no añade ningún retorno de carro por defecto, es necesario añadirlo explícitamente mediante `\\n` o mediante el uso del *helper* `echo\\n` para cumplir con el formato YAML:

```
all:
  traduccion:
    formato: <?php echo\\n sfConfig::get('sf_i18n') == true ? 'xliff' : 'none' ?>
```

Cuidado Recuerda que en el entorno de producción, se utiliza una cache para la configuración, por lo que los archivos de configuración solamente se procesan (y en este caso, se ejecuta su código PHP) una vez después de borrar la cache.

5.6.3. Utilizar tu propio archivo YAML

La clase `sfYaml` permite procesar de forma sencilla cualquier archivo en formato YAML. Se trata de un procesador (*parser*) de archivos YAML que los convierte en arrays asociativos de PHP. El listado 5-22 muestra un archivo YAML de ejemplo y el listado 5-23 muestra como transformarlo en código PHP:

Listado 5-22 - Archivo de prueba llamado prueba.yml

```
casa:
  familia:
    apellido:      García
    padres:        [Antonio, María]
    hijos:         [Jose, Manuel, Carmen]
  direccion:
    numero:        34
    calle:         Gran Vía
    ciudad:        Cualquiera
    codigopostal:  12345
```

Listado 5-23 - Uso de la clase sfYaml para transformar el archivo YAML en un array asociativo

```
$prueba = sfYaml::load('/ruta/a/prueba.yml');
print_r($prueba);

Array(
  [casa] => Array(
    [familia] => Array(
      [apellido] => García
      [padres] => Array(
        [0] => Antonio
        [1] => María
```



```
        )
        [hijos] => Array(
            [0] => Jose
            [1] => Manuel
            [2] => Carmen
        )
    )
    [direccion] => Array(
        [numero] => 34
        [calle] => Gran Vía
        [ciudad] => Cualquiera
        [codigopostal] => 12345
    )
)
)
```

5.7. Resumen

El sistema de configuración de Symfony utiliza el lenguaje YAML por ser muy sencillo y fácil de leer. Los desarrolladores cuentan con la posibilidad de definir varios entornos de ejecución y con la opción de utilizar la configuración en cascada, lo que ofrece una gran versatilidad a su trabajo. Las opciones de configuración se pueden acceder desde el código de la aplicación mediante el objeto `sfConfig`, sobre todo las opciones de configuración de la aplicación que se definen en el archivo `app.yml`.

Aunque Symfony cuenta con muchos archivos de configuración, su ventaja es que así es más adaptable. Además, recuerda que solo las aplicaciones que requieren de una configuración muy personalizada tienen que utilizar estos archivos de configuración.

Capítulo 6. El Controlador

En Symfony, la capa del controlador, que contiene el código que liga la lógica de negocio con la presentación, está dividida en varios componentes que se utilizan para diversos propósitos:

- El controlador frontal es el único punto de entrada a la aplicación. Carga la configuración y determina la acción a ejecutarse.
- Las acciones contienen la lógica de la aplicación. Verifican la integridad de las peticiones y preparan los datos requeridos por la capa de presentación.
- Los objetos `request`, `response` y `session` dan acceso a los parámetros de la petición, las cabeceras de las respuestas y a los datos persistentes del usuario. Se utilizan muy a menudo en la capa del controlador.
- Los filtros son trozos de código ejecutados para cada petición, antes o después de una acción. Por ejemplo, los filtros de seguridad y validación son comúnmente utilizados en aplicaciones web. Puedes extender el framework creando tus propios filtros.

Este capítulo describe todos estos componentes, pero no te abrumes porque sean muchos componentes. Para una página básica, es probable que solo necesites escribir algunas líneas de código en la clase de la acción, y eso es todo. Los otros componentes del controlador solamente se utilizan en situaciones específicas.

6.1. El Controlador Frontal

Todas las peticiones web son manejadas por un solo controlador frontal, que es el punto de entrada único de toda la aplicación en un entorno determinado.

Cuando el controlador frontal recibe una petición, utiliza el sistema de enrutamiento para asociar el nombre de una acción y el nombre de un módulo con la URL escrita (o pinchada) por el usuario. Por ejemplo, la siguientes URL llama al script `index.php` (que es el controlador frontal) y será entendido como llamada a la acción `miAccion` del módulo `mimodulo`:

```
| http://localhost/index.php/mimodulo/miAccion
```

Si no estás interesado en los mecanismos internos de Symfony, eso es todo que necesitas saber sobre el controlador frontal. Es un componente imprescindible de la arquitectura MVC de Symfony, pero raramente necesitarás cambiarlo. Si no quieres conocer *las tripas* del controlador frontal, puedes saltarte el resto de esta sección.

6.1.1. El Trabajo del Controlador Frontal en Detalle

El controlador frontal se encarga de despachar las peticiones, lo que implica algo más que detectar la acción que se ejecuta. De hecho, ejecuta el código común a todas las acciones, incluyendo:

1. Carga la clase de configuración del proyecto y las librerías de Symfony.
2. Crea la configuración de la aplicación y el contexto de Symfony.

3. Carga e inicializa las clases del núcleo del framework.
4. Carga la configuración.
5. Decodifica la URL de la petición para determinar la acción a ejecutar y los parámetros de la petición.
6. Si la acción no existe, redireccionará a la acción del error 404.
7. Activa los filtros (por ejemplo, si la petición necesita autenticación).
8. Ejecuta los filtros, primera pasada.
9. Ejecuta la acción y produce la vista.
10. Ejecuta los filtros, segunda pasada.
11. Muestra la respuesta.

6.1.2. El Controlador Frontal por defecto

El controlador frontal por defecto, llamado `index.php` y ubicado en el directorio `web/` del proyecto, es un simple script, como lo muestra el Listado 6-1.

Listado 6-1 - El Controlador Frontal por Omisión

```
<?php

require_once(dirname(__FILE__).'../config/ProjectConfiguration.class.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend', 'prod',
false);
sfContext::createInstance($configuration)->dispatch();
```

El controlador frontal incluye la configuración de la aplicación, lo que corresponde a los puntos 2, 3 y 4 anteriores. La llamada al método `dispatch()` del objeto `sfController` (que es el controlador principal de la arquitectura MVC de Symfony) despacha la petición, lo que corresponde a los puntos 5, 6 y 7 anteriores. El resto de tareas las realiza la cadena de filtros, tal y como se explica más adelante en este capítulo.

6.1.3. Llamando a Otro Controlador Frontal para Cambiar el Entorno

Cada entorno dispone de un controlador frontal. De hecho, es la existencia del controlador frontal lo que define un entorno. El entorno se define mediante el segundo argumento que se pasa en la llamada al método `ProjectConfiguration::getApplicationConfiguration()`.

Para cambiar el entorno en el que se está viendo la aplicación, simplemente se elige otro controlador frontal. Los controladores frontales disponibles cuando creas una aplicación con la tarea `generate:app` son `index.php` para el entorno de producción y `frontend_dev.php` para el entorno de desarrollo (suponiendo que tu aplicación se llame `frontend`). La configuración por defecto de `mod_rewrite` utiliza `index.php` cuando la URL no contiene el nombre de un script correspondiente a un controlador frontal. Así que estas dos URL muestran la misma página (`mimodulo/index`) en el entorno de producción:

```
http://localhost/index.php/mimodulo/index
http://localhost/mimodulo/index
```

y esta URL muestra la misma página en el entorno de desarrollo:

```
| http://localhost/frontend_dev.php/mimodulo/index
```

Crear un nuevo entorno es tan fácil como crear un nuevo controlador frontal. Por ejemplo, puede ser necesario un entorno llamado *staging* que permita a tus clientes probar la aplicación antes de ir a producción. Para crear el entorno *staging*, simplemente copia `web/frontend_dev.php` en `web/frontend_staging.php` y cambia el valor del segundo argumento de `ProjectConfiguration::getApplicationConfiguration()` a *staging*. Ahora en todos los archivos de configuración, puedes añadir una nueva sección *staging*: para establecer los valores específicos para este entorno, como se muestra en el Listado 6-2

Listado 6-2 - Ejemplo de `app.yml` con valores específicos para el entorno *staging*

```
staging:
  mail:
    webmaster: falso@misitio.com
    contacto: falso@misitio.com
all:
  mail:
    webmaster: webmaster@misitio.com
    contacto: contacto@mysite.com
```

Si quieres ver cómo se comporta la aplicación en el nuevo entorno, utiliza el nuevo controlador frontal:

```
| http://localhost/frontend_staging.php/mimodulo/index
```

6.2. Acciones

Las acciones son el corazón de la aplicación, puesto que contienen toda la lógica de la aplicación. Las acciones utilizan el modelo y definen variables para la vista. Cuando se realiza una petición web en una aplicación Symfony, la URL define una acción y los parámetros de la petición.

6.2.1. La clase de la acción

Las acciones son métodos con el nombre `executeNombreAccion` de una clase llamada `nombreModuloActions` que hereda de la clase `sfActions` y se encuentran agrupadas por módulos. La clase que representa las acciones de un módulo se encuentra en el archivo `actions.class.php`, en el directorio `actions/` del módulo.

El listado 6-3 muestra un ejemplo de un archivo `actions.class.php` con una única acción `index` para todo el módulo `mimodulo`.

Listado 6-3 - Ejemplo de la clase de la acción, en `app/frontend/modules/mimodulo/actions/actions.class.php`

```
class mimoduloActions extends sfActions
{
  public function executeIndex()
  {
    // ...
  }
}
```

Cuidado Aunque en PHP no se distinguen las mayúsculas y minúsculas de los nombres de los métodos, Symfony si los distingue. Así que se debe tener presente que los métodos de las acciones deben comenzar con `execute` en minúscula, seguido por el nombre exacto de la acción con la primera letra en mayúscula.

Para ejecutar un acción, se debe llamar al script del controlador frontal con el nombre del módulo y de la acción como parámetros. Por defecto, se añade `nombre_modulo/nombre_accion` al script. Esto significa que la acción del listado 6-3 se puede ejecutar llamándola con la siguiente URL:

```
| http://localhost/index.php/mimodulo/index
```

Añadir más acciones simplemente significa agregar más métodos `execute` al objeto `sfActions`, como se muestra en el listado 6-4.

Listado 6-4 - Clase con dos acciones, en `frontend/modules/mimodulo/actions/actions.class.php`

```
| class mimoduloActions extends sfActions
| {
|     public function executeIndex()
|     {
|         // ...
|     }
|
|     public function executeListar()
|     {
|         // ...
|     }
| }
```

Si el tamaño de la clase de la acción crece demasiado, probablemente tendrás que refactorizar la clase para mover algo de código a la capa del modelo. El código de las acciones debería ser muy corto (no más que unas pocas líneas), y toda la lógica del negocio debería encontrarse en el modelo.

Aun así, el número de acciones en un módulo puede llegar a ser tan importante que sea necesario dividir las en 2 módulos.

En los ejemplos de código dados en este libro, probablemente has notado que la apertura y cierre de llaves (`{` y `}`) ocupan una línea cada una. Este estándar hace al código más fácil de leer.

Entre otras normas que sigue el código de Symfony, la indentación se realiza siempre con 2 espacios en blanco; nunca se utilizan los tabuladores. La razón es que los tabuladores se muestran con distinta anchura en función del editor de textos utilizado, y porque el código que mezcla tabuladores con espacios en blanco es bastante difícil de leer.

Los archivos PHP del núcleo de Symfony y los archivos generados no terminan con la etiqueta de cierre habitual `?>`. La razón es que esta etiqueta no es obligatoria y puede provocar problemas con la salida producida si se incluyen por error espacios en blanco después de la etiqueta de cierre.

Y si eres de los que te fijas en los detalles, verás que ninguna línea de código de Symfony termina con un espacio en blanco. En esta ocasión la razón no es técnica, sino que simplemente las líneas de código que terminan con espacios en blancos se ven feas en el editor de texto de Fabien.

6.2.2. Sintaxis alternativa para las clases de las Acciones

Se puede utilizar una sintaxis alternativa para distribuir las acciones en archivos separados, un archivo por acción. En este caso, cada clase acción extiende `sfAction` (en lugar de `sfActions`) y su nombre es `nombreAccionAction`. El nombre del método es simplemente `execute`. El nombre del archivo es el mismo que el de la clase. Esto significa que el equivalente del Listado 6-4 puede ser escrito en dos archivos mostrados en los listados 6-5 y 6-6.

Listado 6-5 - Archivo de una sola acción, en `frontend/modules/mimodulo/actions/indexAction.class.php`

```
class indexAction extends sfAction
{
    public function execute($peticion)
    {
        // ...
    }
}
```

Listado 6-6 - Archivo de una sola acción, en `frontend/modules/mimodulo/actions/listAction.class.php`

```
class listarAction extends sfAction
{
    public function execute($peticion)
    {
        // ...
    }
}
```

6.2.3. Obteniendo Información en las Acciones

Las clases de las acciones ofrecen un método para acceder a la información relacionada con el controlador y los objetos del núcleo de Symfony. El listado 6-7 muestra como utilizarlos.

Listado 6-7 - Métodos comunes de `sfActions`

```
class mimoduloActions extends sfActions
{
    public function executeIndex($peticion)
    {
        // Obteniendo parametros de la petición
        $password = $peticion->getParameter('password');

        // Obteniendo información del controlador
        $nombreModulo = $this->getModuleName();
        $nombreAccion = $this->getActionName();

        // Obteniendo objetos del núcleo del framework
    }
}
```

```

    $sesionUsuario = $this->getUser();
    $respuesta      = $this->getResponse();
    $controlador    = $this->getController();
    $contexto       = $this->getContext();

    // Creando variables de la acción para pasar información a la plantilla
    $this->setVar('parametro', 'valor');
    $this->parametro = 'valor';           // Versión corta.
}

```

En el controlador frontal ya se ha visto una llamada a `sfContext::getInstance()`. En una acción, el método `getContext()` devuelve el mismo *singleton*. Se trata de un objeto muy útil que guarda una referencia a todos los objetos del núcleo de Symfony relacionados con una petición dada, y ofrece un método accesor para cada uno de ellos:

- `sfController`: El objeto controlador (`->getController()`)
- `sfRequest`: El objeto de la petición (`->getRequest()`)
- `sfResponse`: El objeto de la respuesta (`->getResponse()`)
- `sfUser`: El objeto de la sesión del usuario (`->getUser()`)
- `sfDatabaseConnection`: La conexión a la base de datos (`->getDatabaseConnection()`)
- `sfLogger`: El objeto para los logs (`->getLogger()`)
- `sfI18N`: El objeto de internacionalización (`->getI18N()`)

Se puede llamar al singleton `sfContext::getInstance()` desde cualquier parte del código.

6.2.4. Terminación de las Acciones

Existen varias alternativas posibles cuando se termina la ejecución de una acción. El valor retornado por el método de la acción determina como será producida la vista. Para especificar la plantilla que se utiliza al mostrar el resultado de la acción, se emplean las constantes de la clase `sfView`.

Si existe una vista por defecto que se debe llamar (este es el caso más común), la acción debería terminar de la siguiente manera:

```
| return sfView::SUCCESS;
```

Symfony buscará entonces una plantilla llamada `nombreAccionSuccess.php`. Este comportamiento se ha definido como el comportamiento por defecto, por lo que si omites la sentencia `return` en el método de la acción, Symfony también buscará una plantilla llamada `nombreAccionSuccess.php`. Las acciones vacías también siguen este comportamiento. El listado 6-8 muestra un ejemplo de terminaciones exitosas de acciones.

Listado 6-8 - Acciones que llaman a las plantillas `indexSuccess.php` y `listarSuccess.php`

```

| public function executeIndex()
| {
|     return sfView::SUCCESS;

```

```

    }

    public function executeListar()
    {
    }

```

Si existe una vista de error que se debe llamar, la acción deberá terminar de la siguiente manera:

```
| return sfView::ERROR;
```

Symfony entonces buscará una plantilla llamada `nombreAccionError.php`.

Para utilizar una vista personalizada, se debe utilizar el siguiente valor de retorno:

```
| return 'MiResultado';
```

Symfony entonces buscará una plantilla llamada `nombreAccionMiResultado.php`.

Si no se utiliza ninguna vista --por ejemplo, en el caso de una acción ejecutada en un archivo de lotes-- la acción debe terminar de la siguiente forma:

```
| return sfView::NONE;
```

En este caso, no se ejecuta ninguna plantilla. De esta forma, se evita por completo la capa de vista y se establece directamente el código HTML producido por la acción. Como muestra el Listado 6-9, Symfony provee un método `renderText()` específico para este caso. Este método puede ser útil cuando se necesita una respuesta muy rápida en una acción, como por ejemplo para las interacciones creadas con Ajax, como se verá en el Capítulo 11.

Listado 6-9 - Evitando la vista mediante una respuesta directa y un valor de retorno `sfView::NONE`

```

public function executeIndex()
{
    $this->getResponse()->setContent("<html><body>␣Hola Mundo!</body></html>");

    return sfView::NONE;
}

// Es equivalente a
public function executeIndex()
{
    return $this->renderText("<html><body>␣Hola Mundo!</body></html>");
}

```

En algunos casos, se necesita una respuesta vacía pero con algunas cabeceras definidas (sobre todo la cabecera X-JSON). Para conseguirlo, se definen las cabeceras con el objeto `sfResponse`, que se ve en el próximo capítulo, y se devuelve como valor de retorno la constante `sfView::HEADER_ONLY`, como muestra el Listado 6-10.

Listado 6-10 - Evitando la producción de la vista y enviando solo cabeceras

```

public function executeActualizar()
{
    $salida = '<"titulo","Mi carta sencilla",["nombre","Sr. Pérez"]>';
    $this->getResponse()->setHttpHeader("X-JSON", '('.$salida.')');
}

```



```
|  
|  
|     return sfView::HEADER_ONLY;  
| }  
|
```

Si la acción debe ser producida por una plantilla específica, se debe prescindir de la sentencia `return` y se debe utilizar el método `setTemplate()` en su lugar.

```
| $this->setTemplate('miPlantillaPersonalizada');
```

6.2.5. Saltando a Otra Acción

En algunos casos, la ejecución de un acción termina solicitando la ejecución de otra acción. Por ejemplo, una acción que maneja el envío de un formulario en una solicitud POST normalmente redirecciona a otra acción después de actualizar la base de datos. Otro ejemplo es el de crear un alias de una acción: la acción `index` normalmente se utiliza para mostrar un listado y de hecho se suele redireccionar a la acción `list`.

La clase de la acción provee dos métodos para ejecutar otra acción:

- Si la acción pasa la llamada hacia otra acción (*forward*):

```
| $this->forward('otroModulo', 'index');
```

- Si la acción produce un redireccionamiento web (*redirect*):

```
| $this->redirect('otroModulo/index');  
| $this->redirect('http://www.google.com/');
```

Nota El código que se encuentra después de una llamada a los métodos `forward` o `redirect` en una acción nunca se ejecuta. Se puede considerar que estas llamadas son equivalentes a la sentencia `return`. Estos métodos lanzan una excepción `sfStopException` para detener la ejecución de la acción; esta excepción es interceptada más adelante por Symfony y simplemente se ignora.

La elección entre `redirect` y `forward` es a veces engañosa. Para elegir la mejor solución, ten en cuenta que un `forward` es una llamada interna a la aplicación y transparente para el usuario. En lo que concierne al usuario, la URL mostrada es la misma que la solicitada. Por el contrario, un `redirect` resulta en un mensaje al navegador del usuario, involucrando una nueva petición por parte del mismo y un cambio en la URL final resultante.

Si la acción es llamada desde un formulario enviado con `method="post"`, deberías siempre realizar un `redirect`. La principal ventaja es que si el usuario recarga la página resultante, el formulario no será enviado nuevamente; además, el botón de retroceder funciona como se espera, ya que muestra el formulario y no una alerta preguntando al usuario si desea reenviar una petición POST.

Existe un tipo especial de `forward` que se utiliza comúnmente. El método `forward404()` redirecciona a una acción de Página no encontrada. Este método se utiliza normalmente cuando un parámetro necesario para la ejecución de la acción no está presente en la petición (por tanto detectando una URL mal escrita). El Listado 6-11 muestra un ejemplo de una acción mostrar que espera un parámetro llamado `id`.

Listado 6-11 - Uso del método `forward404()`

```
public function executeVer($peticion)
{
    $articulo = ArticuloPeer::retrieveByPK($peticion->getParameter('id'));
    if (!$articulo)
    {
        $this->forward404();
    }
}
```

Sugerencia Si estás buscando la acción y la plantilla del error 404, las puedes encontrar en el directorio `$sf_symfony_lib_dir/controller/default/`. Se puede personalizar esta página agregado un módulo `default` a la aplicación, sobrescribiendo el del framework, y definiendo una acción `error404` y una plantilla `error404Success` dentro del nuevo módulo. Otro método alternativo es el de establecer las constantes `error_404_module` y `error_404_action` en el archivo `settings.yml` para utilizar una acción existente.

La experiencia muestra que, la mayoría de las veces, una acción hace un `redirect` o un `forward` después de probar algo, como en el listado 6-12. Por este motivo, la clase `sfActions` tiene algunos métodos más, llamados `forwardIf()`, `forwardUnless()`, `forward404If()`, `forward404Unless()`, `redirectIf()` y `redirectUnless()`. Estos métodos simplemente requieren un parámetro que representa la condición cuyo resultado se emplea para ejecutar el método. El método se ejecuta si el resultado de la condición es `true` y el método es de tipo `xxxIf()` o si el resultado de la condición es `false` y el método es de tipo `xxxUnless()`, como se muestra en el listado 6-12.

Listado 6-12 - Uso del método `forward404If()`

```
// Esta acción es equivalente a la mostrada en el Listado 6-12
public function executeVer($peticion)
{
    $articulo = ArticuloPeer::retrieveByPK($peticion->getParameter('id'));
    $this->forward404If(!$articulo);
}

// Esta acción también es equivalente
public function executeVer()
{
    $articulo = ArticuloPeer::retrieveByPK($peticion->getParameter('id'));
    $this->forward404Unless($articulo);
}
```

El uso de estos métodos permite mantener el código de las acciones muy corto y también lo hacen más fácil de leer.

Sugerencia Cuando la acción llama al método `forward404()` o alguno de sus similares, Symfony lanza una excepción `sfError404Exception` que maneja la respuesta al error 404. Esto significa que si se quiere mostrar un mensaje de error de tipo 404 desde cualquier parte del código desde donde no se quiere acceder al controlador, se puede lanzar una excepción similar.

6.2.6. Repitiendo Código para varias Acciones de un Módulo

La convención en el nombre de las acciones `executeNombreAccion()` (en el caso de una clase de tipo `sfActions`) o `execute()` (en el caso de una clase `sfAction`) garantiza que Symfony encontrará el método de la acción. Además, permite crear métodos propios que no serán considerados como acciones, siempre que su nombre no empiece con `execute`.

Existe otra convención útil cuando se necesita ejecutar repetidamente en cada acción una serie de sentencias antes de ejecutar la propia acción. Esas sentencias comunes se pueden colocar en el método `preExecute()` de la clase de la acción. De forma análoga, se pueden definir sentencias que se ejecuten después de cada acción añadiéndolas al método `postExecute()`. La sintaxis de estos métodos se muestra en el Listado 6-13.

Listado 6-13 - Usando los métodos `preExecute()`, `postExecute()` y otros métodos propios en la clase de la acción

```
class mimoduloActions extends sfActions
{
    public function preExecute()
    {
        // El código insertado aquí se ejecuta al principio de cada llamada a una acción
        // ...
    }

    public function executeIndex($peticion)
    {
        // ...
    }

    public function executeListar($peticion)
    {
        // ...
        $this->miPropioMetodo(); // Se puede acceder a cualquier método de la clase acción
    }

    public function postExecute()
    {
        // El código insertado aquí se ejecuta al final de cada llamada a la acción
        ...
    }

    protected function miPropioMetodo()
    {
        // Se pueden crear métodos propios, siempre que su nombre no comience por "execute"
        // En ese case, es mejor declarar los métodos como protected o private
        // ...
    }
}
```

6.3. Accediendo a la petición

El primer argumento de cualquier método que ejecuta una acción es el objeto que representa a la petición, llamada `sfWebRequest` en Symfony. De todos sus métodos, ya se ha mostrado el método `getParameter('miparametro')` para obtener el valor de un parámetro de la petición a partir de su nombre. La tabla 6-1 resume algunos de los métodos más útiles de `sfWebRequest`.

Tabla 6-1. Métodos del objeto `sfWebRequest`

Nombre	Función	Ejemplo de salida producida
Información sobre la petición		
isMethod(\$metodo)	Permite descubrir si el método empleado es POST o GET	Devuelve true o false
getMethod()	Método de la petición	Devuelve la constante sfRequest::GET o sfRequest::POST
getMethodName()	Nombre del método de petición	POST
getHttpHeader('Server')	Valor de una cabecera HTTP	Apache/2.0.59 (Unix) DAV/2 PHP/5.1.6
getCookie('foo')	Valor de una cookie	valor
isXmlHttpRequest() (1)	¿Es una petición AJAX?	true
isSecure()	¿Es una petición SSL?	true
Parámetros de la petición		
hasParameter('parametro')	¿Existe el parámetro en la petición?	true
getParameter('parametro')	Valor del parámetro	valor
getParameterHolder()->getAll()	Array de todos los parámetros de la petición	
Información relacionada con la URI		
getUri()	URI completa	http://localhost/frontend_dev.php/mimodulo/miaccion
getPathInfo()	Información de la ruta	/mimodulo/miaccion
getReferer() (2)	Valor del "referer" de la petición	http://localhost/frontend_dev.php/
getHost()	Nombre del Host	localhost
getScriptName()	Nombre y ruta del controlador frontal	frontend_dev.php
Información del navegador del cliente		

<code>getLanguages()</code>	Array de los lenguajes aceptados	Array([0] => fr [1] => fr_FR [2] => en_US [3] => en)
<code>getCharsets()</code>	Array de los juegos de caracteres aceptados	Array([0] => ISO-8859-1 [1] => UTF-8 [2] => *)
<code>getAcceptableContentType()</code>	Array de los tipos de contenidos aceptados	Array([0] => text/xml [1] => text/html)

(1) Funciona con Prototype, Mootools y jQuery (2) Si se utilizan proxys, su valor puede ser inaccesible

Para peticiones de tipo *multipart* utilizadas cuando el usuario adjunta archivos, el objeto `sfWebRequest` provee métodos para acceder y mover estos archivos, como se muestra en el listado 6-14. Sin embargo, estos métodos han sido declarados como obsoletos en Symfony 1.1 (puedes ver más información en la sección de formularios y en la clase `sfValidatorFile`).

Listado 6-14 - El objeto `sfWebRequest` sabe cómo manejar archivos adjuntos

```
class mimoduloActions extends sfActions
{
    public function executeSubirArchivos($peticion)
    {
        if ($peticion->hasFiles())
        {
            foreach ($peticion->getFileNames() as $archivoSubido)
            {
                $nombreArchivo      = $peticion->getFileName($archivoSubido);
                $tamanoArchivo       = $peticion->getFileSize($archivoSubido);
                $tipoArchivo         = $peticion->getFileType($archivoSubido);
                $archivoErroneo      = $peticion->hasFileError($archivoSubido);
                $directorioSubidas = sfConfig::get('sf_upload_dir');
                $peticion->moveFile($archivoSubido, $directorioSubidas.'/'.$nombreArchivo);
            }
        }
    }
}
```

No tienes que preocuparte sobre si el servidor soporta las variables de PHP `$_SERVER` o `$_ENV`, o acerca de valores por defecto o problemas de compatibilidad del servidor, ya que los métodos de `sfWebRequest` lo hacen todo por tí. Además sus nombres son tan evidentes que no es necesario consultar la documentación de PHP para descubrir cómo obtener información sobre la petición.

Nota El código del ejemplo anterior utiliza como nombre del archivo el mismo nombre del archivo subido por el usuario. Existe la posibilidad de que un usuario malintencionado envíe un archivo con un nombre especialmente preparado para aprovechar algún agujero de seguridad, por lo que es recomendable que generes de forma automática y/o normalices el nombre de todos los archivos subidos.

6.4. Sesiones de Usuario

Symfony maneja automáticamente las sesiones del usuario y es capaz de almacenar datos de forma persistente entre peticiones. Utiliza el mecanismo de manejo de sesiones incluido en PHP y lo mejora para hacerlo mas configurable y más fácil de usar.

6.4.1. Accediendo a la Sesión de Usuario

El objeto sesión del usuario actual se accede en la acción con el método `getUser()`, que es una instancia de la clase `sfUser`. Esta clase dispone de un contenedor de parámetros que permite guardar cualquier atributo del usuario en el. Esta información estará disponible en otras peticiones hasta terminar la sesión del usuario, como se muestra en el Listado 6-15. Los atributos de usuarios pueden guardar cualquier tipo de información (cadenas de texto, arrays y arrays asociativos). Se pueden utilizar para cualquier usuario, incluso si ese usuario no se ha identificado.

Listado 6-15 - El objeto `sfUser` puede contener atributos personalizados del usuario disponibles en todas las peticiones

```
class mimoduloActions extends sfActions
{
    public function executePrimeraPagina($peticion)
    {
        $nombre = $peticion->getParameter('nombre');

        // Guardar información en la sesión del usuario
        $this->getUser()->setAttribute('nombre', $nombre);
    }

    public function executeSegundaPagina()
    {
        // Obtener información de la sesión del usuario con un valor por defecto
        $nombre = $this->getUser()->getAttribute('nombre', 'Anónimo');
    }
}
```

Cuidado Puedes guardar objetos en la sesión del usuario, pero no se recomienda hacerlo. El motivo es que el objeto de la sesión se serializa entre una petición y otra. Cuando la sesión se *deserializa*, la clase del objeto guardado debe haber sido previamente cargada y este no es siempre el caso. Además, puede haber objetos de tipo *"stalled"* si se guardan objetos de Propel.

Como muchos otros *getters* en Symfony, el método `getAttribute()` acepta un segundo parámetro, especificando el valor por defecto a ser utilizado cuando el atributo no está definido. Para verificar si un atributo ha sido definido para un usuario, se utiliza el método `hasAttribute()`. Los atributos se guardan en un contenedor de parámetros que puede ser accedido por el método `getAttributeHolder()`. También permite un borrado rápido de los atributos del usuario con los métodos usuales del contenedor de parámetros, como se muestra en el listado 6-16.

Listado 6-16 - Eliminando información de la sesión del usuario

```
class mimoduloActions extends sfActions
{
    public function executeBorraNombre()
    {
        $this->getUser()->getAttributeHolder()->remove('nombre');
    }

    public function executeLimpia()
    {
        $this->getUser()->getAttributeHolder()->clear();
    }
}
```

Los atributos de la sesión del usuario también están disponibles por defecto en las plantillas mediante la variable `$sf_user`, que almacena el objeto `sfUser` actual, como se muestra en el listado 6-17.

Listado 6-17 - Las plantillas también tienen acceso a los atributos de la sesión del usuario

```
<p>
    Hola, <?php echo $sf_user->getAttribute('nombre') ?>
</p>
```

Nota Si se necesita guardar la información solamente durante la petición actual (por ejemplo, para pasar información a través de una sucesión de llamadas a acciones) es preferible utilizar la clase `sfRequest`, que también tiene métodos `getAttribute()` y `setAttribute()`. Solo los atributos del objeto `sfUser` son persistentes entre peticiones.

6.4.2. Atributos Flash

Un problema recurrente con los atributos del usuario es la limpieza de la sesión del usuario una vez que el atributo no se necesita más. Por ejemplo, puede ser necesario mostrar un mensaje de confirmación después de actualizar información mediante un formulario. Como la acción que maneja el formulario realiza una redirección, la única forma de pasar información desde esta acción a la acción que ha sido redireccionada es almacenar la información en la sesión del usuario. Pero una vez que se muestra el mensaje, es necesario borrar el atributo; ya que de otra forma, permanecerá en la sesión hasta que esta expire.

El atributo de tipo flash es un atributo fugaz que permite definirlo y olvidarse de él, sabiendo que desaparece automáticamente después de la siguiente petición y que deja la sesión limpia para las futuras peticiones. En la acción, se define el atributo flash de la siguiente manera:

```
| $this->getUser()->setFlash('atributo', $valor);
```

La plantilla se procesa y se envía al usuario, quien después realiza una nueva petición hacia otra acción. En esta segunda acción, es posible obtener el valor del atributo flash de esta forma:

```
| $valor = $this->getUser()->getFlash('atributo');
```

Luego ya te puedes olvidar de ese parámetro. Después de mostrar la segunda página, el atributo flash llamado `atributo` desaparece automáticamente. Incluso si no se utiliza el atributo durante la segunda acción, el atributo desaparece igualmente de la sesión.

Si necesitas acceder a un atributo flash desde la plantilla, puedes utilizar el objeto `$sf_user`:

```
<?php if ($sf_user->hasFlash('atributo')): ?>
    <?php echo $sf_user->getFlash('atributo') ?>
<?php endif; ?>
```

O simplemente:

```
<?php echo $sf_user->getFlash('atributo') ?>
```

Los atributos de tipo flash son una forma limpia de pasar información a la próxima petición.

6.4.3. Manejo de Sesiones

El manejo de sesiones de Symfony se encarga de gestionar automáticamente el almacenamiento de los IDs de sesión tanto en el cliente como en el servidor. Sin embargo, si se necesita modificar este comportamiento por defecto, es posible hacerlo. Se trata de algo que solamente lo necesitan los usuarios más avanzados.

En el lado del cliente, las sesiones son manejadas por cookies. La cookie de Symfony se llama `Symfony`, pero se puede cambiar su nombre editando el archivo de configuración `factories.yml`, como se muestra en el Listado 6-18.

Listado 6-18 - Cambiando el nombre de la cookie de sesión, en `apps/frontend/config/factories.yml`

```
all:
  storage:
    class: sfSessionStorage
    param:
      session_name: mi_nombre_cookie
```

Sugerencia La sesión se inicializa (con la función de PHP `session_start()`) solo si el parámetro `auto_start` de `factories.yml` tiene un valor de `true` (que es el caso por defecto). Si se quiere iniciar la sesión manualmente, se debe cambiar el valor de esa opción de configuración del archivo `factories.yml`.

El manejo de sesiones de Symfony esta basado en las sesiones de PHP. Por tanto, si la gestión de la sesión en la parte del cliente se quiere realizar mediante parámetros en la URL en lugar de cookies, se debe modificar el valor de la directiva `use_trans_sid` en el archivo de configuración `php.ini`. No obstante, se recomienda no utilizar esta técnica.

```
session.use_trans_sid = 1
```

En el lado del servidor, Symfony guarda por defecto las sesiones de usuario en archivos. Se pueden almacenar en la base de datos cambiando el valor del parámetro `class` en `factories.yml`, como se muestra en el Listado 6-19.

Listado 6-19 - Cambiando el almacenamiento de las sesiones en el servidor, en `apps/frontend/config/factories.yml`

```
all:
  storage:
    class: sfMySQLSessionStorage
    param:
      db_table: session           # Nombre de la tabla que guarda las sesiones
```



```

        database:      propel                # Nombre de La conexión a base de datos que se
        utiliza
        # Parámetros opcionales
        db_id_col:     sess_id              # Nombre de La columna que guarda el
        identificador de La sesión
        db_data_col:   sess_data            # Nombre de La columna que guarda Los datos de
        La sesión
        db_time_col:   sess_time            # Nombre de La columna que guarda el timestamp
        de La sesión

```

La opción `database` define el nombre de la conexión a base de datos que se utiliza. Posteriormente, Symfony utiliza el archivo `databases.yml` (ver capítulo 8) para determinar los parámetros con los que realiza la conexión (host, nombre de la base de datos, usuario y password).

Las clases disponibles para el almacenamiento de sesiones son `sfMySQLSessionStorage`, `sfMySQLiSessionStorage`, `sfPostgreSQLSessionStorage` y `sfPDOSessionStorage`. La clase recomendada es `sfPDOSessionStorage`. Para deshabilitar completamente el almacenamiento de las sesiones, se puede utilizar la clase `sfNoStorage`.

La expiración de la sesión se produce automáticamente después de 30 minutos. El valor de esta opción se puede modificar para cada entorno en el mismo archivo de configuración `factories.yml`, concretamente en la factoría correspondiente al usuario (user), tal y como muestra el listado 6-20.

Listado 6-20 - Cambiando el tiempo de vida de la sesión, en `apps/frontend/config/factories.yml`

```

all:
  user:
    class:      myUser
    param:
      timeout:  1800          # Tiempo de vida de La sesión en segundos

```

El capítulo 19 explica detalladamente todas las opciones de las factorías.

6.5. Seguridad de la Acción

La posibilidad de ejecutar una acción puede ser restringida a usuarios con ciertos privilegios. Las herramientas proporcionadas por Symfony para este propósito permiten la creación de aplicaciones seguras, en las que los usuarios necesitan estar autenticados antes de acceder a alguna característica o a partes de la aplicación. Añadir esta seguridad a una aplicación requiere dos pasos: declarar los requerimientos de seguridad para cada acción y autenticar a los usuarios con privilegios para que puedan acceder estas acciones seguras.

6.5.1. Restricción de Acceso

Antes de ser ejecutada, cada acción pasa por un filtro especial que verifica si el usuario actual tiene privilegios de acceder a la acción requerida. En Symfony, los privilegios están compuestos por dos partes:

- Las acciones seguras requieren que los usuarios estén autenticados.
- Las credenciales son privilegios de seguridad agrupados bajo un nombre y que permiten organizar la seguridad en grupos.

Para restringir el acceso a una acción se crea y se edita un archivo de configuración YAML llamado 'security.yml' en el directorio config/ del módulo. En este archivo, se pueden especificar los requerimientos de seguridad que los usuarios deberán satisfacer para cada acción o para todas (all) las acciones. El listado 6-21 muestra un ejemplo de security.yml.

Listado 6-21 - Estableciendo restricciones de acceso, en apps/frontend/modules/mimodulo/config/security.yml

```
ver:
  is_secure:  off      # Todos los usuarios pueden ejecutar la acción "ver"

modificar:
  is_secure:  on       # La acción "modificar" es sólo para usuarios autenticados

borrar:
  is_secure:  on       # Sólo para usuarios autenticados
  credentials: admin   # Con credencial "admin"

all:
  is_secure:  off      # off es el valor por defecto
```

Las acciones no incluyen restricciones de seguridad por defecto, así que cuando no existe el archivo security.yml o no se indica ninguna acción en ese archivo, todas las acciones son accesibles por todos los usuarios. Si existe un archivo security.yml, Symfony busca por el nombre de la acción y si existe, verifica que se satisfagan los requerimientos de seguridad. Lo que sucede cuando un usuario trata de acceder una acción restringida depende de sus credenciales:

- Si el usuario está autenticado y tiene las credenciales apropiadas, entonces la acción se ejecuta.
- Si el usuario no está autenticado, es redireccionado a la acción de login.
- Si el usuario está autenticado, pero no posee las credenciales apropiadas, será redirigido a la acción segura por defecto, como muestra la figura 6-1.

Las páginas login y secure son bastante simples, por lo que seguramente será necesario personalizarlas. Se puede configurar que acciones se ejecutan en caso de no disponer de suficientes privilegios en el archivo settings.yml de la aplicación cambiando el valor de las propiedades mostradas en el listado 6-22.

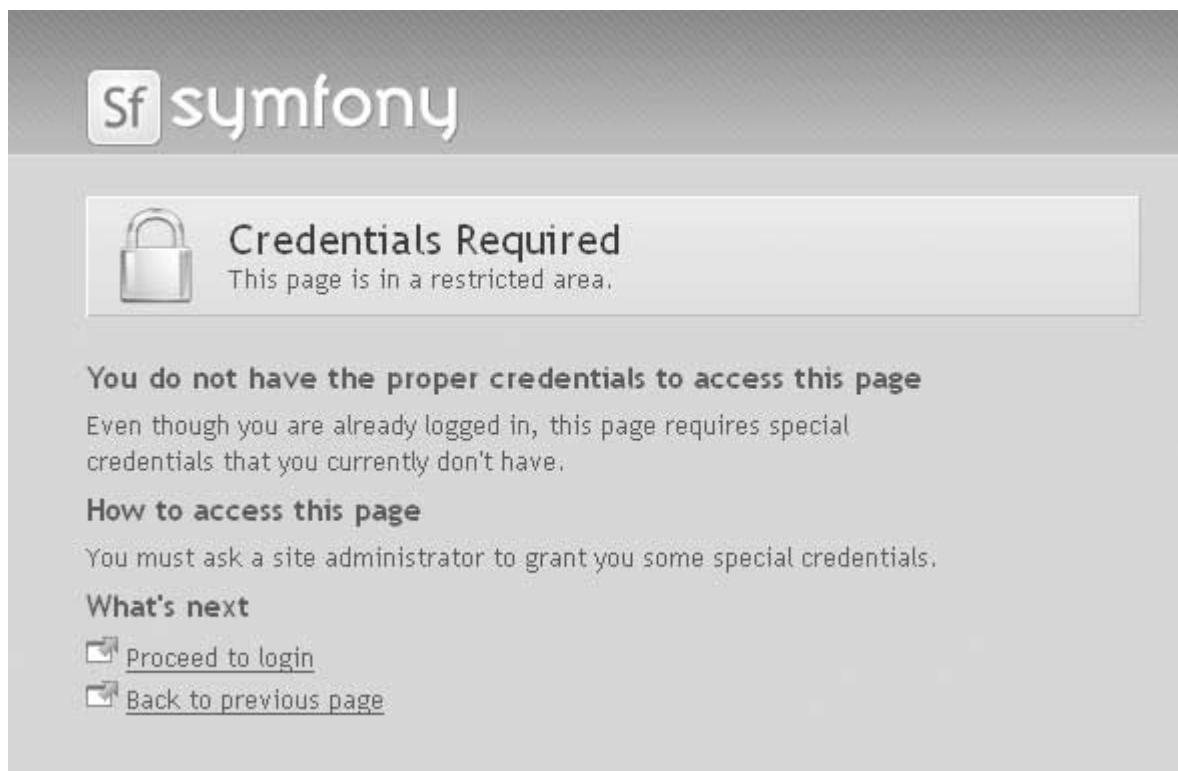


Figura 6.1. La página por defecto de la acción "secure"

Listado 6-22 - Las acciones de seguridad por defecto se definen en `apps/frontend/config/settings.yml`

```
all:
  .actions:
    login_module: default
    login_action: login

    secure_module: default
    secure_action: secure
```

6.5.2. Otorgando Acceso

Para obtener acceso a áreas restringidas, los usuarios necesitan estar autenticados y/o poseer ciertas credenciales. Puedes extender los privilegios del usuario mediante llamadas a métodos del objeto `sfUser`. El estado autenticado se establece con el método `setAuthenticated()` y se puede comprobar con el método `isAuthenticated()`. El listado 6-23 muestra un ejemplo sencillo de autenticación.

Listado 6-23 - Estableciendo el estado de autenticación del usuario

```
class miCuentaActions extends sfActions
{
  public function executeLogin($peticion)
  {
    if ($peticion->getParameter('login') == 'valor')
    {
      $this->getUser()->setAuthenticated(true);
    }
  }
}
```

```

    }

    public function executeLogout()
    {
        $this->getUser()->setAuthenticated(false);
    }
}

```

Las credenciales son un poco más complejas de tratar, ya que se pueden verificar, agregar, quitar y borrar. El listado 6-24 describe los métodos de las credenciales de la clase `sfUser`.

Listado 6-24 - Manejando las credenciales del usuario en la acción

```

class miCuentaActions extends sfActions
{
    public function executeEjemploDeCredenciales()
    {
        $usuario = $this->getUser();

        // Agrega una o más credenciales
        $usuario->addCredential('parametro');
        $usuario->addCredentials('parametro', 'valor');

        // Verifica si el usuario tiene una credencial
        echo $usuario->hasCredential('parametro');           => true

        // Verifica si un usuario tiene una de las credenciales
        echo $usuario->hasCredential(array('parametro', 'valor'));   => true

        // Verifica si el usuario tiene ambas credenciales
        echo $usuario->hasCredential(array('parametro', 'valor'), true);  => true

        // Quitar una credencial
        $usuario->removeCredential('parametro');
        echo $usuario->hasCredential('parametro');           => false

        // Elimina todas las credenciales (útil en el proceso de logout)
        $usuario->clearCredentials();
        echo $usuario->hasCredential('valor');               => false
    }
}

```

Si el usuario tiene la credencial "parametro", entonces ese usuario podrá acceder a las acciones para las cuales el archivo `security.yml` requiere esa credencial. Las credenciales se pueden utilizar también para mostrar contenido autenticado en una plantilla, como se muestra en el listado 6-25.

Listado 6-25 - Tratando con credenciales de usuario en una plantilla

```

<ul>
  <li><?php echo link_to('seccion1', 'content/seccion1') ?></li>
  <li><?php echo link_to('seccion2', 'content/seccion2') ?></li>
  <?php if ($sf_user->hasCredential('seccion3')): ?>
  <li><?php echo link_to('seccion3', 'content/seccion3') ?></li>

```

```
<?php endif; ?>
</ul>
```

Y para el estado de autenticación, las credenciales normalmente se dan a los usuarios durante el proceso de login. Este es el motivo por el que el objeto `sfUser` normalmente se extiende para añadir métodos de login y de logout, de forma que se pueda establecer el estado de seguridad del usuario de forma centralizada.

Sugerencia Entre los plugins de Symfony, `sfGuardPlugin` (<http://trac.symfony-project.com/wiki/sfGuardPlugin>) extiende la clase de sesión para facilitar el proceso de login y logout. El Capítulo 17 contiene más información al respecto.

6.5.3. Credenciales Complejas

La sintaxis YAML utilizada en el archivo `security.yml` permite restringir el acceso a usuarios que tienen una combinación de credenciales, usando asociaciones de tipo AND y OR. Con estas combinaciones, se pueden definir flujos de trabajo y sistemas de manejo de privilegios muy complejos -- como por ejemplo, un sistema de gestión de contenidos (CMS) cuya parte de gestión sea accesible solo a usuarios con credencial `admin`, donde los artículos pueden ser editados solo por usuarios con credenciales de `editor` y publicados solo por aquellos que tienen credencial de `publisher`. El listado 6-26 muestra este ejemplo.

Listado 6-26 - Sintaxis de combinación de credenciales

```
editarArticulo:
  credentials: [ admin, editor ]           # admin AND editor

publicarArticulo:
  credentials: [ admin, publisher ]       # admin AND publisher

gestionUsuarios:
  credentials: [[ admin, superuser ]]     # admin OR superuser
```

Cada vez que se añade un nuevo nivel de corchetes, la lógica cambia entre AND y OR. Así que se pueden crear combinaciones muy complejas de credenciales, como la siguiente:

```
credentials: [[root, [supplier, [owner, quasiowner]], accounts]]
              # root OR (supplier AND (owner OR quasiowner)) OR accounts
```

6.6. Métodos de Validación y Manejo de Errores

Nota Las opciones que se describen en esta sección han sido declaradas obsoletas en Symfony 1.1, por lo que solamente funcionan si activas el plugin `sfCompat10`.

La validación de los datos de la acción -normalmente los parámetros de la petición- es una tarea repetitiva y tediosa. Symfony incluye un sistema de validación, utilizando métodos de la clase acción.

Se ve en primer lugar un ejemplo. Cuando un usuario hace una petición a `miAccion`, Symfony siempre busca primero un método llamado `validateMiAccion()`. Si lo encuentra, Symfony ejecuta ese método. El valor de retorno de esta validación determina el siguiente método que se

ejecuta: si devuelve true, entonces se ejecuta el método `executeMiAccion()`; en otro caso, se ejecuta `handleErrorMiAccion()`. En el caso de que `handleErrorMiAccion()` no exista, Symfony busca un método genérico llamado `handleError()`. Si tampoco existe, simplemente devuelve el valor `sfView::ERROR` para producir la plantilla `miAccionError.php`. La Figura 6-2 ilustra este proceso.

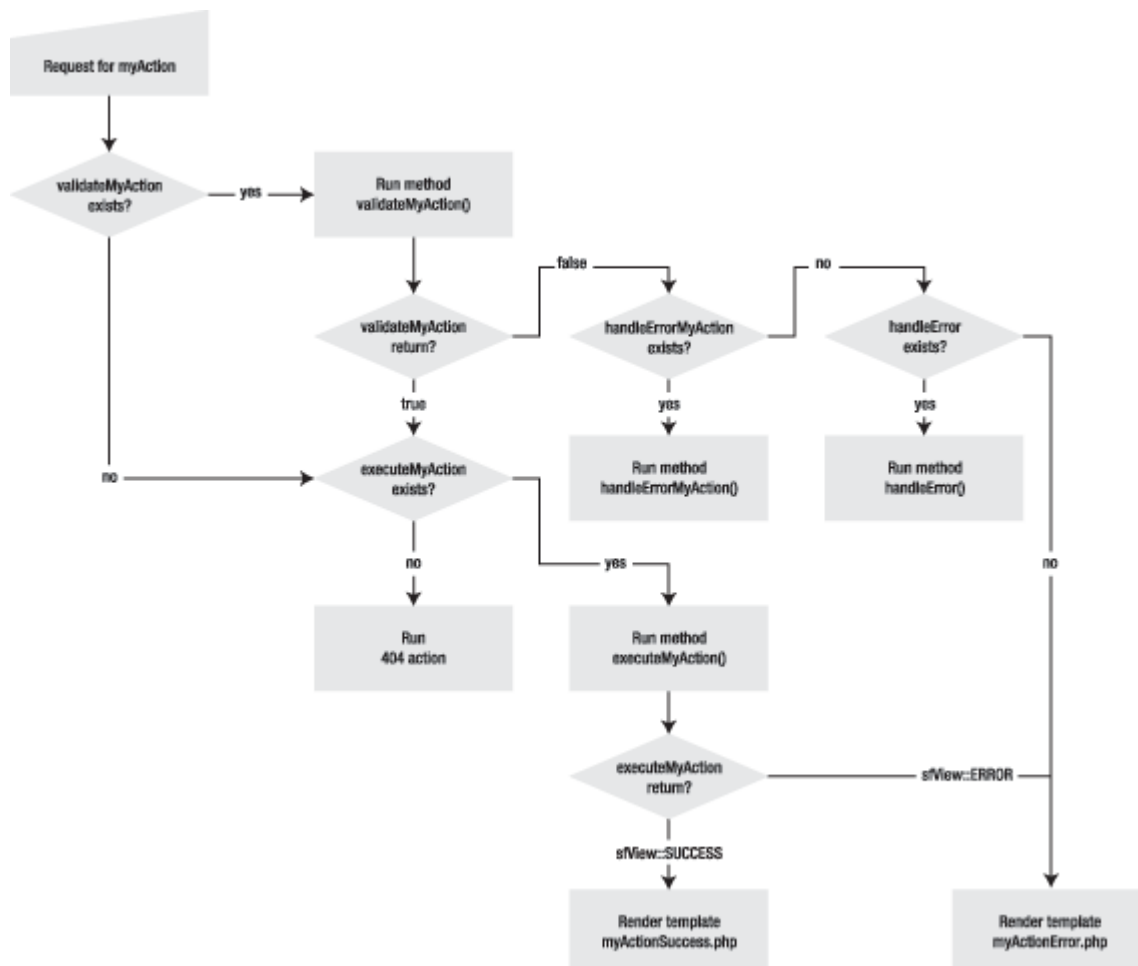


Figura 6.2. El proceso de validación

La clave para un correcto funcionamiento de la validación es respetar la convención de nombres para los métodos de la acción:

- `validateNombreAccion` es el método de validación, que devuelve true o false. Se trata del primer método buscado cuando se solicita la acción `NombreAccion`. Si no existe, la acción se ejecuta directamente.
- `handleErrorNombreAccion` es el método llamado cuando el método de validación falla. Si no existe, entonces se muestra la plantilla Error.
- `executeNombreAccion` es el método de la acción. Debe existir para todas las acciones.

El listado 6-27 muestra un ejemplo de una acción con métodos de validación. Independientemente de si la validación falla o no falla, en el siguiente ejemplo se ejecuta la plantilla `miAccionSuccess.php` pero no con los mismos parámetros.

Listado 6-27 - Ejemplo de métodos de validación

```
class mimoduloActions extends sfActions
{
    public function validateMiAccion($peticion)
    {
        return $peticion->getParameter('id') > 0;
    }

    public function handleErrorMiAccion()
    {
        $this->message = "Parámetros no válidos";

        return sfView::SUCCESS;
    }

    public function executeMiAccion()
    {
        $this->message = "Los parámetros son válidos";
    }
}
```

Se puede incluir cualquier código en el método `validate()`. La única condición es que devuelva un valor `true` o `false`. Como es un método de la clase `sfActions`, tiene acceso a los objetos `sfRequest` y `sfUser`, que pueden ser realmente útiles para validación de los datos de la petición y del contexto.

Se pueden utilizar este mecanismo para implementar la validación de los formularios (esto es, controlar los valores introducidos por el usuario en un formulario antes de procesarlo), pero se trata de una tarea muy repetitiva para la que Symfony proporciona herramientas automatizadas, como las descritas en el Capítulo 10.

6.7. Filtros

El mecanismo de seguridad puede ser entendido como un filtro, por el que debe pasar cada petición antes de ejecutar la acción. Según las comprobaciones realizadas en el filtro, se puede modificar el procesamiento de la petición --por ejemplo, cambiando la acción ejecutada (`default/secure` en lugar de la acción solicitada en el caso del filtro de seguridad). Symfony extiende esta idea a clases de filtros. Se puede especificar cualquier número de clases de filtros a ser ejecutadas antes de que se procese la respuesta, y además hacerlo de forma sistemática para todas las peticiones. Se pueden entender los filtros como una forma de empaquetar cierto código de forma similar a `preExecute()` y `postExecute()`, pero a un nivel superior (para toda una aplicación en lugar de para todo un módulo).

6.7.1. La Cadena de Filtros

Symfony de hecho procesa cada petición como una cadena de filtros ejecutados de forma sucesiva. Cuando el framework recibe una petición, se ejecuta el primer filtro (que siempre es `sfRenderingFilter`). En algún punto, llama al siguiente filtro en la cadena, luego el siguiente, y así sucesivamente. Cuando se ejecuta el último filtro (que siempre es `sfExecutionFilter`), los

filtros anteriores pueden finalizar, y así hasta el filtro de `sfRenderingFilter`. La Figura 6-3 ilustra esta idea con un diagrama de secuencias, utilizando una cadena de filtros simplificada (la cadena real tiene muchos más filtros).

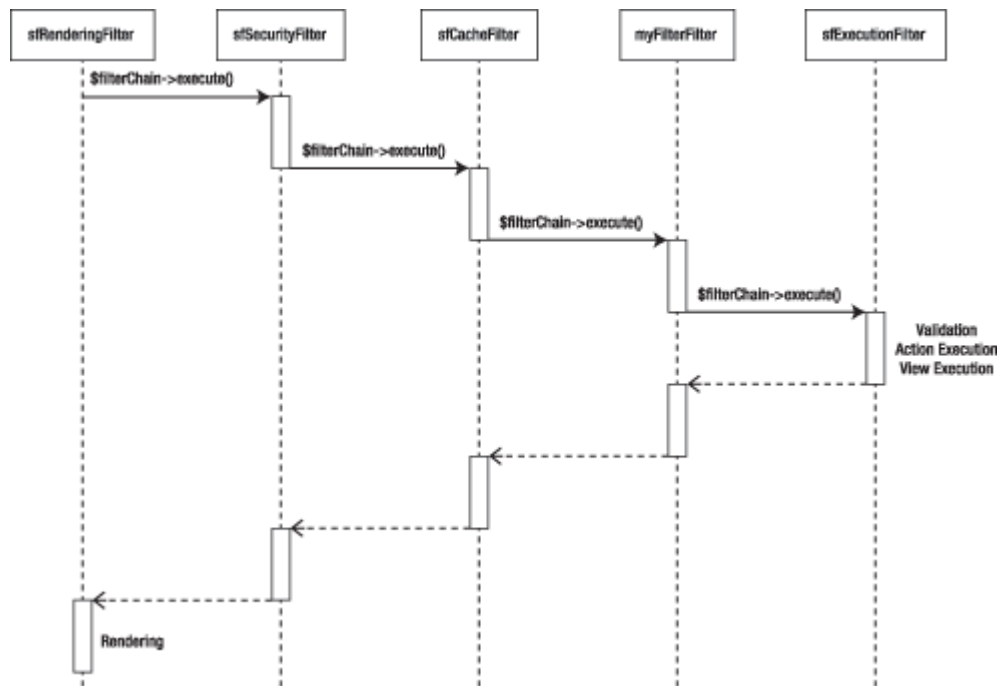


Figura 6.3. Ejemplo de cadena de filtros

Este proceso es la razón de la estructura de las clases de tipo filtro. Todas estas clases extienden la clase `sfFilter` y contienen un método `execute()` que espera un objeto de tipo `$filterChain` como parámetro. En algún punto de este método, el filtro pasa al siguiente filtro en la cadena, llamando a `$filterChain->execute()`. El listado 6-28 muestra un ejemplo. Por lo tanto, los filtros se dividen en dos partes:

- El código que se encuentra antes de la llamada a `$filterChain->execute()` se ejecuta antes de que se ejecute la acción.
- El código que se encuentra después de la llamada a `$filterChain->execute()` se ejecuta después de la acción y antes de producir la vista.

Listado 6-28 - Estructura de la clase filtro

```

class miFiltro extends sfFilter
{
    public function execute ($filterChain)
    {
        // Código que se ejecuta antes de la ejecución de la acción
        ...

        // Ejecutar el siguiente filtro de la cadena
        $filterChain->execute();

        // Código que se ejecuta después de la ejecución de la acción y antes de que se genere la vista
        ...
    }
}
  
```



```

    }
  }

```

La cadena de filtros por defecto se define en el archivo de configuración de la aplicación `filters.yml`, y su contenido se muestra en el listado 6-29. Este archivo lista los filtros que se ejecutan para cada petición.

Listado 6-29 - Cadena de filtros por defecto, en `frontend/config/filters.yml`

```

rendering: ~
security: ~

# Normalmente los filtros propios se insertan aqui

cache:      ~
common:     ~
execution:  ~

```

Estas declaraciones no tienen parámetros (el caracter tilde, ~, significa null en YAML), porque heredan los parámetros definidos en el núcleo de Symfony. En su núcleo, Symfony define las opciones `class` y `param` para cada uno de estos filtros. Por ejemplo, el listado 6-30 muestra los parámetros por defecto para el filtro `rendering`.

Listado 6-30 - Parámetros por defecto del filtro `sfRenderingFilter`, en `$sf_symfony_lib_dir/config/config/filters.yml`

```

rendering:
  class: sfRenderingFilter  # Clase del filtro
  param:                    # Parámetros del filtro
  type: rendering

```

Si se deja el valor vacío (~) en el archivo `filters.yml` de la aplicación, Symfony aplica el filtro con las opciones por defecto definidas en su núcleo.

Se pueden personalizar las cadenas de filtros en varias formas:

- Desactivando algún filtro de la cadena agregando un parámetro `enabled: off`. Para desactivar por ejemplo el filtro `common`, que se encarga de añadir los archivos CSS y JavaScript en la cabecera de la página, se añade la siguiente opción:

```

common:
  enabled: off

```

- No se deben borrar las entradas del archivo `filters.yml` para desactivar un filtro ya que Symfony lanzará una excepción.
- Se pueden añadir declaraciones propias en cualquier lugar de la cadena (normalmente después del filtro `security`) para agregar un filtro propio (como se verá en la próxima sección). En cualquier caso, el filtro `rendering` debe ser siempre la primera entrada, y el filtro `execution` debe ser siempre la última entrada en la cadena de filtros.
- Redefinir la clase y los parámetros por defecto del filtro por defecto (normalmente para modificar el sistema de seguridad y utilizar un filtro de seguridad propio).

6.7.2. Construyendo Tu Propio Filtro

Construir un filtro propio es bastante sencillo. Se debe crear una definición de una clase similar a la demostrada en el listado 6-28, y se coloca en una de los directorios `lib/` del proyecto para aprovechar la carga automática de clases.

Como una acción puede pasar el control o redireccionar hacia otra acción y en consecuencia relanzar toda la cadena de filtros, quizás sea necesario restringir la ejecución de los filtros propios a la primera acción de la petición. El método `isFirstCall()` de la clase `sfFilter` retorna un valor booleano con este propósito. Esta llamada solo tiene sentido antes de la ejecución de una acción.

Este concepto se puede entender fácilmente con un ejemplo. El listado 6-31 muestra un filtro utilizado para auto-loguear a los usuarios con una cookie `MiSitioWeb`, que se supone que se crea en la acción `login`. Se trata de una forma rudimentaria pero que funciona para incluir la característica *Recuérdame* de un formulario de login.

Listado 6-31 - Ejemplo de archivo de clase de filtro, en `apps/frontend/lib/rememberFilter.class.php`

```
class rememberFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // Ejecutar este filtro solo una vez
        if ($this->isFirstCall())
        {
            // Los filtros no tienen acceso directo a los objetos user y request.
            // Se necesita el contexto para obtenerlos
            $peticion = $this->getContext()->getRequest();
            $usuario = $this->getContext()->getUser();

            if ($peticion->getCookie('MiSitioWeb'))
            {
                // Logueado
                $usuario->setAuthenticated(true);
            }
        }

        // Ejecutar el proximo filtro
        $filterChain->execute();
    }
}
```

En ocasiones, en lugar de continuar con la ejecución de la cadena de filtros, se necesita pasar el control a una acción específica al final de un filtro. `sfFilter` no tiene un método `forward()`, pero `sfController` si, por lo que simplemente se puede llamar al siguiente método:

```
| return $this->getContext()->getController()->forward('mimodulo', 'miAccion');
```

Nota La clase `sfFilter` tiene un método `initialize()`, ejecutado cuando se crea el objeto filtro. Se puede redefinir en el filtro propio si se necesita trabajar de forma personalizada con los parámetros de los filtros (definidos en `filters.yml`, como se describe a continuación).

6.7.3. Activación de Filtros y Parámetros

Crear un filtro no es suficiente para activarlo. Se necesita agregar el filtro propio a la cadena, y para eso, se debe declarar la clase del filtro en el archivo `filters.yml`, localizado en el directorio `config/` de la aplicación o del módulo, como se muestra en el listado 6-32.

Listado 6-32 - Ejemplo de archivo de activación de filtro, en `apps/frontend/config/filters.yml`

```
rendering: ~
security: ~

remember:          # Los filtros requieren un nombre único
  class: rememberFilter
  param:
    cookie_name: MiSitioWeb
    condition:   %APP_ENABLE_REMEMBER_ME%

cache: ~
common: ~
execution: ~
```

Cuando se encuentra activo, el filtro se ejecuta en cada petición. El archivo de configuración de los filtros puede contener una o más definiciones de parámetros en la sección `param`. La clase filtro puede obtener estos parámetros con el método `getParameter()`. El listado 6-33 muestra como obtener los valores de los parámetros.

Listado 6-33 - Obteniendo el valor del parámetro, en `apps/frontend/lib/rememberFilter.class.php`

```
class rememberFilter extends sfFilter
{
  public function execute($filterChain)
  {
    // ...

    if ($request->getCookie($this->getParameter('cookie_name')))
    {
      // ...
    }

    // ...
  }
}
```

El parámetro `condition` se comprueba en la cadena de filtros para ver si el filtro debe ser ejecutado. Por lo que las declaraciones del filtro propio puede basarse en la configuración de la aplicación, como muestra el listado 6-32. El filtro `remember` se ejecuta solo si el archivo `app.yml` incluye lo siguiente:

```
all:
  enable_remember_me: on
```

6.7.4. Filtros de Ejemplo

Los filtros son útiles para repetir cierto código en todas las acciones. Por ejemplo, si se utiliza un sistema remoto de estadísticas, puede ser necesario añadir un trozo de código que realice una llamada a un script de las estadísticas en cada página. Este código se puede colocar en el layout global, pero entonces estaría activo para toda la aplicación. Otra forma es colocarlo en un filtro, como se muestra el listado 6-34, y activarlo en cada módulo.

Listado 6-34 - Filtro para el sistema de estadísticas de Google Analytics

```
class sfGoogleAnalyticsFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // No se hace nada antes de la acción
        $filterChain->execute();

        // Decorar la respuesta con el código de Google Analytics
        $codigoGoogle = '
<script src="http://www.google-analytics.com/urchin.js" type="text/javascript">
</script>
<script type="text/javascript">
    _uacct="UA-'. $this->getParameter('google_id').'";urchinTracker();
</script>';
        $respuesta = $this->getContext()->getResponse();
        $respuesta->setContent(str_ireplace('</body>',
        $codigoGoogle. '</body>', $respuesta->getContent()));
    }
}
```

No obstante, este filtro no es perfecto, ya que no se debería añadir el código de Google si la respuesta no es de tipo HTML.

Otro ejemplo es el de un filtro que cambia las peticiones a SSL si no lo son, para hacer más segura la comunicación, como muestra el Listado 6-35.

Listado 6-35 - Filtro de comunicación segura

```
class sfSecureFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $contexto = $this->getContext();
        $peticion = $contexto->getRequest();
        if (!$peticion->isSecure())
        {
            $urlSegura = str_replace('http', 'https', $peticion->getUri());
            return $contexto->getController()->redirect($urlSegura);
            // No se continúa con la cadena de filtros
        }
        else
        {
            // La petición ya es segura, así que podemos continuar
            $filterChain->execute();
        }
    }
}
```

```
| }  
| }
```

Los filtros se utilizan mucho en los plugins, porque permiten extender las características de una aplicación de forma global. El Capítulo 17 incluye más información sobre los plugins, y el wiki del proyecto Symfony (<http://trac.symfony-project.com/>) también tiene más ejemplos de filtros.

6.8. Configuración del Módulo

Algunas características de los módulos dependen de la configuración. Para modificarlas, se debe crear un archivo `module.yml` en el directorio `config/` y se deben definir parámetros para cada entorno (o en la sección `all:` para todos los entornos). El listado 6-36 muestra un ejemplo de un archivo `module.yml` para el módulo `mimodulo`.

Listing 6-36 - Configuración del módulo, en `apps/frontend/modules/mimodulo/config/module.yml`

```
| all:                                # Para todos Los entornos  
|   enabled:      true  
|   is_internal: false  
|   view_class:  sfPHP
```

El parámetro `enabled` permite desactivar todas las acciones en un módulo. En ese caso, todas las acciones se redireccionan a la acción `module_disabled_module/module_disabled_action` (tal y como se define en el archivo `settings.yml`).

El parámetro `is_internal` permite restringir la ejecución de todas las acciones de un módulo a llamadas internas. Esto es útil por ejemplo para acciones de envío de correos electrónicos que se deben llamar desde otras acciones para enviar mensajes de e-mail, pero que no se deben llamar desde el exterior.

El parámetro `view_class` define la clase de la vista. Debe heredar de `sfView`. Sobreescibir este valor permite utilizar otros sistemas de generación de vistas con otros motores de plantillas, como por ejemplo Smarty.

6.9. Resumen

En Symfony, la capa del controlador esta dividida en dos partes: el controlador frontal, que es el único punto de entrada a la aplicación para un entorno dado, y las acciones, que contienen la lógica de las páginas. Una acción puede elegir la forma en la que se ejecuta su vista, devolviendo un valor correspondiente a una de las constantes de la clase `sfView`. Dentro de una acción, se pueden manipular los diferentes elementos del contexto, incluidos el objeto de la petición (`sfRequest`) y el objeto de la sesión del usuario actual (`sfUser`).

Combinando el poder del objeto de sesión, el objeto acción y las configuraciones de seguridad proporcionan sistema de seguridad completo, con restricciones de acceso y credenciales. Los métodos especiales `validate()` y `handleError()` en la acciones permiten gestionar la validación de las peticiones. Y si los métodos `preExecute()` y `postExecute()` se diseñan para la

reutilización de código dentro de un módulo, los filtros permiten la misma reutilización para toda la aplicación ejecutando código del controlador para cada petición.

Capítulo 7. La Vista

La vista se encarga de producir las páginas que se muestran como resultado de las acciones. La vista en Symfony está compuesta por diversas partes, estando cada una de ellas especialmente preparada para que pueda ser fácilmente modificable por la persona que normalmente trabaja con cada aspecto del diseño de las aplicaciones.

- Los diseñadores web normalmente trabajan con las plantillas (que son la presentación de los datos de la acción que se está ejecutando) y con el layout (que contiene el código HTML común a todas las páginas). Estas partes están formadas por código HTML que contiene pequeños trozos de código PHP, que normalmente son llamadas a los diversos *helpers* disponibles.
- Para mejorar la reutilización de código, los programadores suelen extraer trozos de las plantillas y los transforman en componentes y elementos parciales. De esta forma, el layout se modifica para definir zonas en las que se insertan componentes externos. Los diseñadores web también pueden trabajar fácilmente con estos trozos de plantillas.
- Los programadores normalmente centran su trabajo relativo a la vista en los archivos de configuración YAML (que permiten establecer opciones para las propiedades de la respuesta y para otros elementos de la interfaz) y en el objeto respuesta. Cuando se trabaja con variables en las plantillas, deben considerarse los posibles riesgos de seguridad de XSS (*cross-site scripting*) por lo que es necesario conocer las técnicas de escape de los caracteres introducidos por los usuarios.

Independientemente del tipo de trabajo, existen herramientas y utilidades para simplificar y acelerar el trabajo (normalmente tedioso) de presentar los resultados de las acciones. En este capítulo se detallan todas estas herramientas.

7.1. Plantillas

El Listado 7-1 muestra el código típico de una plantilla. Su contenido está formado por código HTML y algo de código PHP sencillo, normalmente llamadas a las variables definidas en la acción (mediante la instrucción `$this->nombre_variable = 'valor';`) y algunos *helpers*.

Listado 7-1 - Plantilla de ejemplo `indexSuccess.php`

```
<h1>Bienvenido</h1>
<p>¡Hola de nuevo, <?php echo $nombre ?>!</p>
<ul>¿Qué es lo que quieres hacer?
  <li><?php echo link_to('Leer los últimos artículos', 'articulo/leer') ?></li>
  <li><?php echo link_to('Escribir un nuevo artículo', 'articulo/escribir') ?></li>
</ul>
```

Como se explica en el Capítulo 4, es recomendable utilizar la sintaxis alternativa de PHP en las plantillas para hacerlas más fáciles de leer a aquellos desarrolladores que desconocen PHP. Se debería minimizar en lo posible el uso de código PHP en las plantillas, ya que estos archivos son los que se utilizan para definir la interfaz de la aplicación, y muchas veces son diseñados y

modificados por otros equipos de trabajo especializados en el diseño de la presentación y no de la lógica del programa. Además, incluir la lógica dentro de las acciones permite disponer de varias plantillas para una sola acción sin tener que duplicar el código.

7.1.1. Helpers

Los *helpers* son funciones de PHP que devuelven código HTML y que se utilizan en las plantillas. En el listado 7-1, la función `link_to()` es un *helper*. A veces, los *helpers* solamente se utilizan para ahorrar tiempo, agrupando en una sola instrucción pequeños trozos de código utilizados habitualmente en las plantillas. Por ejemplo, es fácil imaginarse la definición de la función que representa a este *helper*:

```
<?php echo input_tag('nick') ?>
=> <input type="text" name="nick" id="nick" value="" />
```

La función debería ser como la que se muestra en el listado 7-2.

Listado 7-2 - Ejemplo de definición de helper

```
function input_tag($name, $value = null)
{
    return '<input type="text" name="'. $name. '" id="'. $name. '" value="'. $value. '" />';
}
```

En realidad, la función `input_tag()` que incluye Symfony es un poco más complicada que eso, ya que permite indicar un tercer parámetro que contiene otros atributos de la etiqueta `<input>`. Se puede consultar su sintaxis completa y sus opciones en la documentación de la API: http://www.symfony-project.org/api/1_1/

La mayoría de las veces los *helpers* incluyen cierta inteligencia que evita escribir bastante código:

```
<?php echo auto_link_text('Por favor, visita nuestro sitio web www.ejemplo.com') ?>
=> Por favor, visita nuestro sitio web <a
href="http://www.ejemplo.com">www.ejemplo.com</a>
```

Los *helpers* facilitan la creación de las plantillas y producen el mejor código HTML posible en lo que se refiere al rendimiento y a la accesibilidad. Aunque se puede usar HTML normal y corriente, los *helpers* normalmente son más rápidos de escribir.

Sugerencia Quizás te preguntes por qué motivo los *helpers* se nombran con la sintaxis de los guiones bajos en vez de utilizar el método `camelCase` que se utiliza en el resto de Symfony. El motivo es que los *helpers* son funciones, y todas las funciones de PHP utilizan la sintaxis de los guiones bajos.

7.1.1.1. Declarando los Helpers

Los archivos de Symfony que contienen los *helpers* no se cargan automáticamente (ya que contienen funciones, no clases). Los *helpers* se agrupan según su propósito. Por ejemplo el archivo llamado `TextHelper.php` contiene todas las funciones de los *helpers* relacionados con el texto, que se llaman "grupo de helpers de Text". De esta forma, si una plantilla va a utilizar un *helper*, se debe cargar previamente el grupo al que pertenece el *helper* mediante la función

`use_helper()`. El listado 7-3 muestra una plantilla que hace uso del *helper* `auto_link_text()`, que forma parte del grupo `Text`.

Listado 7-3 - Declarando el uso de un helper

```
// Esta plantilla utiliza un grupo de helpers específicos
<?php use_helper('Text') ?>
...
<h1>Descripción</h1>
<p><?php echo auto_link_text($descripcion) ?></p>
```

Sugerencia Si se necesita declarar más de un grupo de helpers, se deben añadir más argumentos a la llamada de la función `use_helper()`. Si por ejemplo se necesitan cargar los helpers `Text` y `Javascript`, la llamada a la función debe ser `<?php echo use_helper('Text', 'Javascript') ?>`.

Por defecto algunos de los helpers están disponibles en las plantillas sin necesidad de ser declarados. Estos *helpers* pertenecen a los siguientes grupos:

- **Helper:** se necesita para incluir otros *helpers* (de hecho, la función `use_helper()` también es un *helper*)
- **Tag:** *helper* básico para etiquetas y que utilizan casi todos los *helpers*
- **Url:** *helpers* para la gestión de enlaces y URL
- **Asset:** *helpers* que añaden elementos a la sección `<head>` del código HTML y que proporcionan enlaces sencillos a elementos externos (imágenes, archivos JavaScript, hojas de estilo, etc.)
- **Partial:** *helpers* que permiten incluir trozos de plantillas
- **Cache:** manipulación de los trozos de código que se han añadido a la cache
- **Form:** *helpers* para los formularios

El archivo `settings.yml` permite configurar la lista de *helpers* que se cargan por defecto en todas las plantillas. De esta forma, se puede modificar su configuración si se sabe por ejemplo que no se van a usar los *helpers* relacionados con la cache o si se sabe que siempre se van a necesitar los *helpers* relacionados con el grupo `Text`. Este cambio puede aumentar ligeramente la velocidad de ejecución de la aplicación. Los 4 primeros *helpers* de la lista anterior (`Helper`, `Tag`, `Url` y `Asset`) no se pueden eliminar, ya que son obligatorios para que funcione correctamente el mecanismo de las plantillas. Por este motivo ni siquiera aparecen en la lista de *helpers* estándares.

Sugerencia Si se quiere utilizar un helper fuera de una plantilla, se puede cargar un grupo de helpers desde cualquier punto de la aplicación mediante la función `sfLoader::loadHelpers($helpers)`, donde la variable `$helpers` es el nombre de un grupo de helpers o un array con los nombres de varios grupos de helpers. Por tanto, si se quiere utilizar `auto_link_text()` dentro de una acción, es necesario llamar primero a `sfLoader::loadHelpers('Text')`.

7.1.1.2. Los helpers habituales

Algunos *helpers* se explican en detalle en los siguientes capítulos, en función de la característica para la que han sido creados. El listado 7-4 incluye una pequeña lista de los helpers que más se utilizan y muestra también el código HTML que generan.

Listado 7-4 - Los *helpers* por defecto más utilizados

```
// Grupo Helper
<?php use_helper('NombreHelper') ?>
<?php use_helper('NombreHelper1', 'NombreHelper2', 'NombreHelper3') ?>

// Grupo Tag
<?php echo tag('input', array('name' => 'parametro', 'type' => 'text')) ?>
<?php echo tag('input', 'name=parametro type=text') ?> // Sintaxis alternativa para
Las opciones
=> <input name="parametro" type="text" />
<?php echo content_tag('textarea', 'contenido de prueba', 'name=parametro') ?>
=> <textarea name="parametro">contenido de prueba</textarea>

// Grupo Url
<?php echo link_to('Píname', 'mimodulo/miaccion') ?>
=> <a href="/ruta/a/miaccion">Píname</a> // Depende del sistema de enrutamiento

// Grupo Asset
<?php echo image_tag('miimagen', 'alt=imagen size=200x100') ?>
=> 
<?php echo javascript_include_tag('miscrypt') ?>
=> <script language="JavaScript" type="text/javascript" src="/js/miscrypt.js"></script>
<?php echo stylesheet_tag('estilo') ?>
=> <link href="/stylesheets/estilo.css" media="screen" rel="stylesheet" type="text/css" />
```

Symfony incluye muchos otros *helpers* y describirlos todos requeriría de un libro entero. La mejor referencia para estudiar los *helpers* es la documentación de la API, que se puede consultar en http://www.symfony-project.org/api/1_1/, donde todos los *helpers* incluyen documentación sobre su sintaxis, opciones y ejemplos.

7.1.1.3. Crea tus propios helpers

Symfony incluye numerosos *helpers* que realizan distintas funcionalidades, pero si no se encuentra lo que se necesita, es probable que tengas que crear un nuevo *helper*. Crear un *helper* es muy sencillo.

Las funciones del *helper* (funciones normales de PHP que devuelven código HTML) se deben guardar en un archivo llamado `NombreHelper.php`, donde `Nombre` es el nombre del nuevo grupo de *helpers*. El archivo se debe guardar en el directorio `apps/frontend/lib/helper/` (o en cualquier directorio `helper/` que esté dentro de cualquier directorio `lib/` del proyecto) para que la función `use_helper('Nombre')` pueda encontrarlo de forma automática y así poder incluirlo en la plantilla.

Sugerencia Este mecanismo permite incluso redefinir los helpers de Symfony. Para redefinir por ejemplo todos los helpers del grupo Text, se puede crear un archivo llamado `TextHelper.php` y guardarlo en el directorio `apps/frontend/lib/helper/`. Cada vez que se llame a la función `use_helper('Text')`, Symfony carga el nuevo grupo de helpers en vez del grupo por defecto. Hay que ser cuidadoso con este método, ya que como el archivo original no se carga, el nuevo grupo de helpers debe redefinir todas y cada una de las funciones del grupo original, ya que de otra forma no estarán disponibles las funciones no definidas.

7.1.2. Layout de las páginas

La plantilla del listado 7-1 no es un documento XHTML válido. Le faltan la definición del DOCTYPE y las etiquetas `<html>` y `<body>`. El motivo es que estos elementos se encuentran en otro lugar de la aplicación, un archivo llamado `layout.php` que contiene el layout de la página. Este archivo, que también se denomina plantilla global, almacena el código HTML que es común a todas las páginas de la aplicación, para no tener que repetirlo en cada página. El contenido de la plantilla se integra en el layout, o si se mira desde el otro punto de vista, el layout *decora* la plantilla. Este comportamiento es una implementación del patrón de diseño llamado "decorator" y que se muestra en la figura 7-1.

Sugerencia Para obtener más información sobre el patrón "decorator" y sobre otros patrones de diseño, se puede consultar el libro *"Patterns of Enterprise Application Architecture"* escrito por Martin Fowler (Addison-Wesley, ISBN: 0-32112-742-0).

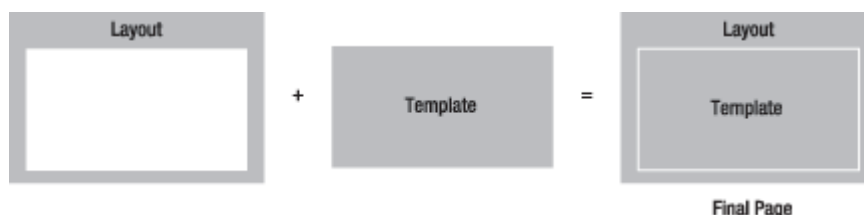


Figura 7.1. Plantilla decorada con un layout

El listado 7-5 muestra el layout por defecto, que se encuentra en el directorio `templates/`.

Listado 7-5 - Layout por defecto, en `miproyecto/apps/frontend/templates/layout.php`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <?php include_http_metas() ?>
    <?php include_metas() ?>
    <?php include_title() ?>
    <link rel="shortcut icon" href="/favicon.ico" />
  </head>
  <body>
    <?php echo $sf_content ?>
  </body>
</html>
```

Los *helpers* utilizados en la sección `<head>` obtienen información del objeto respuesta y en la configuración de la vista. La etiqueta `<body>` muestra el resultado de la plantilla. Utilizando este

layout, la configuración por defecto y la plantilla de ejemplo del listado 7-1, la vista generada sería la del listado 7-6.

Listado 7-6 - Unión del layout, la configuración de la vista y la plantilla

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <meta name="title" content="symfony project" />
    <meta name="robots" content="index, follow" />
    <meta name="description" content="symfony project" />
    <meta name="keywords" content="symfony, project" />
    <title>symfony project</title>
    <link rel="stylesheet" type="text/css" href="/css/main.css" />
    <link rel="shortcut icon" href="/favicon.ico">
  </head>
  <body>
    <h1>Bienvenido</h1>
    <p>¡Hola de nuevo, <?php echo $nombre ?>!</p>
    <ul>¿Qué es lo que quieres hacer?
      <li><?php echo link_to('Leer los últimos artículos', 'articulo/leer') ?></li>
      <li><?php echo link_to('Escribir un nuevo artículo', 'articulo/escribir') ?></li>
    </ul>
  </body>
</html>
```

La plantilla global puede ser adaptada completamente para cada aplicación. Se puede añadir todo el código HTML que sea necesario. Normalmente se utiliza el layout para mostrar la navegación, el logotipo del sitio, etc. Incluso es posible definir más de un layout y decidir en cada acción el layout a utilizar. No te preocupes ahora por la forma de incluir archivos de JavaScript y hojas de estilos, ya que se explica en la sección "Configuración de la Vista" más adelante en este capítulo.

7.1.3. Atajos de plantilla

Symfony incluye una serie de variables propias en todas las plantillas. Estas variables se pueden considerar *atajos* que permiten el acceso directo a la información más utilizada en las plantillas, mediante los siguientes objetos internos de Symfony:

- `$sf_context`: el objeto que representa a todo el contexto (es una instancia de `sfContext`)
- `$sf_request`: el objeto petición (es una instancia de `sfRequest`)
- `$sf_params`: los parámetros del objeto petición
- `$sf_user`: el objeto de sesión del usuario actual (es una instancia de `sfUser`)

En el capítulo anterior se detallaban algunos métodos útiles de los objetos `sfRequest` y `sfUser`. En las plantillas se pueden invocar todos esos métodos mediante las variables `$sf_request` y `$sf_user`. Por ejemplo, si la petición incluye un parámetro llamado `total`, desde la plantilla se puede acceder a su valor de la siguiente manera:

```
// Método Largo
<?php echo $sf_request->getParameter('total') ?>

// Método corto (atajo)
<?php echo $sf_params->get('total') ?>

// Son equivalentes al siguiente código de la acción
echo $peticion->getParameter('total')
```

7.2. Fragmentos de código

En ocasiones es necesario incluir cierto código HTML o PHP en varias páginas. Para no tener que repetirlo, casi siempre es suficiente con utilizar la instrucción `include()`.

Si por ejemplo varias de las plantillas de la aplicación utilizan el mismo fragmento de código, se puede guardar en un archivo llamado `miFragmento.php` en el directorio global de plantillas (`miproyecto/apps/frontend/templates/`) e incluirlo en las plantillas mediante la instrucción siguiente:

```
| <?php include(sfConfig::get('sf_app_template_dir').'/miFragmento.php') ?>
```

Sim embargo, esta forma de trabajar con fragmentos de código no es muy limpia, sobre todo porque puede que los nombres de las variables utilizadas no coincidan en el fragmento de código y en las distintas plantillas. Además, el sistema de cache de Symfony (que se explica en el Capítulo 12) no puede detectar el uso de `include()`, por lo que no se puede incluir en la cache el código del fragmento de forma independiente al de las plantillas. Symfony define 3 alternativas al uso de la instrucción `include()` y que permiten manejar de forma inteligente los fragmentos de código:

- Si el fragmento contiene poca lógica, se puede utilizar un archivo de plantilla al que se le pasan algunas variables. En este caso, se utilizan los elementos parciales (*partial*).
- Si la lógica es compleja (por ejemplo se debe acceder a los datos del modelo o se debe variar los contenidos en función de la sesión) es preferible separar la presentación de la lógica. En este caso, se utilizan componentes (*component*).
- Si el fragmento va a reemplazar una zona específica del layout, para la que puede que exista un contenido por defecto, se utiliza un *slot*.

Nota Existe otro tipo de fragmento de código, llamado "slot de componentes", que se utiliza cuando el fragmento depende del contexto (por ejemplo si el fragmento debe ser diferente para las acciones de un mismo módulo). Más tarde en este capítulo se explican los "slots de componentes".

Todos estos fragmentos se incluyen mediante los *helpers* del grupo llamado `Partial`. Estos *helpers* están disponibles en cualquier plantilla de Symfony sin necesidad de declararlos al principio.

7.2.1. Elementos parciales

Un elemento parcial es un trozo de código de plantilla que se puede reutilizar. Por ejemplo, en una aplicación de publicación, el código de plantilla que se encarga de mostrar un artículo se utiliza en la página de detalle del artículo, en la página que lista los mejores artículo y en la página que muestra los últimos artículos. Se trata de un código perfecto para definirlo como elemento parcial, tal y como muestra la figura 7-2.

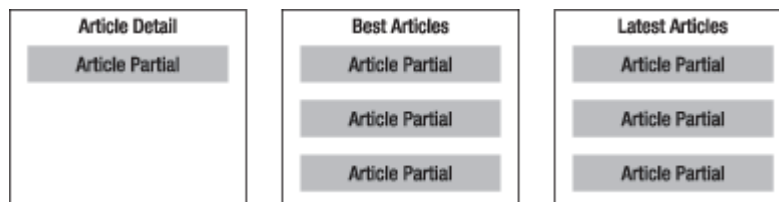


Figura 7.2. Reutilización de elementos parciales en las plantillas

Al igual que las plantillas, los elementos parciales son archivos que se encuentran en el directorio `templates/`, y que contienen código HTML y código PHP. El nombre del archivo de un elemento parcial siempre comienza con un guión bajo (`_`), lo que permite distinguir a los elementos parciales de las plantillas, ya que todos se encuentran en el mismo directorio `templates/`.

Una plantilla puede incluir elementos parciales independientemente de que estos se encuentren en el mismo módulo, en otro módulo o en el directorio global `templates/`. Los elementos parciales se incluyen mediante el *helper* `include_partial()`, al que se le pasa como parámetro el nombre del módulo y el nombre del elemento parcial (sin incluir el guión bajo del principio y la extensión `.php` del final), tal y como se muestra en el listado 7-7.

Listado 7-7 - Incluir elementos parciales en una plantilla del módulo `mimodulo`

```
// Incluir el elemento parcial de frontend/modules/mimodulo/templates/_miparcial1.php
// Como la plantilla y el elemento parcial están en el mismo módulo,
// se puede omitir el nombre del módulo
<?php include_partial('miparcial1') ?>

// Incluir el elemento parcial de frontend/modules/otromodulo/templates/_miparcial2.php
// En este caso es obligatorio indicar el nombre del módulo
<?php include_partial('otromodulo/miparcial2') ?>

// Incluir el elemento parcial de frontend/templates/_miparcial3.php
// Se considera que es parte del módulo 'global'
<?php include_partial('global/miparcial3') ?>
```

Los elementos parciales pueden acceder a los *helpers* y atajos de plantilla que proporciona Symfony. Pero como los elementos parciales se pueden llamar desde cualquier punto de la aplicación, no tienen acceso automático a las variables definidas por la acción que ha incluido la plantilla en la que se encuentra el elemento parcial, a no ser que se pase esas variables explícitamente en forma de parámetro. Si por ejemplo se necesita que un elemento parcial tenga acceso a una variable llamada `$total`, la acción pasa esa variable a la plantilla y después la plantilla se la pasa al *helper* como el segundo parámetro de la llamada a la función `include_partial()`, como se muestra en los listado 7-8, 7-9 y 7-10.

Listado 7-8 - La acción define una variable, en mimodulo/actions/actions.class.php

```
class mimoduloActions extends sfActions
{
    public function executeIndex()
    {
        $this->total = 100;
    }
}
```

Listado 7-9 - La plantilla pasa la variable al elemento parcial, en mimodulo/templates/indexSuccess.php

```
<p>ⓂHola Mundo!</p>
<?php include_partial('miparcial', array('mitotal' => $total)) ?>
```

Listado 7-10 - El elemento parcial ya puede usar la variable, en mimodulo/templates/_miparcial.php

```
<p>Total: <?php echo $mitotal ?></p>
```

Sugerencia Hasta ahora, todos los helpers se llamaban con la función `<?php echo nombreFuncion() ?>`. Por el contrario, el helper utilizado con los elementos parciales se llama mediante `<?php include_partial() ?>`, sin incluir el `echo`, para hacer su comportamiento más parecido a la instrucción de PHP `include()`. Si alguna vez se necesita obtener el contenido del elemento parcial sin mostrarlo, se puede utilizar la función `get_partial()`. Todos los helpers de tipo `include_` de este capítulo, tienen una función asociada que comienza por `get_` y que devuelve los contenidos que se pueden mostrar directamente con una instrucción `echo`.

Sugerencia Una de las novedades de Symfony 1.1 es que las acciones no sólo pueden acabar generando una plantilla, sino que también pueden finalizar generando un elemento parcial o un componente. Los métodos `renderPartial()` y `renderComponent()` de la clase de las acciones permite la reutilización del código. Además, esta técnica también puede hacer uso de la caché de los elementos parciales (ver capítulo 12). Las variables definidas en una acción se pasan de forma automática al elemento parcial y al componente, a menos que definas un array asociativo de variables y lo pases como segundo argumento del método.

```
public function executeMiAccion()
{
    $this->variable1 = 1234;
    $this->variable2 = 4567;

    return $this->renderPartial('mimodulo/miparcial');
}
```

En el código anterior, el elemento parcial tiene acceso a las variables `variable1` y `variable2`. Sin embargo, si la última instrucción de la acción es la siguiente:

```
return $this->renderPartial('mimodulo/miparcial', array('variable1' =>
    $this->variable1));
```

En este último caso, el elemento parcial sólo tiene acceso a la variable llamada `variable1`.

7.2.2. Componentes

En el Capítulo 2, el primer script de ejemplo se dividía en dos partes para separar la lógica de la presentación. Al igual que el patrón MVC se aplica a las acciones y las plantillas, es posible dividir un elemento parcial en su parte de lógica y su parte de presentación. En este caso, se necesitan los componentes.

Un componente es como una acción, solo que mucho más rápido. La lógica del componente se guarda en una clase que hereda de `sfComponents` y que se debe guardar en el archivo `action/components.class.php`. Su presentación se guarda en un elemento parcial. Los métodos de la clase `sfComponents` empiezan con la palabra `execute`, como sucede con las acciones, y pueden pasar variables a su presentación de la misma forma en la que se pasan variables en las acciones. Los elementos parciales que se utilizan como presentación de un componente, se deben llamar igual que los componentes, sustituyendo la palabra `execute` por un guión bajo. La tabla 7-1 compara las convenciones en los nombres de las acciones y los componentes.

Tabla 7-1. Convenciones en el nombrado de las acciones y de los componentes

Convención	Acciones	Componentes
Archivo de la lógica	<code>actions.class.php</code>	<code>components.class.php</code>
Clase de la que hereda la lógica	<code>sfActions</code>	<code>sfComponents</code>
Nombre de los métodos	<code>executeMiAccion()</code>	<code>executeMiComponente()</code>
Nombre del archivo de presentación	<code>miAccionSuccess.php</code>	<code>_miComponente.php</code>

Sugerencia De la misma forma que es posible separar los archivos de las acciones, la clase `sfComponents` dispone de una equivalente llamada `sfComponent` y que permite crear archivos individuales para cada componente siguiendo una sintaxis similar.

Por ejemplo, se puede definir una zona lateral que muestra las últimas noticias de un determinado tema que depende del perfil del usuario y que se va a reutilizar en varias páginas. Las consultas necesarias para mostrar las noticias son demasiado complejas como para incluirlas en un elemento parcial, por lo que se deben incluir en un archivo similar a las acciones, es decir, en un componente. La figura 7-3 ilustra este ejemplo.

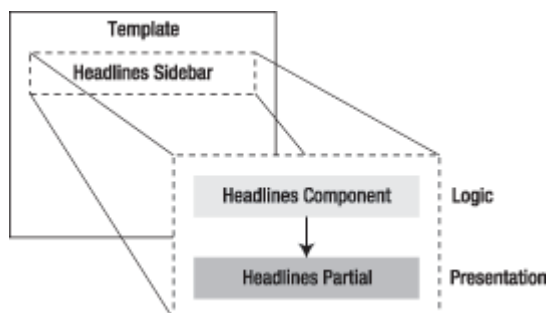


Figura 7.3. Uso de componentes en las plantillas

En este ejemplo, mostrado en los listados 7-11 y 7-12, el componente se define en su propio módulo (llamado `news`), pero se pueden mezclar componentes y acciones en un único módulo, siempre que tenga sentido hacerlo desde un punto de vista funcional.

Listado 7-11 - La clase de los componentes, en modules/news/actions/components.class.php

```

<?php

class newsComponents extends sfComponents
{
    public function executeHeadlines()
    {
        $c = new Criteria();
        $c->addDescendingOrderByColumn(NewsPeer::PUBLISHED_AT);
        $c->setLimit(5);
        $this->news = NewsPeer::doSelect($c);
    }
}

```

Listado 7-12 - El elemento parcial, en modules/news/templates/_headlines.php

```

<div>
    <h1>Últimas noticias</h1>
    <ul>
        <?php foreach($news as $headline): ?>
            <li>
                <?php echo $headline->getPublishedAt() ?>
                <?php echo link_to($headline->getTitle(), 'news/show?id='.$headline->getId())
            ?>
            </li>
        <?php endforeach ?>
    </ul>
</div>

```

Ahora, cada vez que se necesite el componente en una plantilla, se puede incluir de la siguiente forma:

```

| <?php include_component('news', 'headlines') ?>

```

Al igual que sucede con los elementos parciales, se pueden pasar parámetros adicionales a los componentes mediante un array asociativo. Dentro del elemento parcial se puede acceder directamente a los parámetros mediante su nombre y en el componente se puede acceder a ellos mediante el uso de `$this`. El listado 7-13 muestra un ejemplo.

Listado 7-13 - Paso de parámetros a un componente y a su plantilla

```

// Llamada al componente
<?php include_component('news', 'headlines', array('parametro' => 'valor')) ?>

// Dentro del componente
echo $this->parametro;
=> 'valor'

// Dentro del elemento parcial _headlines.php
echo $parametro;
=> 'valor'

```

Se pueden incluir componentes dentro de otros componentes y también en el layout global como si fuera una plantilla normal. Al igual que en las acciones, los métodos `execute` de los componentes pueden pasar variables a sus elementos parciales relacionados y pueden tener acceso a los mismos atajos. Pero las similitudes se quedan solo en eso. Los componentes no pueden manejar la seguridad ni la validación, no pueden ser llamados desde Internet (solo desde la propia aplicación) y no tienen distintas posibilidades para devolver sus resultados. Por este motivo, los componentes son más rápidos que las acciones.

7.2.3. Slots

Los elementos parciales y los componentes están especialmente diseñados para reutilizar código. Sin embargo, en muchas ocasiones se necesitan fragmentos de código que rellenen un layout con más de una zona variable. Por ejemplo se puede necesitar añadir etiquetas personalizadas en la sección `<head>` del layout en función del contenido de la acción. También se puede dar el caso de un layout que tiene una zona de contenidos dinámicos que se rellena con el resultado de la acción y muchas otras zonas pequeñas que tienen un contenido por defecto definido en el layout pero que puede ser modificado en la plantilla.

En los casos descritos anteriormente la solución más adecuada es un *slot*. Básicamente, un slot es una zona que se puede definir en cualquier elemento de la vista (layout, plantilla o elemento parcial). La forma de rellenar esa zona es similar a establecer el valor de una variable. El código de relleno se almacena de forma global en la respuesta, por lo que se puede definir en cualquier sitio (layout, plantilla o elemento parcial). Se debe definir un slot antes de utilizarlo y también hay que tener en cuenta que el layout se ejecuta después de la plantilla (durante el proceso de *decoración*) y que los elementos parciales se ejecutan cuando los llama una plantilla. Como todo esto suena demasiado abstracto, se va a ver su funcionamiento con un ejemplo.

Imagina que se dispone de un layout con una zona para la plantilla y 2 slots: uno para el lateral de la página y otro para el pie de página. El valor de los slots se define en la plantilla. Durante el proceso de decoración, el layout integra en su interior el código de la plantilla, por lo que los slots se rellenan con los valores que se han definido anteriormente, tal y como muestra la figura 7-4. De esta forma, el lateral y el pie de página pueden depender de la acción. Se puede aproximar a la idea de tener un layout con uno o más agujeros que se rellenan con otro código.

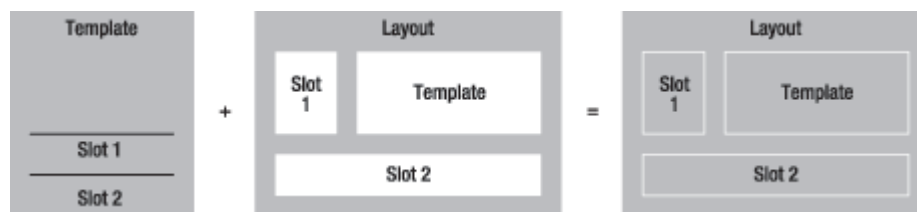


Figura 7.4. La plantilla define el valor de los slots del layout

Su funcionamiento se puede comprender mejor viendo algo de código. Para incluir un slot se utiliza el *helper* `include_slot()`. El *helper* `has_slot()` devuelve un valor `true` si el slot ya ha sido definido antes, permitiendo de esta forma establecer un mecanismo de protección frente a errores. El listado 7-14 muestra como definir la zona para el slot lateral en el layout y su contenido por defecto.

Listado 7-14 - Incluir un slot llamado lateral en el layout

```

<div id="lateral">
  <?php if (has_slot('lateral')): ?>
    <?php include_slot('lateral') ?>
  <?php else: ?>
    <!-- código del lateral por defecto -->
    <h1>Zona cuyo contenido depende del contexto</h1>
    <p>Esta zona contiene enlaces e información sobre
      el contenido principal de la página.</p>
  <?php endif; ?>
</div>

```

Como es muy común mostrar un contenido por defecto cuando un slot no está definido, el *helper* `include_slot` devuelve un valor *booleano* que indica si el slot está definido. El listado 7-15 muestra cómo utilizar este valor de retorno para simplificar el código.

Listado 7-15 - Redefiniendo el contenido del slot lateral en el layout

```

<div id="lateral">
  <?php if (!include_slot('lateral')): ?>
    <!-- contenido por defecto del lateral -->
    <h1>Zona de contenidos contextuales</h1>
    <p>En esta zona se muestran enlaces e información
      que dependen del contenido principal de la página</p>
  <?php endif; ?>
</div>

```

Las plantillas pueden definir los contenidos de un slot (e incluso los elementos parciales pueden hacerlo). Como los slots se definen para mostrar código HTML, Symfony proporciona métodos útiles para indicar ese código HTML: se puede escribir el código del slot entre las llamadas a las funciones `slot()` y `end_slot()`, como se muestra en el listado 7-16.

Listado 7-16 - Redefiniendo el contenido del slot lateral en la plantilla

```

// ...
<?php slot('lateral') ?>
  <!-- Código específico para el lateral de esta plantilla -->
  <h1>Detalles del usuario</h1>
  <p>Nombre: <?php echo $user->getName() ?></p>
  <p>Email: <?php echo $user->getEmail() ?></p>
<?php end_slot() ?>

```

El código incluido entre las llamadas a los *helpers* del slot se ejecutan en el contexto de las plantillas, por lo que tienen acceso a todas las variables definidas por la acción. Symfony añade de forma automática en el objeto `response` el resultado del código anterior. No se muestra directamente en la plantilla, sino que se puede acceder a su código mediante la llamada a la función `include_slot()`, como se muestra en el listado 7.14.

Los slots son muy útiles cuando se tienen que definir zonas que muestran contenido que depende del contexto de la página. También se puede utilizar para añadir código HTML al layout solo para algunas acciones. Por ejemplo, una plantilla que muestra la lista de las últimas noticias puede necesitar incluir un enlace a un canal RSS dentro de la sección `<head>` del layout. Esto se

puede conseguir añadiendo un slot llamado `feed` en el layout y que sea redefinido en la plantilla del listado de noticias.

Si el contenido de un slot es muy corto, como por ejemplo cuando se utilizan slots para mostrar el título de las páginas, se puede pasar todo el contenido como el segundo argumento de la llamada al método `slot()`, como muestra el listado 7-17.

Listado 7-17 - Utilizando el método `slot()` con contenidos cortos

```
| <?php slot('titulo', 'El título de la página') ?>
```

Los usuarios que trabajan con plantillas normalmente son diseñadores web, que no conocen muy bien el funcionamiento de Symfony y que pueden tener problemas para encontrar los fragmentos de plantilla, ya que pueden estar desperdigados por toda la aplicación. Los siguientes consejos pueden hacer más fácil su trabajo con las plantillas de Symfony.

En primer lugar, aunque los proyectos de Symfony contienen muchos directorios, todos los layouts, plantillas y fragmentos de plantillas son archivos que se encuentran en directorios llamados `templates/`. Por tanto, en lo que respecta a un diseñador web, la estructura de un proyecto queda reducida a:

```
miproyecto/
  apps/
    aplicacion1/
      templates/      # Layouts de la aplicacion 1
      modules/
        modulo1/
          templates/  # Plantillas y elementos parciales del modulo 1
        modulo2/
          templates/  # Plantillas y elementos parciales del modulo 2
        modulo3/
          templates/  # Plantillas y elementos parciales del modulo 3
```

El resto de directorios pueden ser ignorados por el diseñador.

Cuando se encuentren con una función del tipo `include_partial()`, los diseñadores web sólo tienen que preocuparse por el primer argumento de la función. La estructura del nombre de este argumento es `nombre_modulo/nombre_elemento_parcial`, lo que significa que el código se encuentra en el archivo `modules/nombre_modulo/templates/_nombre_elemento_parcial.php`.

En los helpers de tipo `include_component()`, el nombre del módulo y el nombre del elemento parcial son los dos primeros argumentos. Por lo demás, para empezar a diseñar plantillas de aplicaciones Symfony sólo es necesario tener una idea general sobre lo que son los helpers y cuales son los más utilizados en las plantillas.

7.3. Configuración de la vista

En Symfony, la vista está formada por dos partes:

- La presentación HTML del resultado de la acción (que se guarda en la plantilla, en el layout y en los fragmentos de plantilla)
- El resto, que incluye entre otros los siguientes elementos:

- Declaraciones <meta>: palabras clave (keywords), descripción (description), duración de la cache, etc.
- El título de la página: no solo es útil para los usuarios que tienen abiertas varias ventanas del navegador, sino que también es muy importante para que los buscadores indexen bien la página.
- Inclusión de archivos: de JavaScript y de hojas de estilos.
- Layout: algunas acciones necesitan un layout personalizado (ventanas emergentes, anuncios, etc.) o puede que no necesiten cargar ningún layout (por ejemplo en las acciones relacionadas con Ajax).

En la vista, todo lo que no es HTML se considera configuración de la propia vista y Symfony permite 2 formas de manipular esa configuración. La forma habitual es mediante el archivo de configuración `view.yml`. Se utiliza cuando los valores de configuración no dependen del contexto o de alguna consulta a la base de datos. Cuando se trabaja con valores dinámicos que cambian con cada acción, se recurre al segundo método para establecer la configuración de la vista: añadir los atributos directamente en el objeto `sfResponse` durante la acción.

Nota Si un mismo parámetro de configuración se establece mediante el objeto `sfResponse` y mediante el archivo `view.yml`, tiene preferencia el valor establecido mediante el objeto `sfResponse`.

7.3.1. El archivo `view.yml`

Cada módulo contiene un archivo `view.yml` que define las opciones de su propia vista. De esta forma, es posible definir en un único archivo las opciones de la vista para todo el módulo entero y las opciones para cada vista. Las claves de primer nivel en el archivo `view.yml` son el nombre de cada módulo que se configura. El listado 7-18 muestra un ejemplo de configuración de la vista.

Listado 7-18 - ejemplo de archivo `view.yml` de módulo

```
editSuccess:
  metas:
    title: Edita tu perfil

editError:
  metas:
    title: Error en la edición del perfil

all:
  stylesheets: [mi_estilo]
  metas:
    title: Mi sitio web
```

Sugerencia Se debe tener en cuenta que las claves principales del archivo `view.yml` son los nombres de las vistas, no los nombres de las acciones. Recuerda que el nombre de una vista se compone de un nombre de acción y un resultado de acción. Si por ejemplo la acción `edit` devuelve un valor igual a `sfView::SUCCESS` (o no devuelve nada, ya que este es el valor devuelto por defecto), el nombre de la vista sería `editSuccess`.

Las opciones por defecto para el módulo entero se definen bajo la clave `all`: en el archivo `view.yml` del módulo. Las opciones por defecto para todas las vistas de la aplicación se definen en el archivo `view.yml` de la aplicación. Una vez más, se tiene la configuración en cascada:

- En `apps/frontend/modules/mimodulo/config/view.yml`, las definiciones de cada vista solo se aplican a una vista y además sus valores tienen preferencia sobre las opciones generales del módulo.
- En `apps/frontend/modules/mimodulo/config/view.yml`, las definiciones bajo `all`: se aplican a todas las acciones del módulo y tienen preferencia sobre las definiciones de la aplicación.
- En `apps/frontend/config/view.yml`, las definiciones bajo `default`: se aplican a todos los módulos y todas las acciones de la aplicación.

Sugerencia Por defecto no existen los archivos `view.yml` de cada módulo. Por tanto la primera vez que se necesita configurar una opción a nivel de módulo, se debe crear un nuevo archivo llamado `view.yml` en el directorio `config/`.

Después de ver la plantilla por defecto en el listado 7-5 y un ejemplo de la respuesta generada en el listado 7-6, puede que te preguntes dónde se definen las cabeceras de la página. En realidad, las cabeceras salen de las opciones de configuración por defecto definidas en el archivo `view.yml` de la aplicación que se muestra en el listado 7-19.

Listado 7-19 - Archivo de configuración por defecto de la vista de la aplicación, en `apps/frontend/config/view.yml`

```
default:
  http_metas:
    content-type: text/html

  metas:
    #title:      symfony project
    #description: symfony project
    #keywords:   symfony, project
    #language:   en
    robots:     index, follow

  stylesheets:  [main]

  javascripts:  []

  has_layout:   on
  layout:       layout
```

Cada una de estas opciones se explica en detalle en la sección "Opciones de configuración de la vista".

7.3.2. El objeto respuesta (response)

Aunque el objeto `response` (objeto respuesta) es parte de la vista, normalmente se modifica en la acción. Las acciones acceden al objeto respuesta creado por Symfony, y llamado `sResponse`,

mediante el método `getResponse()`. El listado 7-20 muestra algunos de los métodos de `sfResponse` que se utilizan habitualmente en las acciones.

Listado 7-20 - Las acciones pueden acceder a los métodos del objeto `sfResponse`

```
class mimoduloActions extends sfActions
{
    public function executeIndex()
    {
        $respuesta = $this->getResponse();

        // Cabeceras HTTP
        $respuesta->setContentType('text/xml');
        $respuesta->setHttpHeader('Content-Language', 'en');
        $respuesta->setStatusCode(403);
        $respuesta->addVaryHttpHeader('Accept-Language');
        $respuesta->addCacheControlHttpHeader('no-cache');

        // Cookies
        $respuesta->setCookie($nombre, $contenido, $expiracion, $ruta, $dominio);

        // Atributos Meta y cabecera de La página
        $respuesta->addMeta('robots', 'NONE');
        $respuesta->addMeta('keywords', 'palabra1 palabra2');
        $respuesta->setTitle('Mi Página de Ejemplo');
        $respuesta->addStyleSheet('mi_archivo_css');
        $respuesta->addJavaScript('mi_archivo_javascript');
    }
}
```

Además de los métodos *setter* mostrados anteriormente para establecer el valor de las propiedades, la clase `sfResponse` también dispone de métodos *getter* que devuelven el valor de los atributos de la respuesta.

Los *setters* que establecen propiedades de las cabeceras de las páginas son uno de los puntos fuertes de Symfony. Como las cabeceras se envían lo más tarde posible (se envían en `sfRenderingFilter`) es posible modificar su valor todas las veces que sea necesario y tan tarde como haga falta. Además, incluyen atajos muy útiles. Por ejemplo, si no se indica el charset cuando se llama al método `setContentType()`, Symfony añade de forma automática el valor del charset definido en el archivo `settings.yml`.

```
$respuesta->setContentType('text/xml');
echo $respuesta->getContentType();
=> 'text/xml; charset=utf-8'
```

Los códigos de estado de las respuestas creadas por Symfony siguen la especificación de HTTP. De esta forma, los errores devuelven un código de estado igual a 500, las páginas que no se encuentran devuelven un código 404, las páginas normales devuelven el código 200, las páginas que no han sido modificadas se reducen a una simple cabecera con el código 304 (en el Capítulo 12 se explica con detalle), etc. Este comportamiento por defecto se puede redefinir para establecer códigos de estado personalizados, utilizando el método `setStatusCode()` sobre la

respuesta. Se puede especificar un código propio junto con un mensaje personalizado o solamente un código, en cuyo caso Symfony añade el mensaje más común para ese código.

```
| $respuesta->setStatusCode(404, 'Esta página no existe');
```

Sugerencia Symfony normaliza el nombre de las cabeceras antes de enviarlas. De esta forma, no es necesario preocuparse si se ha escrito `content-language` en vez de `Content-Language` cuando se utiliza el método `setHTTPHeader()`, ya que Symfony se encarga de transformar el primer nombre indicado en el segundo nombre, que es el correcto.

7.3.3. Opciones de configuración de la vista

Puede que hayas observado que existen 2 tipos diferentes de opciones para la configuración de la vista:

- Las opciones que tienen un único valor (el valor es una cadena de texto en el archivo `view.yml` y el objeto respuesta utiliza un método `set` para ellas)
- Las opciones que tienen múltiples valores (el archivo `view.yml` utiliza arrays para almacenar los valores y el objeto respuesta utiliza métodos de tipo `add`)

Hay que tener en cuenta por tanto que la configuración en cascada va sobrescribiendo los valores de las opciones de un solo valor y va añadiendo valores a las opciones que permiten valores múltiples. Este comportamiento se entiende mejor a medida que se avanza en este capítulo.

7.3.3.1. Configuración de las etiquetas <meta>

La información que almacenan las etiquetas <meta> de la respuesta no se muestra en el navegador, pero es muy útil para los buscadores. Además, permiten controlar la cache de cada página. Las etiquetas <meta> se pueden definir dentro de las claves `http_metas:` y `metas:` en el archivo `view.yml`, como se muestra en el listado 7-21, o utilizando los métodos `addHttpMeta()` y `addMeta()` del objeto respuesta dentro de la acción, como muestra el listado 7-22.

Listado 7-21 - Definir etiquetas <meta> en forma de clave: valor dentro del archivo `view.yml`

```
http_metas:
  cache-control: public

metas:
  description:  Página sobre economía en Francia
  keywords:    economía, Francia
```

Listado 7-22 - Definir etiquetas <meta> como opciones de la respuesta dentro de la acción

```
| $this->getResponse()->addHttpMeta('cache-control', 'public');
| $this->getResponse()->addMeta('description', 'Página sobre economía en Francia');
| $this->getResponse()->addMeta('keywords', 'economía, Francia');
```

Si se añade un nuevo valor a una clave que ya tenía establecido otro valor, se reemplaza el valor anterior por el nuevo valor establecido. Para las etiquetas <meta>, se puede añadir al método

`addHttpMeta()` (y también a `setHTTPHeader()`) un tercer parámetro con un valor de `false` para que añadan el valor indicado al valor que ya existía y así no lo reemplacen.

```
$this->getResponse()->addHttpMeta('accept-language', 'en');
$this->getResponse()->addHttpMeta('accept-language', 'fr', false);
echo $this->getResponse()->getHTTPHeader('accept-language');
=> 'en, fr'
```

Para añadir las etiquetas `<meta>` en la página que se envía al usuario, se deben utilizar los *helpers* `include_http metas()` y `include metas()` dentro de la sección `<head>` (que es por ejemplo lo que hace el layout por defecto, como se vio en el listado 7-5). Symfony construye las etiquetas `<meta>` definitivas juntando de forma automática el valor de todas las opciones de todos los archivos `view.yml` (incluyendo el archivo por defecto mostrado en el listado 7-18) y el valor de todas las opciones establecidas mediante los métodos de la respuesta. Por tanto, el ejemplo del listado 7-21 acaba generando las etiquetas `<meta>` del listado 7-23.

Listado 7-23 - Etiquetas `<meta>` que se muestran en la página final generada

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta http-equiv="cache-control" content="public" />
<meta name="robots" content="index, follow" />
<meta name="description" content="FPágina sobre economía en Francia" />
<meta name="keywords" content="economía, Francia" />
```

Como característica adicional, la cabecera HTTP de la respuesta incluye el contenido establecido en `http-metas` aunque no se utilice el *helper* `include_http metas()` en el layout o incluso cuando no se utiliza ningún layout. Por ejemplo, si se necesita enviar el contenido de una página como texto plano, se puede utilizar el siguiente archivo de configuración `view.yml`:

```
http_metas:
  content-type: text/plain

has_layout: false
```

7.3.3.2. Configuración del título

El título de las páginas web es un aspecto clave para los buscadores. Además, es algo muy cómodo para los navegadores modernos que incluyen la navegación con pestañas. En HTML, el título se define como una etiqueta y como parte de la metainformación de la página, así que en el archivo `view.yml` el título aparece como descendiente de la clave `metas`. El listado 7-24 muestra la definición del título en el archivo `view.yml` y el listado 7-25 muestra la definición en la acción.

Listado 7-24 - Definición del título en `view.yml`

```
indexSuccess:
  metas:
    title: Los tres cerditos
```

Listado 7-25 - Definición del título en la acción (es posible crear títulos dinámicamente)

```
$this->getResponse()->setTitle(sprintf('Los %d cerditos', $numero));
```

En la sección <head> del documento final, se incluye la etiqueta <meta name="title"> sólo si se utiliza el *helper* `include metas()`, y se incluye la etiqueta <title> sólo si se utiliza el *helper* `include_title()`. Si se utilizan los dos *helpers* (como se muestra en el layout por defecto del listado 7-5) el título aparece dos veces en el documento (como en el listado 7-6), algo que es completamente correcto.

7.3.3.3. Configuración para incluir archivos

Como se muestra en el listado 7-26, es muy sencillo añadir una hoja de estilos concreta o un archivo de JavaScript en la vista.

Listado 7-26 - Incluir archivos CSS y JavaScript

```
# En el archivo view.yml
indexSuccess:
  stylesheets: [miestilo1, miestilo2]
  javascripts: [miscript]
// En la acción
$this->getResponse()->addStylesheet('miestilo1');
$this->getResponse()->addStylesheet('miestilo2');
$this->getResponse()->addJavascript('miscript');

// En la plantilla
<?php use_stylesheet('miestilo1') ?>
<?php use_stylesheet('miestilo2') ?>
<?php use_javascript('miscript') ?>
```

En cualquier caso, el argumento necesario es el nombre del archivo. Si la extensión del archivo es la que le corresponde normalmente (.css para las hojas de estilos y .js para los archivos de JavaScript) se puede omitir la extensión. Si el directorio donde se encuentran los archivos también es el habitual (/css/ para las hojas de estilos y /js/ para los archivos de JavaScript) también se puede omitir. Symfony es lo bastante *inteligente* como para añadir la ruta y la extensión correcta.

Al contrario que lo que sucede en la definición de los elementos *meta* y *title*, no es necesario utilizar ningún *helper* en las plantillas o en el layout para incluir estos archivos. Por tanto, la configuración mostrada en los listados anteriores genera el código HTML mostrado en el listado 7-27, independientemente del contenido de la plantilla o del layout.

Listado 7-27 - Resultado de incluir los archivos - No es necesario llamar a ningún *helper* en el layout

```
<head>
...
<link rel="stylesheet" type="text/css" media="screen" href="/css/miestilo1.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/miestilo2.css" />
<script language="javascript" type="text/javascript" src="/js/miscript.js">
</script>
</head>
```

Nota Para incluir las hojas de estilo y los archivos JavaScript, se utiliza un filtro llamado `sfCommonFilter`. El filtro busca la etiqueta <head> de la respuesta y añade las etiquetas <link> y

<script> justo antes de cerrar la cabecera con la etiqueta </head>. Por tanto, no se pueden incluir este tipo de archivos si no existe una etiqueta <head> en el layout o en las plantillas.

Recuerda que se sigue aplicando la configuración en cascada, por lo que cualquier archivo que se incluya desde el archivo `view.yml` de la aplicación se muestra en cualquier página de la aplicación. Los listados 7-28, 7-29 y 7-30 muestran este funcionamiento.

Listado 7-28 - Ejemplo de archivo `view.yml` de aplicación

```
default:
  stylesheets: [principal]
```

Listado 7-29 - Ejemplo de archivo `view.yml` de módulo

```
indexSuccess:
  stylesheets: [especial]

all:
  stylesheets: [otra]
```

Listado 7-30 - Vista generada para la acción `indexSuccess`

```
<link rel="stylesheet" type="text/css" media="screen" href="/css/principal.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/otra.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/especial.css" />
```

Si no se quiere incluir un archivo definido en alguno de los niveles de configuración superiores, se puede añadir un signo - delante del nombre del archivo en la configuración de más bajo nivel, como se muestra en el listado 7-31.

Listado 7-31 - Ejemplo de archivo `view.yml` en el módulo y que evita incluir un archivo incluido desde el nivel de configuración de la aplicación

```
indexSuccess:
  stylesheets: [-principal, especial]

all:
  stylesheets: [otra]
```

Para eliminar todas las hojas de estilos o todos los archivos de JavaScript, se puede utilizar `-*` como nombre de archivo, tal y como muestra el listado 7-32.

Listado 7-32 - Ejemplo de archivo `view.yml` en el módulo que elimina todos los archivos incluidos desde el nivel de configuración de la aplicación

```
indexSuccess:
  stylesheets: [-*]
  javascripts: [-*]
```

Se puede ser todavía más preciso al incluir los archivos, ya que se puede utilizar un parámetro adicional para indicar la posición en la que se debe incluir el archivo (sólo se puede indicar la posición primera o la última). El listado 7-33 muestra un ejemplo de su uso para hojas de estilos, pero también funciona para los archivos JavaScript.

Listado 7-33 - Especificando la posición en la que se incluyen los archivos de hojas de estilos

```
# En el archivo view.yml
indexSuccess:
  stylesheets: [especial: { position: first }]
// En la acción
$this->getResponse()->addStylesheet('especial', 'first');

// En la plantilla
<?php use_stylesheet('especial', 'first') ?>
```

A partir de la versión 1.1 de Symfony también es posible evitar la modificación del nombre del archivo incluido, de forma que las etiquetas `<link>` y `<script>` resultantes muestren la misma ruta que se ha indicado, tal y como muestra el listado 7-34.

Listado 7-34 - Incluyendo hojas de estilos sin modificar su nombre

```
# En el archivo view.yml
indexSuccess:
  stylesheets: [principal, secundaria: { raw_name: true }]
// En la acción
$this->getResponse()->addStylesheet('secundaria', '', array('raw_name' => true));

// En la plantilla
<?php use_stylesheet('secundaria', '', array('raw_name' => true)) ?>

// La vista resultante de cualquiera de las instrucciones anteriores
<link rel="stylesheet" type="text/css" media="print" href="secundaria" />
```

Para modificar el atributo `media` de la hoja de estilos incluida, se pueden modificar las opciones por defecto de Symfony, como se muestra en el listado 7-35.

Listado 7-35 - Definir el atributo media al añadir una hoja de estilos

```
# En el archivo view.yml
indexSuccess:
  stylesheets: [principal, impresora: { media: print }]
// En la acción
$this->getResponse()->addStylesheet('impresora', '', array('media' => 'print'));

// En la plantilla
<?php use_stylesheet('impresora', '', array('media' => 'print')) ?>

// La vista resultante de cualquiera de las instrucciones anteriores
<link rel="stylesheet" type="text/css" media="print" href="/css/impresora.css" />
```

7.3.3.4. Configuración del layout

Dependiendo de la estructura gráfica del sitio web, pueden definirse varios layouts. Los sitios web clásicos tienen al menos dos layouts: el layout por defecto y el layout que muestran las ventanas emergentes.

Como se ha visto, el layout por defecto se define en `miproyecto/apps/frontend/templates/layout.php`. Los layouts adicionales también se definen en el mismo directorio `templates/`. Para

que una vista utilice un layout específico como por ejemplo `frontend/templates/mi_layout.php`, se debe utilizar la sintaxis del listado 7-36.

Listado 7-36 - Definición del layout

```
# En el archivo view.yml
indexSuccess:
  layout: mi_layout
// En la acción
$this->setLayout('mi_layout');

// En la plantilla
<?php decorate_with('mi_layout') ?>
```

Algunas vistas no requieren el uso de ningún layout (por ejemplo las páginas de texto y los canales RSS). En ese caso, se puede eliminar el uso del layout tal y como se muestra en el listado 7-37.

Listado 7-37 - Eliminar el layout

```
# En el archivo view.yml
indexSuccess:
  has_layout: false
// En la acción
$this->setLayout(false);

// En la plantilla
<?php decorate_with(false) ?>
```

Nota Las vistas de las acciones que utilizan Ajax no tienen definido ningún layout por defecto.

7.4. Slots de componentes

Si se combina el poder de los componentes que se han visto anteriormente y las opciones de configuración de la vista, se consigue un modelo de desarrollo de la vista completamente nuevo: el sistema de slots de componentes. Se trata de una alternativa a los *slots* que se centra en la reutilización y en la separación en capas. De esta forma, los slots de componentes están mucho más estructurados que los *slots*, pero son un poco más lentos de ejecutar.

Al igual que los *slots*, los slots de componentes son zonas que se pueden definir en los elementos de la vista. La principal diferencia reside en la forma en la que se decide qué código rellena esas zonas. En un *slot* normal, el código se establece en otro elemento de la vista; en un slot de componentes, el código es el resultado de la ejecución de un componente, y el nombre de ese componente se obtiene de la configuración de la vista. Después de verlos en la práctica, es sencillo entender el comportamiento de los slots de componentes.

Para definir la zona del slot de componentes, se utiliza el *helper* `include_component_slot()`. El parámetro de esta función es el nombre que se asigna al slot de componentes. Imagina por ejemplo que el archivo `layout.php` de la aplicación tiene un lateral de contenidos cuya información depende de la página en la que se muestra. El listado 7-38 muestra como se incluiría este slot de componentes.

Listado 7-38 - Incluir un slot de componentes de nombre lateral

```
...
<div id="lateral">
  <?php include_component_slot('lateral') ?>
</div>
```

La correspondencia entre el nombre del slot de componentes y el nombre del propio componente se define en la configuración de la vista. Por ejemplo, se puede establecer el componente por defecto para el slot `lateral` debajo de la clave `components` del archivo `view.yml` de la aplicación. La clave de la opción de configuración es el nombre del slot de componentes; el valor de la opción es un array que contiene el nombre del módulo y el nombre del componente. El listado 7-29 muestra un ejemplo.

Listado 7-39 - Definir el slot de componentes por defecto para lateral, en frontend/config/view.yml

```
default:
  components:
    lateral: [mimodulo, default]
```

De esta forma, cuando se ejecuta el layout, el slot de componentes `lateral` se rellena con el resultado de ejecutar el método `executeDefault()` de la clase `mimoduloComponents` del módulo `mimodulo`, y este método utiliza la vista del elemento parcial `_default.php` que se encuentra en `modules/mimodulo/templates/`.

La configuración en cascada permite redefinir esta opción en cualquier módulo. Por ejemplo, en el módulo `user` puede ser más útil mostrar el nombre del usuario y el número de artículos que ha publicado. En ese caso, se puede particularizar el slot `lateral` mediante las siguientes opciones en el archivo `view.yml` del módulo, como se muestra en el listado 7-40.

Listado 7-40 - Particularizando el slot de componentes lateral, en frontend/modules/user/config/view.yml

```
all:
  components:
    lateral: [mimodulo, user]
```

El listado 7-41 muestra el código del componente necesario para este slot.

Listado 7-41 - Componentes utilizados por el slot lateral, en modules/mimodulo/actions/components.class.php

```
class mimoduloComponents extends sfComponents
{
    public function executeDefault()
    {
    }

    public function executeUser()
    {
        $this->usuario_actual = $this->getUser()->getCurrentUser();
        $c = new Criteria();
        $c->add(ArticlePeer::AUTHOR_ID, $this->usuario_actual->getId());
        $this->numero_articulos = ArticlePeer::doCount($c);
    }
}
```

```
| }
| }
```

El listado 7-42 muestra la vista de estos 2 componentes.

Listado 7-42 - Elementos parciales utilizados por el slot de componentes lateral, en modules/mimodulo/templates/

```
| // _default.php
| <p>El contenido de esta zona depende de la página en la que se muestra.</p>
|
| // _user.php
| <p>Nombre de usuario: <?php echo $usuario_actual->getName() ?></p>
| <p><?php echo $numero_articulos ?> artículos publicados</p>
```

Los slots de componentes se pueden utilizar para añadir en las páginas web las *"migas de pan"*, los menús de navegación que dependen de cada página y cualquier otro contenido que se deba insertar de forma dinámica. Como componentes que son, se pueden utilizar en el layout global y en cualquier plantilla, e incluso en otros componentes. La configuración que indica el componente de un slot siempre se extrae de la configuración de la última acción que se ejecuta.

Para evitar que se utilice un slot de componentes en un módulo determinado, se puede declarar un par módulo/componente vacío, tal y como muestra el listado 7-43.

Listado 7-43 - Deshabilitar un slot de componentes en view.yml

```
| all:
|   components:
|     lateral: []
```

7.5. Mecanismo de escape

Cuando se insertan datos generados dinámicamente en una plantilla, se debe asegurar la integridad de los datos. Por ejemplo, si se utilizan datos obtenidos mediante formularios que pueden rellenar usuarios anónimos, existe un gran riesgo de que los contenidos puedan incluir scripts y otros elementos maliciosos que se encargan de realizar ataques de tipo XSS (*cross-site scripting*). Por tanto, se debe aplicar un mecanismo de escape a todos los datos mostrados, de forma que ninguna etiqueta HTML pueda ser peligrosa.

Imagina por ejemplo que un usuario rellena un campo de formulario con el siguiente valor:

```
| <script>alert(document.cookie)</script>
```

Si se muestran directamente los datos, el navegador ejecuta el código JavaScript introducido por el usuario, que puede llegar a ser mucho más peligroso que el ejemplo anterior que simplemente muestra un mensaje. Por este motivo, se deben aplicar mecanismos de escape a los valores introducidos antes de mostrarlos, para que se transformen en algo como:

```
| &lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

Los datos se pueden escapar manualmente utilizando la función `htmlspecialchars()` de PHP, pero es un método demasiado repetitivo y muy propenso a cometer errores. En su lugar, Symfony incluye un sistema conocido como *mecanismo de escape de los datos* que se aplica a

todos los datos mostrados mediante las variables de las plantillas. El mecanismo se activa mediante un único parámetro en el archivo `settings.yml` de la aplicación.

7.5.1. Activar el mecanismo de escape

El mecanismo de escape de datos se configura de forma global para toda la aplicación en el archivo `settings.yml`. El sistema de escape se controla con 2 parámetros: la estrategia (`escaping_strategy`) define la forma en la que las variables están disponibles en la vista y el método (`escaping_method`) indica la función que se aplica a los datos.

En principio, lo único necesario para activar el mecanismo de escape es establecer para la opción `escaping_strategy` el valor `on` en vez de su valor por defecto `off`, tal y como muestra el listado 7-44.

Listado 7-44 - Activar el mecanismo de escape, en `frontend/config/settings.yml`

```
all:
  .settings:
    escaping_strategy: on
    escaping_method:   ESC_SPECIALCHARS
```

Esta configuración aplica la función `htmlspecialchars()` a los datos de todas las variables mostradas. Si se define una variable llamada `prueba` en la acción con el siguiente contenido:

```
| $this->prueba = '<script>alert(document.cookie)</script>';
```

Con el sistema de escape activado, al mostrar esta variable en una plantilla, se mostrarán los siguientes datos:

```
| echo $prueba;
=> &lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

Además, todas las plantillas tienen acceso a una variable llamada `$sf_data`, que es un objeto contenedor que hace referencia a todas las variables a las que se les ha aplicado el mecanismo de escape. Por lo tanto, se puede acceder al valor de la variable `prueba` mediante la siguiente instrucción:

```
| echo $sf_data->get('prueba');
=> &lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

Sugerencia El objeto `$sf_data` implementa la interfaz `Array`, por lo que en vez de utilizar la sintaxis `$sf_data->get('mivariable')`, se puede obtener la variable mediante `$sf_data['mivariable']`. Sin embargo, no se trata realmente de un array, por lo que no se pueden utilizar funciones como por ejemplo `print_r()`.

La variable `$sf_data` también da acceso a los datos originales o *datos en crudo* de la variable. Se trata de una opción muy útil por ejemplo cuando la variable contiene código HTML que se quiere incluir directamente en el navegador para que sea interpretado en vez de mostrado (solo se debería utilizar esta opción si se confía plenamente en el contenido de esa variable). Para acceder a los datos originales se puede utilizar el método `getRaw()`.

```
| echo $sf_data->getRaw('prueba');
=> <script>alert(document.cookie)</script>
```


Si una variable almacena código HTML, cada vez que se necesita el código HTML original, es necesario acceder a sus datos originales, de forma que el código HTML se interprete y no se muestre en el navegador. Por este motivo el layout por defecto utiliza la instrucción `$sf_data->getRaw('sf_content')` para incluir el contenido de la plantilla, en vez de utilizar directamente el método `$sf_content`, que provocaría resultados no deseados cuando se activa el mecanismo de escape.

Cuando el valor de la opción `escaping_strategy` es `off`, la variable `$sf_data` también está disponible, pero en este caso siempre devuelve los datos originales de las variables.

Sugerencia La versión 1.0 de Symfony define otros dos valores para la opción `escaping_strategy`. El valor `bc` se convierte en el valor `off`, mientras que el valor `both` se convierte en `on`. Si utilizas cualquiera de esos valores, la aplicación no deja de funcionar, pero en los archivos de log se muestra un mensaje de error.

7.5.2. Los helpers útiles para el mecanismo de escape

Los *helpers* utilizados en el mecanismo de escape son funciones que devuelven el valor modificado correspondiente al valor que se les pasa. Se pueden utilizar como valor de la opción `escaping_method` en el archivo `settings.yml` o para especificar un método concreto de escape para los datos de una vista. Los *helpers* disponibles son los siguientes:

- `ESC_RAW`: no modifica el valor original.
- `ESC_SPECIALCHARS`: aplica la función `htmlspecialchars()` de PHP al valor que se le pasa.
- `ESC_ENTITIES`: aplica la función `htmlspecialchars()` de PHP al valor que se le pasa y utiliza la opción `ENT_QUOTES` para el estilo de las comillas.
- `ESC_JS`: modifica un valor que corresponde a una cadena de JavaScript que va a ser utilizada como HTML. Se trata de una opción muy útil para escapar valores cuando se emplea JavaScript para modificar de forma dinámica el contenido HTML de la página.
- `ESC_JS_NO_ENTITIES`: modifica un valor que va a ser utilizado en una cadena de JavaScript pero no le añade las entidades HTML correspondientes. Se trata de una opción muy útil para los valores que se van a mostrar en los cuadros de diálogo (por ejemplo para una variable llamada `miCadena` en la instrucción `javascript:alert(miCadena);`).

7.5.3. Aplicando el mecanismo de escape a los arrays y los objetos

No solo las cadenas de caracteres pueden hacer uso del mecanismo de escape, sino que también se puede aplicar a los arrays y los objetos. El mecanismo de escape se aplica en cascada a todos los arrays u objetos. Si la estrategia empleada es `on`, el listado 7-45 muestra el mecanismo de escape aplicado en cascada.

Listado 7-45 - El mecanismo de escape se puede aplicar a los arrays y los objetos

```
// Definición de la clase
class miClase
{
    public function pruebaCaracterEspecial($valor = '')
```

```

    {
        return '<'.$valor.'>';
    }
}

// En la acción
$this->array_prueba = array('&', '<', '>');
$this->array_de_arrays = array(array('&'));
$this->objeto_prueba = new miClase();

// En la plantilla
<?php foreach($array_prueba as $valor): ?>
    <?php echo $valor ?>
<?php endforeach; ?>
=> & < >
<?php echo $array_de_arrays[0][0] ?>
=> &
<?php echo $objeto_prueba->pruebaCaracterEspecial('&') ?>
=> &&&

```

De hecho, el tipo de las variables en la plantilla no es el tipo que le correspondería a la variable original. El mecanismo de escape *"decora"* las variables y las transforma en objetos especiales:

```

<?php echo get_class($array_prueba) ?>
=> sfOutputEscaperArrayDecorator
<?php echo get_class($objeto_prueba) ?>
=> sfOutputEscaperObjectDecorator

```

Esta es la razón por la que algunas funciones PHP habituales (como `array_shift()`, `print_r()`, etc.) no funcionan en los arrays a los que se ha aplicado el mecanismo de escape. No obstante, se puede seguir accediendo mediante `[]`, se pueden recorrer con `foreach` y proporcionan el dato correcto al utilizar la función `count()` (aunque `count()` solo funciona con la versión 5.2 o posterior de PHP). Como en las plantillas los datos (casi) siempre se acceden en *modo solo lectura*, la mayor parte de las veces se accede a los datos mediante los métodos que sí funcionan.

De todas formas, todavía es posible acceder a los datos originales mediante el objeto `$sf_data`. Además, los métodos de los objetos a los que se aplica el mecanismo de escape se modifican para que acepten un parámetro adicional: el método de escape. Así, se puede utilizar un método de escape diferente cada vez que se accede al valor de una variable en una plantilla, o incluso es posible utilizar el *helper* `ESC_RAW` para desactivar el sistema de escape para una variable concreta. El listado 7-46 muestra un ejemplo.

Listado 7-46 - Los métodos de los objetos a los que se aplica el mecanismo de escape aceptan un parámetro adicional

```

<?php echo $objeto_prueba->pruebaCaracterEspecial('&') ?>
=> &&&
// Las siguientes 3 líneas producen el mismo resultado
<?php echo $objeto_prueba->pruebaCaracterEspecial('&', ESC_RAW) ?>
<?php echo $sf_data->getRaw('objeto_prueba')->pruebaCaracterEspecial('&') ?>
<?php echo $sf_data->get('objeto_prueba', ESC_RAW)->pruebaCaracterEspecial('&') ?>
=> &&

```

Si se incluyen muchos objetos en las plantillas, el truco de añadir un parámetro adicional a los métodos se utiliza mucho, ya que es el método más rápido de obtener los datos originales al ejecutar el método.

Cuidado Las variables de Symfony también se modifican al activar el mecanismo de escape. Por tanto, las variables `$sf_user`, `$sf_request`, `$sf_param` y `$sf_context` siguen funcionando, pero sus métodos devuelven sus datos modificados, a no ser que se utilice la opción `ESC_RAW` como último argumento de las llamadas a los métodos.

Sugerencia Aunque los ataques de tipo XSS son una de las amenazas más habituales de los sitios web, no son la única. Los ataques CSRF también son muy populares, por lo que a partir de la versión 1.1 de Symfony también se ha incluido un mecanismo de protección contra los ataques CSRF. El capítulo 6 explica con más detalle el nuevo filtro CSRF.

7.6. Resumen

Existen numerosas herramientas y utilidades para manipular la capa correspondiente a la presentación. Las plantillas se pueden construir en pocos segundos, gracias al uso de los *helpers*. Los layouts, los elementos parciales, los componentes y los slots de componentes permiten aplicar los conceptos de modularidad y reutilización de componentes. La configuración de la vista aprovecha la velocidad de YAML para manejar la mayoría de cabeceras de las páginas. La configuración en cascada evita tener que definir todas las opciones para cada vista. Si una modificación de la presentación requiere el uso de datos dinámicos, se puede realizar la modificación en la acción mediante el objeto `sfResponse`. Además, la vista puede protegerse ante ataques de tipo XSS gracias al mecanismo de escape de los datos de las variables.

Capítulo 8. El modelo

Hasta ahora, la mayor parte de los contenidos se ha dedicado a la construcción de páginas y al procesamiento de peticiones y respuestas. Sin embargo, la lógica de negocio de las aplicaciones web depende casi siempre en su modelo de datos. El componente que se encarga por defecto de gestionar el modelo en Symfony es una capa de tipo ORM (*object/relational mapping*) realizada mediante el proyecto Propel (<http://propel.phpdb.org/>). En las aplicaciones Symfony, el acceso y la modificación de los datos almacenados en la base de datos se realiza mediante objetos; de esta forma nunca se accede de forma explícita a la base de datos. Este comportamiento permite un alto nivel de abstracción y permite una fácil portabilidad.

En este capítulo se explica como crear el modelo de objetos de datos, y la forma en la que se acceden y modifican los datos mediante Propel. Además, se muestra la integración de Propel en Symfony.

8.1. ¿Por qué utilizar un ORM y una capa de abstracción?

Las bases de datos son relacionales. PHP 5 y Symfony están orientados a objetos. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Como se explicó en el Capítulo 1, esta interfaz se llama ORM (*object-relational mapping*) o "*mapeo de objetos a bases de datos*", y está formada por objetos que permiten acceder a los datos y que contienen en sí mismos el código necesario para hacerlo.

La principal ventaja que aporta el ORM es la reutilización, permitiendo llamar a los métodos de un objeto de datos desde varias partes de la aplicación e incluso desde diferentes aplicaciones. La capa ORM también encapsula la lógica de los datos; como por ejemplo, el cálculo de la puntuación de un usuario de un foro en función de las aportaciones que ha realizado al foro y en función del éxito de esas aportaciones. Cuando una página quiere mostrar esa puntuación de un usuario, simplemente invoca un método del modelo de datos, sin preocuparse de cómo se realiza el cálculo. Si el método de cálculo sufre alguna variación, solo es necesario modificar el método que calcula la puntuación en el modelo, sin necesidad de modificar el resto de la aplicación.

La utilización de objetos en vez de registros y de clases en vez de tablas, tiene otra ventaja: permite añadir métodos accesorios en los objetos que no tienen relación directa con una tabla. Si se dispone por ejemplo de una tabla llamada `cliente` con dos campos llamados `nombre` y `apellidos`, puede que se necesite un dato llamado `NombreCompleto` que incluya y combine el nombre y los apellidos. En el mundo orientado a objetos, es tan fácil como añadir un método accesor a la clase `Cliente`, como se muestra en el listado 8-1. Desde el punto de vista de la aplicación, no existen diferencias entre los atributos `Nombre`, `Apellidos`, `NombreCompleto` de la clase `Cliente`. Solo la propia clase es capaz de determinar si un atributo determinado se corresponde con una columna de la base de datos.

Listado 8-1 - Los métodos accesorios en la clase del modelo permiten ocultar la estructura real de la tabla de la base de datos

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

Todo el código repetitivo de acceso a los datos y toda la lógica de negocio de los propios datos se puede almacenar en esos objetos. Imagina que se ha definido la clase `CarritoCompra` en la que se almacenan `Productos` (que son objetos). Para obtener el precio total del carrito de la compra antes de realizar el pago, se puede crear un método que encapsula el proceso de cálculo, tal y como se muestra en el listado 8-2.

Listado 8-2 - Los métodos accesoros ocultan la lógica de los datos

```
public function getTotal()
{
    $total = 0;
    foreach ($this->getProductos() as $producto)
    {
        $total += $producto->getPrecio() * $producto->getCantidad();
    }

    return $total;
}
```

Existe otra consideración importante que hay que tener en cuenta cuando se crean elementos de acceso a los datos: las empresas que crean las bases de datos utilizan variantes diferentes del lenguaje SQL. Si se cambia a otro sistema gestor de bases de datos, es necesario reescribir parte de las consultas SQL que se definieron para el sistema anterior. Si se crean las consultas mediante una sintaxis independiente de la base de datos y un componente externo se encarga de traducirlas al lenguaje SQL concreto de la base de datos, se puede cambiar fácilmente de una base de datos a otra. Este es precisamente el objetivo de las capas de abstracción de bases de datos. Esta capa obliga a utilizar una sintaxis específica para las consultas y a cambio realiza el trabajo sucio de optimizar y adaptar el lenguaje SQL a la base de datos concreta que se está utilizando.

La principal ventaja de la capa de abstracción es la portabilidad, porque hace posible el cambiar la aplicación a otra base de datos, incluso en mitad del desarrollo de un proyecto. Si se debe desarrollar rápidamente un prototipo de una aplicación y el cliente no ha decidido todavía la base de datos que mejor se ajusta a sus necesidades, se puede construir la aplicación utilizando SQLite y cuando el cliente haya tomado la decisión, cambiar fácilmente a MySQL, PostgreSQL o Oracle. Solamente es necesario cambiar una línea en un archivo de configuración y todo funciona correctamente.

Symfony utiliza Propel como ORM y Propel utiliza Creole como capa de abstracción de bases de datos. Estos 2 componentes externos han sido desarrollados por el equipo de Propel, y están completamente integrados en Symfony, por lo que se pueden considerar una parte más del framework. Su sintaxis y sus convenciones, que se describen en este capítulo, se han adaptado de forma que difieran lo menos posible de las de Symfony.

Nota En una aplicación de Symfony, todas las aplicaciones comparten el mismo modelo. Esa es precisamente la razón de ser de los proyectos: una agrupación de aplicaciones que dependen de

un modelo común. Este es el motivo por el que el modelo es independiente de las aplicaciones y los archivos del modelo se guardan en el directorio `lib/model/` de la raíz del proyecto.

8.2. Esquema de base de datos de Symfony

Para crear el modelo de objetos de datos que utiliza Symfony, se debe traducir el modelo relacional de la base de datos a un modelo de objetos de datos. Para realizar ese mapeo o traducción, el ORM necesita una descripción del modelo relacional, que se llama "esquema" (*schema*). En el esquema se definen las tablas, sus relaciones y las características de sus columnas.

La sintaxis que utiliza Symfony para definir los esquemas hace uso del formato YAML. Los archivos `schema.yml` deben guardarse en el directorio `mi proyecto/config/`.

Nota Symfony también puede trabajar con el formato nativo de los esquemas en Propel, que está basado en XML. Más adelante en este capítulo se explican los detalles en la sección "Más allá del `schema.yml`: `schema.xml`".

8.2.1. Ejemplo de esquema

¿Cómo se traduce la estructura de una base de datos a un esquema? La mejor forma de entenderlo es mediante un ejemplo. En el ejemplo se supone que se tiene una base de datos de un blog con dos tablas: `blog_articulo` y `blog_comentario`, con la estructura que se muestra en la figura 8-1.

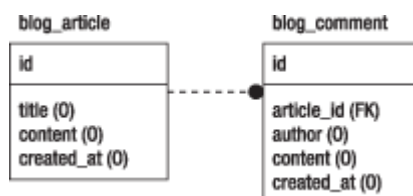


Figura 8.1. Estructura de tablas de la base de datos del blog

En este caso, el archivo `schema.yml` debería ser el del listado 8-3.

Listado 8-3 - Ejemplo de `schema.yml`

```

propel:
  blog_articulo:
    _attributes: { phpName: Articulo }
    id:
      titulo:      varchar(255)
      contenido:   longvarchar
      created_at:
    blog_comentario:
      _attributes: { phpName: Comentario }
      id:
      articulo_id:
      autor:       varchar(255)
      contenido:   longvarchar
      created_at:
  
```

Observa como el nombre de la propia base de datos (`blog`) no aparece en el archivo `schema.yml`. En su lugar, la base de datos se describe bajo el nombre de una conexión (`propel` en el ejemplo anterior). El motivo es que las opciones de conexión con la base de datos pueden depender del entorno en el que se está ejecutando la aplicación. Si se accede a la aplicación en el entorno de desarrollo, es posible que se acceda a la base de datos de desarrollo (por ejemplo `blog_dev`) pero con el mismo esquema que en la base de datos de producción. Las opciones de conexión con la base de datos se especifican en el archivo `databases.yml`, que se describe más adelante en este capítulo en la sección "Conexiones con la base de datos". El esquema no contiene ningún tipo de opción para la conexión a la base de datos, solo el nombre de la conexión, para mantener la abstracción de la base de datos.

8.2.2. Sintaxis básica de los esquemas

En el archivo `schema.yml`, la primera clave representa el nombre de la conexión. Puede contener varias tablas, cada una con varias columnas. Siguiendo la sintaxis de YAML, las claves terminan con dos puntos (`:`) y la estructura se define mediante la indentación con espacios, no con tabuladores.

Cada tabla puede definir varios atributos, incluyendo el atributo `phpName` (que es el nombre de la clase PHP que será generada para esa tabla). Si no se menciona el atributo `phpName` para una tabla, Symfony crea una clase con el mismo nombre que la tabla al que se aplica las normas del *camelCase*.

Sugerencia La convención *camelCase* elimina los guiones bajos de las palabras y pasa a mayúsculas la primera letra de cada palabra. Las versiones *camelCase* por defecto de `blog_articulo` y `blog_comentario` son `BlogArticulo` y `BlogComentario`. El nombre de esta convención para generar nombres viene del aspecto de las mayúsculas en una palabra larga, parecido a las jorobas de un camello.

Las tablas contienen columnas y el valor de las columnas se puede definir de 3 formas diferentes:

- Si no se indica nada, Symfony intenta adivinar los atributos más adecuados para la columna en función de su nombre y de una serie de convenciones que se explican en la sección "Columnas vacías" un poco más adelante en este Capítulo. Por ejemplo, en el listado 8-3 no es necesario definir la columna `id`. Symfony por defecto la trata como de tipo entero (*integer*), cuyo valor se auto-incrementa y además, clave principal de la tabla. En la tabla `blog_comentario`, la columna `articulo_id` se trata como una clave externa a la tabla `blog_articulo` (las columnas que acaban en `_id` se consideran claves externas, y su tabla relacionada se determina automáticamente en función de la primera parte del nombre de la columna). Las columnas que se llaman `created_at` automáticamente se consideran de tipo `timestamp`. Para este tipo de columnas, no es necesario definir su tipo. Esta es una de las razones por las que es tan fácil crear archivos `schema.yml`.
- Si sólo se define un atributo, se considera que es el tipo de columna. Symfony entiende los tipos de columna habituales: `boolean`, `integer`, `float`, `date`, `varchar(tamaño)`, `longvarchar` (que se convierte, por ejemplo, en tipo `text` en MySQL), etc. Para contenidos

de texto de más de 256 caracteres, se utiliza el tipo `longvarchar`, que no tiene tamaño definido (pero que no puede ser mayor que 65KB en MySQL). Los tipos `date` y `timestamp` tienen las limitaciones habituales de las fechas de Unix y no pueden almacenar valores anteriores al 1 de Enero de 1970. Como puede ser necesario almacenar fechas anteriores (por ejemplo para las fechas de nacimiento), existe un formato de fechas "*anteriores a Unix*" que son `bu_date` and `bu_timestamp`.

- Si se necesitan definir otros atributos a la columna (por ejemplo su valor por defecto, si es obligatorio o no, etc.), se indican los atributos como pares `clave: valor`. Esta sintaxis avanzada del esquema se describe más adelante en este capítulo.

Las columnas también pueden definir el atributo `phpName`, que es la versión modificada de su nombre según las convenciones habituales (Id, Título, Contenido, etc) y que normalmente no es necesario redefinir.

Las tablas también pueden definir claves externas e índices de forma explícita, además de incluir definiciones específicas de su estructura para ciertas bases de datos. En la sección "Sintaxis avanzada del esquema" se detallan estos conceptos.

8.3. Las clases del modelo

El esquema se utiliza para construir las clases del modelo que necesita la capa del ORM. Para reducir el tiempo de ejecución de la aplicación, estas clases se generan mediante una tarea de línea de comandos llamada `propel:build-model`.

```
| > php symfony propel:build-model
```

Sugerencia Después de construir el modelo, es necesario borrar la caché interna de Symfony mediante el comando `php symfony cc` para que Symfony sea capaz de encontrar los nuevos modelos.

Al ejecutar ese comando, se analiza el esquema y se generan las clases base del modelo, que se almacenan en el directorio `lib/model/om/` del proyecto:

- `BaseArticulo.php`
- `BaseArticuloPeer.php`
- `BaseComentario.php`
- `BaseComentarioPeer.php`

Además, se crean las verdaderas clases del modelo de datos en el directorio `lib/model/`:

- `Articulo.php`
- `ArticuloPeer.php`
- `Comentario.php`
- `ComentarioPeer.php`

Sólo se han definido dos tablas y se han generado ocho archivos. Aunque este hecho no es nada extraño, merece una explicación.

8.3.1. Clases base y clases personalizadas

¿Por qué es útil mantener dos versiones del modelo de objetos de datos en dos directorios diferentes?

Puede ser necesario añadir métodos y propiedades personalizadas en los objetos del modelo (piensa por ejemplo en el método `getNombreCompleto()` del listado 8-1). También es posible que a medida que el proyecto se esté desarrollando, se añadan tablas o columnas. Además, cada vez que se modifica el archivo `schema.yml` se deben regenerar las clases del modelo de objetos mediante el comando `propel-build-model`. Si se añaden los métodos personalizados en las clases que se generan, se borrarían cada vez que se vuelven a generar esas clases.

Las clases con nombre `Base` del directorio `lib/model/om/` son las que se generan directamente a partir del esquema. Nunca se deberían modificar esas clases, porque cada vez que se genera el modelo, se borran todas las clases.

Por otra parte, las clases de objetos propias que están en el directorio `lib/model` heredan de las clases con nombre `Base`. Estas clases no se modifican cuando se ejecuta la tarea `propel:build-model`, por lo que son las clases en las que se añaden los métodos propios.

El listado 8-4 muestra un ejemplo de una clase propia del modelo creada la primera vez que se ejecuta la tarea `propel:build-model`.

Listado 8-4 - Archivo de ejemplo de una clase del modelo, en `lib/model/Articulo.php`

```
class Articulo extends BaseArticulo
{
}
```

Esta clase hereda todos los métodos de la clase `BaseArticulo`, pero no le afectan las modificaciones en el esquema.

Este mecanismo de clases personalizadas que heredan de las clases base permite empezar a programar desde el primer momento, sin ni siquiera conocer el modelo relacional definitivo de la base de datos. La estructura de archivos creada permite personalizar y evolucionar el modelo.

8.3.2. Clases objeto y clases "peer"

`Articulo` y `Comentario` son clases objeto que representan un registro de la base de datos. Permiten acceder a las columnas de un registro y a los registros relacionados. Por tanto, es posible obtener el título de un artículo invocando un método del objeto `Articulo`, como se muestra en el listado 8-5.

Listado 8-5 - Las clases objeto disponen de *getters* para los registros de las columnas

```
$articulo = new Articulo();
// ...
$titulo = $articulo->getTitulo();
```

`ArticuloPeer` y `ComentarioPeer` son clases de tipo *"peer"*; es decir, clases que tienen métodos estáticos para trabajar con las tablas de la base de datos. Proporcionan los medios necesarios para obtener los registros de las tablas. Sus métodos devuelven normalmente un objeto o una colección de objetos de la clase objeto relacionada, como se muestra en el listado 8-6.

Listado 8-6 - Las clases "peer" contienen métodos estáticos para obtener registros de la base de datos

```
// $articulos es un array de objetos de la clase Articulo
$articulos = ArticuloPeer::retrieveByPk(array(123, 124, 125));
```

Nota Desde el punto de vista del modelo de datos, no puede haber objetos de tipo *"peer"*. Por este motivo los métodos de las clases *"peer"* se acceden mediante `::` (para invocarlos de forma estática), en vez del tradicional `->` (para invocar los métodos de forma tradicional).

La combinación de las clases objeto y las clases *"peer"* y las versiones básicas y personalizadas de cada una hace que se generen 4 clases por cada tabla del esquema. En realidad, existe una quinta clase que se crea en el directorio `lib/model/map/` y que contiene metainformación relativa a la tabla que es necesaria para la ejecución de la aplicación. Pero como es una clase que seguramente no se modifica nunca, es mejor olvidarse de ella.

8.4. Acceso a los datos

En Symfony, el acceso a los datos se realiza mediante objetos. Si se está acostumbrado al modelo relacional y a utilizar consultas SQL para acceder y modificar los datos, los métodos del modelo de objetos pueden parecer complicados. No obstante, una vez que se prueba el poder de la orientación a objetos para acceder a los datos, probablemente te gustará mucho más.

En primer lugar, hay que asegurarse de que se utiliza el mismo vocabulario. Aunque el modelo relacional y el modelo de objetos utilizan conceptos similares, cada uno tiene su propia nomenclatura:

| Relacional | Orientado a objetos |
|----------------|---------------------|
| Tabla | Clase |
| Fila, registro | Objeto |
| Campo, columna | Propiedad |

8.4.1. Obtener el valor de una columna

Cuando Symfony construye el modelo, crea una clase de objeto base para cada una de las tablas definidas en `schema.yml`. Cada una de estas clases contiene una serie de constructores y accesorios por defecto en función de la definición de cada columna: los métodos `new`, `getXXX()` y `setXXX()` permiten crear y obtener las propiedades de los objetos, como se muestra en el listado 8-7.

Listado 8-7 - Métodos generados en una clase objeto

```
$articulo = new Articulo();
$articulo->setTitulo('Mi primer artículo');
```

```
$articulo->setContenido('Este es mi primer artículo. \n Espero que te guste.');
```

```
$titulo    = $articulo->getTitulo();  
$contenido = $articulo->getContenido();
```

Nota La clase objeto generada se llama `Articulo`, que es el valor de la propiedad `phpName` para la tabla `blog_articulo`. Si no se hubiera definido la propiedad `phpName`, la clase se habría llamado `BlogArticulo`. Los métodos accesorios (`get` y `set`) utilizan una variante de *camelCase* aplicada al nombre de las columnas, por lo que el método `getTitulo()` obtiene el valor de la columna `titulo`.

Para establecer el valor de varios campos a la vez, se puede utilizar el método `fromArray()`, que también se genera para cada clase objeto, como se muestra en el listado 8-8.

Listado 8-8 - El método `fromArray()` es un *setter* múltiple

```
$articulo->fromArray(array(  
    'titulo'    => 'Mi primer artículo',  
    'contenido' => 'Este es mi primer artículo. \n Espero que te guste.'  
));
```

8.4.2. Obtener los registros relacionados

La columna `articulo_id` de la tabla `blog_comentario` define implícitamente una clave externa a la tabla `blog_articulo`. Cada comentario está relacionado con un artículo y un artículo puede tener muchos comentarios. Las clases generadas contienen 5 métodos que traducen esta relación a la forma orientada a objetos, de la siguiente manera:

- `$comentario->getArticulo()`: para obtener el objeto `Articulo` relacionado
- `$comentario->getArticuloId()`: para obtener el ID del objeto `Articulo` relacionado
- `$comentario->setArticulo($articulo)`: para definir el objeto `Articulo` relacionado
- `$comentario->setArticuloId($id)`: para definir el objeto `Articulo` relacionado a partir de un ID
- `$articulo->getComentarios()`: para obtener los objetos `Comentario` relacionados

Los métodos `getArticuloId()` y `setArticuloId()` demuestran que se puede utilizar la columna `articulo_id` como una columna normal y que se pueden indicar las relaciones manualmente, pero esto no es muy interesante. La ventaja de la forma orientada a objetos es mucho más evidente en los otros 3 métodos. El listado 8-9 muestra como utilizar los *setters* generados.

Listado 8-9 - Las claves externas se traducen en un *setter* especial

```
$comentario = new Comentario();  
$comentario->setAutor('Steve');  
$comentario->setContenido('Es el mejor artículo que he leído nunca!');
```

```
// Añadir este comentario al anterior objeto $articulo  
$comentario->setArticulo($articulo);
```

```
// Sintaxis alternativa
```

```
// Solo es correcta cuando el objeto artículo ya
// ha sido guardado anteriormente en la base de datos
$comentario->setArticuloId($articulo->getId());
```

El listado 8-10 muestra como utilizar los *getters* generados automáticamente. También muestra como encadenar varias llamadas a métodos en los objetos del modelo.

Listado 8-10 - Las claves externas se traducen en *getters* especiales

```
// Relación de "muchos a uno"
echo $comentario->getArticulo()->getTitulo();
=> Mi primer artículo
echo $comentario->getArticulo()->getContenido();
=> Este es mi primer artículo.
    Espero que te guste.

// Relación "uno a muchos"
$comentarios = $articulo->getComentarios();
```

El método `getArticulo()` devuelve un objeto de tipo `Articulo`, que permite utilizar el método accesor `getTitulo()`. Se trata de una alternativa mucho mejor que realizar la unión de las tablas manualmente, ya que esto último necesitaría varias líneas de código (empezando con la llamada al método `$comment->getArticuloId()`).

La variable `$comentarios` del listado 8-10 contiene un array de objetos de tipo `Comentario`. Se puede mostrar el primer comentario mediante `$comentarios[0]` o se puede recorrer la colección entera mediante `foreach ($comentarios as $comentario)`.

Nota Los objetos del modelo se definen siguiendo la convención de utilizar nombres en singular, y ahora se puede entender la razón. La clave externa definida en la tabla `blog_comentario` crea el método `getComentarios()`, cuyo nombre se crea añadiendo una letra `s` al nombre del objeto `Comentario`. Si el nombre del modelo fuera plural, la generación automática llamaría `getComentarioss()` a ese método, lo cual no tiene mucho sentido.

8.4.3. Guardar y borrar datos

Al utilizar el constructor `new` se crea un nuevo objeto, pero no un registro en la tabla `blog_articulo`. Si se modifica el objeto, tampoco se reflejan esos cambios en la base de datos. Para guardar los datos en la base de datos, se debe invocar el método `save()` del objeto.

```
| $articulo->save();
```

El ORM de Symfony es lo bastante inteligente como para detectar las relaciones entre objetos, por lo que al guardar el objeto `$articulo` también se guarda el objeto `$comentario` relacionado. También detecta si ya existía el objeto en la base de datos, por lo que el método `save()` a veces se traduce a una sentencia `INSERT` de SQL y otras veces se traduce a una sentencia `UPDATE`. La clave primaria se establece de forma automática al llamar al método `save()`, por lo que después de guardado, se puede obtener la nueva clave primaria del objeto mediante `$articulo->getId()`.

Sugerencia Para determinar si un objeto es completamente nuevo, se puede utilizar el método `isNew()`. Para detectar si un objeto ha sido modificado y por tanto se debe guardar en la base de datos, se puede utilizar el método `isModified()`.

Si lees los comentarios que insertan los usuarios en tus artículos, puede que te desanimes un poco para seguir publicando cosas en Internet. Si además no captas la ironía de los comentarios, puedes borrarlos fácilmente con el método `delete()`, como se muestra en el listado 8-11.

Listado 8-11 - Borrar registros de la base de datos mediante el método `delete()` del objeto relacionado

```
foreach ($articulo->getComentarios() as $comentario)
{
    $comentario->delete();
}
```

Sugerencia Después de ejecutar el método `delete()`, el objeto sigue disponible hasta que finaliza la ejecución de la petición actual. Para determinar si un objeto ha sido borrado de la base de datos, se puede utilizar el método `isDeleted()`.

8.4.4. Obtener registros mediante la clave primaria

Si se conoce la clave primaria de un registro concreto, se puede utilizar el método `retrieveByPk()` de la clase *peer* para obtener el objeto relacionado.

```
| $articulo = ArtículoPeer::retrieveByPk(7);
```

El archivo `schema.yml` define el campo `id` como clave primaria de la tabla `blog_articulo`, por lo que la sentencia anterior obtiene el artículo cuyo `id` sea igual a 7. Como se ha utilizado una clave primaria, solo se obtiene un registro; la variable `$articulo` contiene un objeto de tipo `Articulo`.

En algunos casos, la clave primaria está formada por más de una columna. Es esos casos, el método `retrieveByPK()` permite indicar varios parámetros, uno para cada columna de la clave primaria.

También se pueden obtener varios objetos a la vez mediante sus claves primarias, invocando el método `retrieveByPKs()`, que espera como argumento un array de claves primarias.

8.4.5. Obtener registros mediante Criterias

Cuando se quiere obtener más de un registro, se debe utilizar el método `doSelect()` de la clase *peer* correspondiente a los objetos que se quieren obtener. Por ejemplo, para obtener objetos de la clase `Articulo`, se llama al método `ArticuloPeer::doSelect()`.

El primer parámetro del método `doSelect()` es un objeto de la clase `Criterias`, que es una clase para definir consultas simples sin utilizar SQL, para conseguir la abstracción de base de datos.

Un objeto `Criterias` vacío devuelve todos los objetos de la clase. Por ejemplo, el código del listado 8-12 obtiene todos los artículos de la base de datos.

Listado 8-12 - Obtener registros mediante Criterias con el método `doSelect()` (Criterias vacío)

```

$c = new Criteria();
$articulos = ArtículoPeer::doSelect($c);

// Genera la siguiente consulta SQL
SELECT blog_articulo.ID, blog_articulo.TITLE, blog_articulo.CONTENIDO,
       blog_articulo.CREATED_AT
FROM   blog_articulo;

```

Invocar el método `::doSelect()` es mucho más potente que una simple consulta SQL. En primer lugar, se optimiza el código SQL para la base de datos que se utiliza. En segundo lugar, todos los valores pasados a `Criteria` se escapan antes de insertarlos en el código SQL, para evitar los problemas de *SQL injection*. En tercer lugar, el método devuelve un array de objetos y no un *result set*. El ORM crea y rellena de forma automática los objetos en función del *result set* de la base de datos. Este proceso se conoce con el nombre de *hydrating*.

Para las selecciones más complejas de objetos, se necesitan equivalentes a las sentencias `WHERE`, `ORDER BY`, `GROUP BY` y demás de SQL. El objeto `Criteria` dispone de métodos y parámetros para indicar todas estas condiciones. Por ejemplo, para obtener todos los comentarios escritos por el usuario Steve y ordenados por fecha, se puede construir un objeto `Criteria` como el del listado 8-13.

Listado 8-13 - Obtener registros mediante `Criteria` con el método `doSelect()` (`Criteria` con condiciones)

```

$c = new Criteria();
$c->add(ComentarioPeer::AUTOR, 'Steve');
$c->addAscendingOrderByColumn(ComentarioPeer::CREATED_AT);
$comentarios = ComentarioPeer::doSelect($c);

// Genera la siguiente consulta SQL
SELECT blog_comentario.ARTICULO_ID, blog_comentario.AUTOR, blog_comentario.CONTENIDO,
       blog_comentario.CREATED_AT
FROM   blog_comentario
WHERE  blog_comentario.autor = 'Steve'
ORDER BY blog_comentario.CREATED_AT ASC;

```

Las constantes de clase que se pasan como parámetros a los métodos `add()` hacen referencia a los nombres de las propiedades. Su nombre se genera a partir del nombre de las columnas en mayúsculas. Por ejemplo, para indicar la columna contenido de la tabla `blog_articulo`, se utiliza la constante de clase llamada `ArtículoPeer::CONTENIDO`.

Nota ¿Por qué se utiliza `ComentarioPeer::AUTOR` en vez de `blog_comentario.AUTOR`, que es en definitiva el valor que se va a utilizar en la consulta SQL? Supon que se debe modificar el nombre del campo de la tabla y en vez de autor ahora se llama *contributor*. Si se hubiera utilizado el valor `blog_comentario.AUTOR`, es necesario modificar ese valor en cada llamada al modelo. Sin embargo, si se utiliza el valor `ComentarioPeer::AUTOR`, solo es necesario cambiar el nombre de la columna en el archivo `schema.yml`, manteniendo el atributo `phpName` a un valor igual a `AUTOR` y reconstruir el modelo.

La tabla 8-1 compara la sintaxis de SQL y del objeto `Criteria`.

Tabla 8-1 - Sintaxis de SQL y del objeto `Criteria`

| SQL | Criteria |
|--|--|
| WHERE columna = valor | ->add(columna, valor); |
| WHERE columna <> valor | ->add(columna, valor, Criteria::NOT_EQUAL); |
| Otros operadores de comparación | |
| > , < | Criteria::GREATER_THAN,
Criteria::LESS_THAN |
| >=, <= | Criteria::GREATER_EQUAL,
Criteria::LESS_EQUAL |
| IS NULL, IS NOT NULL | Criteria::ISNULL, Criteria::ISNOTNULL |
| LIKE, ILIKE | Criteria::LIKE, Criteria::ILIKE |
| IN, NOT IN | Criteria::IN, Criteria::NOT_IN |
| Otras palabras clave de SQL | |
| ORDER BY columna ASC | ->addAscendingOrderByColumn(columna); |
| ORDER BY columna DESC | ->addDescendingOrderByColumn(columna); |
| LIMIT limite | ->setLimit(limite) |
| OFFSET desplazamiento | ->setOffset(desplazamiento) |
| FROM tabla1, tabla2 WHERE tabla1.col1 = tabla2.col2 | ->addJoin(col1, col2) |
| FROM tabla1 LEFT JOIN tabla2 ON tabla1.col1 = tabla2.col2 | ->addJoin(col1, col2, Criteria::LEFT_JOIN) |
| FROM tabla1 RIGHT JOIN tabla2 ON tabla1.col1 = tabla2.col2 | ->addJoin(col1, col2, Criteria::RIGHT_JOIN) |

Sugerencia La mejor forma de descubrir y entender los métodos disponibles en las clases generadas automáticamente es echar un vistazo a los archivos Base del directorio lib/model/om/. Los nombres de los métodos son bastante explícitos, aunque si se necesitan más comentarios sobre esos métodos, se puede establecer el parámetro `propel.builder.addComentarios` a `true` en el archivo de configuración `config/propel.ini` y después volver a reconstruir el modelo.

El listado 8-14 muestra otro ejemplo del uso de `Criteria` con condiciones múltiples. En el ejemplo se obtienen todos los comentarios del usuario Steve en los artículos que contienen la palabra `enjoy` y además, ordenados por fecha.

Listado 8-14 - Otro ejemplo para obtener registros mediante `Criteria` con el método `doSelect()` (`Criteria` con condiciones)

```
$c = new Criteria();
$c->add(ComentarioPeer::AUTOR, 'Steve');
$c->addJoin(ComentarioPeer::ARTICULO_ID, ArtículoPeer::ID);
$c->add(ArtículoPeer::CONTENIDO, '%enjoy%', Criteria::LIKE);
$c->addAscendingOrderByColumn(ComentarioPeer::CREATED_AT);
```

```
$comentarios = ComentarioPeer::doSelect($c);

// Genera la siguiente consulta SQL
SELECT blog_comentario.ID, blog_comentario.ARTICULO_ID, blog_comentario.AUTOR,
       blog_comentario.CONTENIDO, blog_comentario.CREATED_AT
FROM   blog_comentario, blog_articulo
WHERE  blog_comentario.AUTOR = 'Steve'
       AND blog_articulo.CONTENIDO LIKE '%enjoy%'
       AND blog_comentario.ARTICULO_ID = blog_articulo.ID
ORDER BY blog_comentario.CREATED_AT ASC
```

De la misma forma que el lenguaje SQL es sencillo pero permite construir consultas muy complejas, el objeto *Criteria* permite manejar condiciones de cualquier nivel de complejidad. Sin embargo, como muchos programadores piensan primero en el código SQL y luego lo traducen a las condiciones de la lógica orientada a objetos, es difícil comprender bien el objeto *Criteria* cuando se utiliza las primeras veces. La mejor forma de aprender es mediante ejemplos y aplicaciones de prueba. El sitio web del proyecto Symfony esá lleno de ejemplos de cómo construir objetos de tipo *Criteria* para todo tipo de situaciones.

Además del método `doSelect()`, todas las clases *peer* tienen un método llamado `doCount()`, que simplemente cuenta el número de registros que satisfacen las condiciones pasadas como parámetro y devuelve ese número como un entero. Como no se devuelve ningún objeto, no se produce el proceso de *hydrating* y por tanto el método `doCount()` es mucho más rápido que `doSelect()`.

Las clases *peer* también incluyen métodos `doDelete()`, `doInsert()` y `doUpdate()` (todos ellos requieren como parámetro un objeto de tipo *Criteria*). Estos métodos permiten realizar consultas de tipo DELETE, INSERT y UPDATE a la base de datos. Se pueden consultar las clases *peer* generadas automáticamente para descubrir más detalles de estos métodos de Propel.

Por último, si solo se quiere obtener el primer objeto, se puede reemplazar el método `doSelect()` por `doSelectOne()`. Es muy útil cuando se sabe que las condiciones de *Criteria* solo van a devolver un resultado, y su ventaja es que el método devuelve directamente un objeto en vez de un array de objetos.

Sugerencia Cuando una consulta `doSelect()` devuelve un número muy grande de resultados, normalmente sólo se quieren mostrar unos pocos en la respuesta. Symfony incluye una clase especial para paginar resultados llamada `sfPropelPager`, que realiza la paginación de forma automática y cuya documentación y ejemplos de uso se puede encontrar en http://www.symfony-project.org/cookbook/1_1/pager

8.4.6. Uso de consultas con código SQL

A veces, no es necesario obtener los objetos, sino que solo son necesarios algunos datos calculados por la base de datos. Por ejemplo, para obtener la fecha de creación de todos los artículos, no tiene sentido obtener todos los artículos y después recorrer el array de los resultados. En este caso es preferible obtener directamente el resultado, ya que se evita el proceso de *hydrating*.

Por otra parte, no deberían utilizarse instrucciones PHP de acceso a la base de datos, porque se perderían las ventajas de la abstracción de bases de datos. Lo que significa que se debe evitar el ORM (Propel) pero no la abstracción de bases de datos (Creole).

Para realizar consultas a la base de datos con Creole, es necesario realizar los siguientes pasos:

1. Obtener la conexión con la base de datos.
2. Construir la consulta.
3. Crear una sentencia con esa consulta.
4. Iterar el *result set* que devuelve la ejecución de la sentencia.

Aunque lo anterior puede parecer un galimatías, el código del listado 8-15 es mucho más explícito.

Listado 8-15 - Consultas SQL personalizadas con Creole

```
$conexion = Propel::getConnection();
$consulta = 'SELECT MAX(%s) AS max FROM %s';
$consulta = sprintf($consulta, ArtículoPeer::CREATED_AT, ArtículoPeer::TABLE_NAME);
$sentencia = $conexion->prepareStatement($consulta);
$resultset = $sentencia->executeQuery();
$resultset->next();
$max = $resultset->getInt('max');
```

Al igual que sucede con las selecciones realizadas con Propel, las consultas con Creole son un poco complicadas de usar al principio. Los ejemplos de las aplicaciones existentes y de los tutoriales pueden ser útiles para descubrir la forma de hacerlas.

Sugerencia Si se salta esa forma de acceder y se intenta acceder de forma directa a la base de datos, se corre el riesgo de perder la seguridad y la abstracción proporcionadas por Creole. Aunque es más largo hacerlo con Creole, es la forma de utilizar las buenas prácticas que aseguran un buen rendimiento, portabilidad y seguridad a la aplicación. Esta recomendación es especialmente útil para las consultas que contienen parámetros cuyo origen no es confiable (como por ejemplo un usuario de Internet). Creole se encarga de escapar los datos para mantener la seguridad de la base de datos. Si se accede directamente a la base de datos, se corre el riesgo de sufrir ataques de tipo *SQL-injection*.

8.4.7. Uso de columnas especiales de fechas

Normalmente, cuando una tabla tiene una columna llamada `created_at`, se utiliza para almacenar un timestamp de la fecha de creación del registro. La misma idea se aplica a las columnas `updated_at`, cuyo valor se debe actualizar cada vez que se actualiza el propio registro.

La buena noticia es que Symfony reconoce estos nombres de columna y se ocupa de actualizar su valor de forma automática. No es necesario establecer manualmente el valor de las columnas `created_at` y `updated_at`; se actualizan automáticamente, tal y como muestra el listado 8-16. Lo mismo se aplica a las columnas llamadas `created_on` y `updated_on`.

Listado 8-16 - Las columnas `created_at` y `updated_at` se gestionan automáticamente

```
$comentario = new Comentario();
$comentario->setAutor('Steve');
```

```
$comentario->save();

// Muestra la fecha de creación
echo $comentario->getCreatedAt();
=> [fecha de la operación INSERT de la base de datos]
```

Además, los *getters* de las columnas de fechas permiten indicar el formato de la fecha como argumento:

```
| echo $comentario->getCreatedAt('Y-m-d');
```

Cuando se desarrolla un proyecto con Symfony, normalmente se empieza escribiendo el código de la lógica de dominio en las acciones. Sin embargo, las consultas a la base de datos y la manipulación del modelo no se debería realizar en la capa del controlador. De forma que toda la lógica relacionada con los datos se debería colocar en la capa del modelo. Cada vez que se ejecuta el mismo código en más de un sitio de las acciones, se debería mover ese código al modelo. De esta forma las acciones se mantienen cortas y fáciles de leer.

Por ejemplo, imagina el caso de un blog que tiene que obtener los 10 artículos más populares relacionados con una etiqueta determinada (que se pasa como parámetro). Este código no debería estar en la acción, sino en el modelo. De hecho, si se tiene que mostrar en la plantilla la lista de artículos, la acción debería ser similar a la siguiente:

```
public function executeMuestraArticulosPopularesParaEtiqueta($peticion)
{
    $etiqueta = EtiquetaPeer::retrieveByName($peticion->getParameter('etiqueta'));
    $this->forward404Unless($etiqueta);
    $this->articulos = $etiqueta->getArticulosPopulares(10);
}
```

La acción crea un objeto de clase *Etiqueta* a partir del parámetro de la petición. Después, todo el código necesario para realizar la consulta a la base de datos se almacena en el método *getArticulosPopulares()* de esta clase. La acción es más fácil de leer y el código del modelo se puede reutilizar fácilmente en otra acción.

Mover el código a un lugar más apropiado es una de las técnicas de refactorización. Si se realiza habitualmente, el código resultante es más fácil de mantener y de entender por otros programadores. Una buena regla general sobre cuando refactorizar la capa de los datos es que el código de una acción raramente debería tener más de 10 líneas de código PHP.

8.5. Conexiones con la base de datos

Aunque el modelo de datos es independiente de la base de datos utilizada, es necesario utilizar una base de datos concreta. La información mínima que necesita Symfony para realizar peticiones a la base de datos es su nombre, las credenciales para acceder (usuario, contraseña) y el tipo de base de datos. Las opciones de conexión se pueden configurar pasando un DSN (*data source name*) a la tarea `configure:database`:

```
| > php symfony configure:database "mysql://login:password@localhost/blog"
```

Los datos de conexión dependen del entorno de ejecución. Se pueden definir diferentes opciones para los entornos prod, dev y test y para cualquier otro entorno de la aplicación utilizando la opción env:

```
| > php symfony --env=prod configure:database "mysql://login:password@localhost/blog"
```

Cada aplicación también puede redefinir esta configuración, lo que es útil cuando se quiere disponer de diferentes políticas de seguridad para el frontal y para la parte de administración de la aplicación. De esta forma, es posible disponer de diferentes usuarios de base de datos con privilegios diferentes para cada aplicación:

```
| > php symfony --app=frontend configure:database "mysql://login:password@localhost/blog"
```

En cada entorno también es posible definir varias conexiones diferentes. Cada conexión siempre hace referencia al esquema de datos del mismo nombre. El nombre de la conexión por defecto es propel, que hace referencia al esquema de datos llamado propel (por ejemplo el del listado 8-3). La opción name permite crear una nueva conexión:

```
| > php symfony --name=otraconexion configure:database "mysql://login:password@localhost/blog"
```

Las opciones de conexión con la base de datos también se pueden establecer manualmente en el archivo databases.yml que se encuentra en el directorio config/. El listado 8-17 muestra un ejemplo de ese archivo y el listado 8-18 muestra el mismo ejemplo con la notación extendida.

Listado 8-17 - Opciones abreviadas de conexión con la base de datos

```
| all:
|   propel:
|     class:          sfPropelDatabase
|     param:
|       dsn:          mysql://login:password@localhost/blog
```

Listado 8-18 - Ejemplo de opciones de conexión con la base de datos, en miproyecto/config/databases.yml

```
| prod:
|   propel:
|     param:
|       hostspec:      mi_servidor_datos
|       usuarioname:    mi_nombre_usuario
|       password:      xxxxxxxxxx
|
| all:
|   propel:
|     class:           sfPropelDatabase
|     param:
|       phptype:       mysql      # fabricante de la base de datos
|       hostspec:      localhost
|       database:      blog
|       usuarioname:    login
|       password:      passwd
|       port:          80
```

```

encoding:      utf8      # Codificación utilizada para crear la tabla
persistent:    true      # Utilizar conexiones persistentes

```

Los valores permitidos para el parámetro `phptype` corresponden a los tipos de bases de datos soportados por Creole:

- `mysql`
- `mssql`
- `pgsql`
- `sqlite`
- `oracle`

Los parámetros `hostspect`, `database`, `username` y `password` son las opciones típicas para conectar con una base de datos (servidor, base de datos, nombre de usuario y contraseña).

Si se utiliza una base de datos de tipo SQLite, el parámetro `hostspect` debe indicar la ruta al archivo de base de datos. Si por ejemplo se guarda la base de datos del blog en el archivo `data/blog.db`, las opciones del archivo `databases.yml` serán las del listado 8-19.

Listado 8-19 - Opciones de conexión con una base de datos SQLite utilizando la ruta al archivo como host

```

all:
  propel:
    class:      sfPropelDatabase
    param:
      phptype:  sqlite
      database: %SF_DATA_DIR%/blog.db

```

8.6. Extender el modelo

Los métodos del modelo que se generan automáticamente están muy bien, pero no siempre son suficientes. Si se incluye lógica de negocio propia, es necesario extender el modelo añadiendo nuevos métodos o redefiniendo algunos de los existentes.

8.6.1. Añadir nuevos métodos

Los nuevos métodos se pueden añadir en las clases vacías del modelo que se generan en el directorio `lib/model/`. Se emplea `$this` para invocar a los métodos del objeto actual y `self::` para invocar a los métodos estáticos de la clase actual. No se debe olvidar que las clases personalizadas heredan los métodos de las clases `Base` del directorio `lib/model/om/`.

Por ejemplo, en el objeto `Articulo` generado en el listado 8-3, se puede añadir un método mágico de PHP llamado `__toString()` de forma que al mostrar un objeto de la clase `Articulo` se muestre su título, tal y como se indica en el listado 8-20.

Listado 8-20 - Personalizar el modelo, en `lib/model/Articulo.php`

```
class Artículo extends BaseArticulo
{
    public function __toString()
    {
        return $this->getTitulo(); // getTitulo() se hereda de BaseArticulo
    }
}
```

También se pueden extender las clases *peer*, como por ejemplo para obtener todos los artículos ordenados por fecha de creación, tal y como muestra el listado 8-21.

Listado 8-21 - Personalizando el modelo, en lib/model/ArticuloPeer.php

```
class ArticuloPeer extends BaseArticuloPeer
{
    public static function getTodosOrdenadosPorFecha()
    {
        $c = new Criteria();
        $c->addAscendingOrderByColumn(self::CREATED_AT);
        return self::doSelect($c);
    }
}
```

Los nuevos métodos están disponibles de la misma forma que los métodos generados automáticamente, tal y como muestra el listado 8-22.

Listado 8-22 - El uso de métodos personalizados del modelo es idéntico al de los métodos generados automáticamente

```
foreach (ArticuloPeer::getTodosOrdenadosPorFecha() as $articulo)
{
    echo $articulo; // Se llama al método mágico __toString()
}
```

8.6.2. Redefinir métodos existentes

Si alguno de los métodos generados automáticamente en las clases Base no satisfacen las necesidades de la aplicación, se pueden redefinir en las clases personalizadas. Solamente es necesario mantener el mismo número de argumentos para cada método.

Por ejemplo, el método `$articulo->getComentarios()` devuelve un array de objetos Comentario, sin ningún tipo de ordenamiento. Si se necesitan los resultados ordenados por fecha de creación siendo el primero el comentario más reciente, se puede redefinir el método `getComentarios()`, como muestra el listado 8-23. Como el método `getComentarios()` original (guardado en `lib/model/om/BaseArticulo.php`) requiere como argumentos un objeto de tipo *Criteria* y una conexión, la nueva función debe contener esos mismos parámetros.

Listado 8-23 - Redefiniendo los métodos existentes en el modelo, en lib/model/Articulo.php

```
public function getComentarios($criteria = null, $con = null)
{
    if (is_null($criteria))
    {
```

```

        $criteria = new Criteria();
    }
    else
    {
        // Los objetos se pasan por referencia en PHP5, por lo que se debe clonar
        // el objeto original para no modificarlo
        $criteria = clone $criteria;
    }
    $criteria->addDescendingOrderByColumn(ComentarioPeer::CREATED_AT);

    return parent::getComentarios($criteria, $con);
}

```

El método personalizado acaba llamando a su método padre en la clase Base, lo que se considera una buena práctica. No obstante, es posible saltarse completamente la clase Base y devolver el resultado directamente.

8.6.3. Uso de comportamientos en el modelo

Algunas de las modificaciones que se realizan en el modelo son genéricas y por tanto se pueden reutilizar. Por ejemplo, los métodos que hacen que un objeto del modelo sea reordenable o un bloqueo de tipo *optimistic* en la base de datos para evitar conflictos cuando se guardan de forma concurrente los objetos se pueden considerar extensiones genéricas que se pueden añadir a muchas clases.

Symfony encapsula estas extensiones en "*comportamientos*" (del inglés *behaviors*). Los comportamientos son clases externas que proporcionan métodos extras a las clases del modelo. Las clases del modelo están definidas de forma que se puedan *enganchar* estas clases externas y Symfony extiende las clases del modelo mediante sfMixer (el Capítulo 17 contiene los detalles).

Para habilitar los comportamientos en las clases del modelo, se debe modificar una opción del archivo config/propel.ini:

```
| propel.builder.AddBehaviors = true      // El valor por defecto es false
```

Symfony no incluye por defecto ningún comportamiento, pero se pueden instalar mediante plugins. Una vez que el plugin se ha instalado, se puede asignar un comportamiento a una clase mediante una sola línea de código. Si por ejemplo se ha instalado el plugin sfPropelParanoidBehaviorPlugin en la aplicación, se puede extender la clase Artículo con este comportamiento añadiendo la siguiente línea de código al final del archivo Artículo.class.php:

```
| sfPropelBehavior::add('Articulo', array(
|     'paranoid' => array('column' => 'deleted_at')
| ));
```

Después de volver a generar el modelo, los objetos de tipo Artículo que se borren permanecerán en la base de datos, aunque serán invisibles a las consultas que hacen uso de los métodos del ORM, a no ser que se deshabilite temporalmente el comportamiento mediante sfPropelParanoidBehavior::disable().

Desde la versión 1.1 de Symfony también es posible declarar los comportamientos directamente en el archivo `schema.yml`, incluyéndolos bajo la clave `_behaviors` (ver listado 8-34 más adelante).

La lista de plugins de Symfony disponible en el wiki incluye numerosos comportamientos <http://trac.symfony-project.com/wiki/SymfonyPlugins#Behaviors>. Cada comportamiento tiene su propia documentación y su propia guía de instalación.

8.7. Sintaxis extendida del esquema

Un archivo `schema.yml` puede ser tan sencillo como el mostrado en el listado 8-3. Sin embargo, los modelos relacionales suelen ser complejos. Este es el motivo por el que existe una sintaxis extendida del esquema para que se pueda utilizar en cualquier caso.

8.7.1. Atributos

Se pueden definir atributos específicos para las conexiones y las tablas, tal y como se muestra en el listado 8-24. Estas opciones se establecen bajo la clave `_attributes`.

Listado 8-24 - Atributos de las conexiones y las tablas

```
propel:
  _attributes: { noXsd: false, defaultIdMethod: none, package: lib.model }
  blog_articulo:
    _attributes: { phpName: Articulo }
```

Si se quiere validar el esquema antes de que se genere el código asociado, se debe desactivar en la conexión el atributo `noXSD`. La conexión también permite que se le indique el atributo `defaultIdMethod`. Si no se indica, se utilizará el método nativo de generación de IDs --por ejemplo, `autoincrement` en MySQL o `sequences` en PostgreSQL. El otro valor permitido es `none`.

El atributo `package` es como un namespace; indica la ruta donde se guardan las clases generadas automáticamente. Su valor por defecto es `lib/model/`, pero se puede modificar para organizar el modelo en una estructura de subpaquetes. Si por ejemplo no se quieren mezclar en el mismo directorio las clases del núcleo de la aplicación con las clases de un sistema de estadísticas, se pueden definir dos esquemas diferentes con los paquetes `lib.model.business` y `lib.model.stats`.

Ya se ha visto el atributo de tabla `phpName`, que se utiliza para establecer el nombre de la clase generada automáticamente para manejar cada tabla de la base de datos.

Las tablas que guardan contenidos adaptados para diferentes idiomas (es decir, varias versiones del mismo contenido en una tabla relacionada, para conseguir la internacionalización) también pueden definir dos atributos adicionales (explicados detalladamente en el Capítulo 13), tal y como se muestra en el listado 8-25.

Listado 8-25 - Atributos para las tablas de internacionalización

```
propel:
  blog_articulo:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
```

Cada aplicación puede tener más de un esquema. Symfony tiene en cuenta todos los archivos que acaben en `schema.yml` o `schema.xml` que están en el directorio `config/`. Se trata de una estrategia muy útil cuando la aplicación tiene muchas tablas o si algunas de las tablas no comparten la misma conexión.

Si se consideran los dos siguientes esquemas:

```
// En config/business-schema.yml
propel:
  blog_articulo:
    _attributes: { phpName: Articulo }
    id:
    titulo: varchar(50)
// En config/stats-schema.yml
propel:
  estadisticas_visita:
    _attributes: { phpName: Visita }
    id:
    recurso: varchar(100)
    created_at:
```

Los dos esquemas comparten la misma conexión (`propel`), y las clases `Articulo` y `Visita` se generarán en el mismo directorio `lib/model/`. El resultado es equivalente a si se hubiera escrito solamente un esquema.

También es posible definir esquemas que utilicen diferentes conexiones (por ejemplo `propel` y `propel_bis` definidas en `databases.yml`) y cuyas clases generadas se guarden en subdirectorios diferentes:

```
# En config/business-schema.yml
propel:
  blog_articulo:
    _attributes: { phpName: Articulo, package: lib.model.business }
    id:
    titulo: varchar(50)
# En config/stats-schema.yml
propel_bis:
  estadisticas_visita:
    _attributes: { phpName: Visita, package: lib.model.stat }
    id:
    recurso: varchar(100)
    created_at:
```

Muchas aplicaciones utilizan más de un esquema. Sobre todo los plugins, muchos de los cuales definen su propio esquema y paquete para evitar errores y duplicidades con las clases propias de la aplicación (más detalles en el Capítulo 17).

8.7.2. Detalles de las columnas

La sintaxis básica ofrece dos posibilidades: dejar que Symfony deduzca las características de una columna a partir de su nombre (indicando un valor vacío para esa columna) o definir el tipo de columna con uno de los tipos predefinidos. El listado 8-26 muestra estas 2 opciones.

Listado 8-26 - Atributos básicos de columna

```
propel:
  blog_articulo:
    id:          # Symfony se encarga de esta columna
    titulo: varchar(50) # Definir el tipo explícitamente
```

Se pueden definir muchos más aspectos de cada columna. Si se definen, se utiliza un array asociativo para indicar las opciones de la columna, tal y como muestra el listado 8-27.

Listado 8-27 - Atributos avanzados de columna

```
propel:
  blog_articulo:
    id:      { type: integer, required: true, primaryKey: true, autoIncrement: true }
    name:    { type: varchar(50), default: foobar, index: true }
    group_id: { type: integer, foreignTable: db_group, foreignReference: id, onDelete:
cascade }
```

Los parámetros de las columnas son los siguientes:

- **type:** Tipo de columna. Se puede elegir entre `boolean`, `tinyint`, `smallint`, `integer`, `bigint`, `double`, `float`, `real`, `decimal`, `char`, `varchar(tamaño)`, `longvarchar`, `date`, `time`, `timestamp`, `bu_date`, `bu_timestamp`, `blob` y `clob`.
- **required:** valor booleano. Si vale `true` la columna debe tener obligatoriamente un valor.
- **default:** el valor por defecto.
- **primaryKey:** valor booleano. Si vale `true` indica que es una clave primaria.
- **autoIncrement:** valor booleano. Si se indica `true` para las columnas de tipo `integer`, su valor se auto-incrementará.
- **sequence:** el nombre de la secuencia para las bases de datos que utilizan secuencias para las columnas `autoIncrement` (por ejemplo PostgreSQL y Oracle).
- **index:** valor booleano. Si vale `true`, se construye un índice simple; si vale `unique` se construye un índice único para la columna.
- **foreignTable:** el nombre de una tabla, se utiliza para crear una clave externa a otra tabla.
- **foreignReference:** el nombre de la columna relacionada si las claves externas se definen mediante `foreignTable`.
- **onDelete:** determina la acción que se ejecuta cuando se borra un registro en una tabla relacionada. Si su valor es `setnull`, la columna de la clave externa se establece a `null`. Si su valor es `cascade`, se borra el registro relacionado. Si el sistema gestor de bases de datos no soporta este comportamiento, el ORM lo emula. Esta opción solo tiene sentido para las columnas que definen una `foreignTable` y una `foreignReference`.
- **isCulture:** valor booleano. Su valor es `true` para las columnas de tipo *culture* en las tablas de contenidos adaptados a otros idiomas (más detalles en el Capítulo 13).

8.7.3. Claves externas

Además de los atributos de columna `foreignTable` y `foreignReference`, es posible añadir claves externas bajo la clave `_foreignKeys`: de cada tabla. El esquema del listado 8-28 crea una clave externa en la columna `usuario_id`, que hace referencia a la columna `id` de la tabla `blog_usuario`.

Listado 8-28 - Sintaxis alternativa para las claves externas

```
propel:
  blog_articulo:
    id:
      titulo:      varchar(50)
      usuario_id: { type: integer }
      _foreignKeys:
        -
          foreignTable: blog_usuario
          onDelete:      cascade
          references:
            - { local: usuario_id, foreign: id }
```

La sintaxis alternativa es muy útil para las claves externas múltiples y para indicar un nombre a cada clave externa, tal y como muestra el listado 8-29.

Listado 8-29 - La sintaxis alternativa de las claves externas aplicada a una clave externa múltiple

```
_foreignKeys:
  mi_clave_externa:
    foreignTable: db_usuario
    onDelete:      cascade
    references:
      - { local: usuario_id, foreign: id }
      - { local: post_id, foreign: id }
```

8.7.4. Índices

Además del atributo de columna `index`, es posible añadir claves índices bajo la clave `_indexes`: de cada tabla. Si se quieren crear índices únicos, se debe utilizar la clave `_uniques`. En las columnas que requieren un tamaño, por ejemplo por ser columnas de texto, el tamaño del índice se indica entre paréntesis, de la misma forma que se indica el tamaño de cualquier columna. El listado 8-30 muestra la sintaxis alternativa para los índices.

Listado 8-30 - Sintaxis alternativa para los índices y los índices únicos

```
propel:
  blog_articulo:
    id:
      titulo:      varchar(50)
      created_at:
      _indexes:
        mi_indice: [titulo(10), usuario_id]
      _uniques:
        mi_otro_indice: [created_at]
```

La sintaxis alternativa solo es útil para los índices que se construyen con más de una columna.

8.7.5. Columnas vacías

Cuando Symfony se encuentra con una columna sin ningún valor, utiliza algo de *magia* para determinar su valor. El listado 8-31 muestra los detalles del tratamiento de las columnas vacías.

Listado 8-31 - Los detalles deducidos para las columnas vacías en función de su nombre

```
// Las columnas vacías llamadas "id" se consideran claves primarias
id: { type: integer, required: true, primaryKey: true, autoIncrement: true }

// Las columnas vacías llamadas "XXX_id" se consideran claves externas
loquesea_id: { type: integer, foreignTable: db_loquesea, foreignReference: id }

// Las columnas vacías llamadas created_at, updated_at, created_on y updated_on
// se consideran fechas y automáticamente se les asigna el tipo "timestamp"
created_at: { type: timestamp }
updated_at: { type: timestamp }
```

Para las claves externas, Symfony busca una tabla cuyo phpName sea igual al principio del nombre de la columna; si se encuentra, se utiliza ese nombre de tabla como foreignTable.

8.7.6. Tablas i18n

Symfony permite internacionalizar los contenidos mediante tablas relacionadas. De esta forma, cuando se dispone de contenido que debe ser internacionalizado, se guarda en 2 tablas distintas: una contiene las columnas invariantes y otra las columnas que permiten la internacionalización.

Todo lo anterior se considera de forma implícita cuando en el archivo schema.yml se dispone de una tabla con el nombre cualquiernombre_i18n. Por ejemplo, el esquema que muestra el listado 8-32 se completa automáticamente con los atributos de columna y de tabla necesarios para que funcione el mecanismo de internacionalización. De forma interna, Symfony entiende ese listado como si se hubiera escrito tal y como se muestra en el listado 8-33. El Capítulo 13 explica en detalle la internacionalización.

Listado 8-32 - Mecanismo i18n implícito

```
propel:
  db_group:
    id:
    created_at:

  db_group_i18n:
    name: varchar(50)
```

Listado 8-33 - Mecanismo i18n explícito

```
propel:
  db_group:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
    id:
    created_at:
```

```

db_group_i18n:
  id:      { type: integer, required: true, primaryKey: true, foreignTable:
db_group, foreignReference: id, onDelete: cascade }
  culture: { isCulture: true, type: varchar(7), required: true, primaryKey: true }
  name:    varchar(50)

```

8.7.7. Comportamientos

Los comportamientos son plugins que modifican el modelo de datos para añadir nuevas funcionalidades a las clases de Propel. El capítulo 17 explica los comportamientos en detalle. Desde la versión 1.1 de Symfony también es posible definir los comportamientos directamente en el esquema. Para ello, se añade su nombre y sus opciones bajo la clave `_behaviors` de cada tabla. El listado 8-34 muestra un ejemplo que extiende la clase `BlogArticulo` con el comportamiento llamado `paranoid`.

Listado 8-34 - Declarando los comportamientos

```

propel:
  blog_articulo:
    titulo:      varchar(50)
    _behaviors:
      paranoid:  { column: deleted_at }

```

8.7.8. Más allá del `schema.yml`: `schema.xml`

En realidad, el formato de `schema.yml` es propio de Symfony. Cuando se ejecuta un comando que empieza por `propel-`, Symfony transforma ese archivo en otro archivo llamado `generated-schema.xml`, que es el tipo de archivo que necesita Propel para realizar sus tareas sobre el modelo.

El archivo `schema.xml` contiene la misma información que su equivalente en formato YAML. Por ejemplo, el listado 8-3 se convierte en el archivo XML del listado 8-35.

Listado 8-35 - Ejemplo de `schema.xml`, que se corresponde con el listado 8-3

```

<?xml version="1.0" encoding="UTF-8"?>
<database name="propel" defaultIdMethod="native" noXsd="true" package="lib.model">
  <table name="blog_articulo" phpName="Articulo">
    <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
    <column name="titulo" type="varchar" size="255" />
    <column name="contenido" type="longvarchar" />
    <column name="created_at" type="timestamp" />
  </table>
  <table name="blog_comentario" phpName="Comentario">
    <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
    <column name="articulo_id" type="integer" />
    <foreign-key foreignTable="blog_articulo">
      <reference local="articulo_id" foreign="id"/>
    </foreign-key>
    <column name="autor" type="varchar" size="255" />
    <column name="contenido" type="longvarchar" />
  </table>
</database>

```

```
<column name="created_at" type="timestamp" />
</table>
</database>
```

La descripción del formato `schema.xml` se puede consultar en la documentación y la sección *"Getting started"* del sitio web del proyecto Propel (http://propel.phpdb.org/docs/usuario_guide/chapters/appendices/AppendixB-SchemaReference.html).

El formato del esquema en YAML se diseñó para que los esquemas fueran fáciles de leer y escribir, pero su inconveniente es que los esquemas más complejos no se pueden describir solamente con un archivo `schema.yml`. Por otra parte, el formato XML permite describir completamente el esquema, independientemente de su complejidad e incluye la posibilidad de incluir opciones propias de algunas bases de datos, herencia de tablas, etc.

Symfony también puede trabajar con esquemas escritos en formato XML. Así que no es necesario utilizar el formato YAML propio de Symfony si el esquema es demasiado complejo, si ya dispones de un esquema en formato XML o si estás acostumbrado a trabajar con la sintaxis XML de Propel. Solamente es necesario crear el archivo `schema.xml` en el directorio `config/` del proyecto y construir el modelo.

Todos los detalles incluidos en este capítulo no son específicos de Symfony sino de Propel. Propel es la capa de abstracción de objetos/relacional preferida por Symfony, pero se puede utilizar cualquier otra. No obstante, Symfony se integra mucho mejor con Propel por las siguientes razones:

Todas las clases del modelo de objetos de datos y las clases *Criteria* se cargan de forma automática. La primera vez que se utilizan, Symfony incluye los archivos adecuados y no es necesario preocuparse por añadir las instrucciones que incluyen esos archivos. En Symfony no es necesario arrancar o inicializar Propel. Cuando un objeto utiliza Propel, la librería se inicia automáticamente. Algunos de los *helpers* de Symfony utilizan objetos Propel como parámetros para realizar tareas complejas, como la paginación y el filtrado. Los objetos Propel permiten crear prototipos rápidamente y generar de forma automática la parte de gestión de la aplicación (el Capítulo 14 incluye más detalles). El esquema es mucho más fácil de escribir mediante el archivo `schema.yml`.

Y, como Propel es independiente de la base de datos utilizada, también lo es Symfony.

8.8. No crees el modelo dos veces

La desventaja de utilizar un ORM es que se debe definir la estructura de datos 2 veces: una para la base de datos y otra para el modelo de objetos. Por suerte, Symfony dispone de utilidades de línea de comandos para generar uno en función del otro, de modo que se evita duplicar el trabajo.

8.8.1. Construir la estructura SQL de la base de datos en función de un esquema existente

Si se crea la aplicación escribiendo el archivo `schema.yml`, Symfony puede generar las instrucciones SQL que crean las tablas directamente a partir del modelo de datos en YAML. Para generarlas, se ejecuta el siguiente comando desde el directorio raíz del proyecto:

```
| > php symfony propel:build-sql
```

El anterior comando crea un archivo `lib.model.schema.sql` en el directorio `miproyecto/data/sql/`. El código SQL generado se optimiza para el sistema gestor de bases de datos definido en el parámetro `phptype` del archivo `propel.ini`.

Se puede utilizar directamente el archivo `schema.sql` para construir la base de datos. Por ejemplo, en MySQL se puede ejecutar lo siguiente:

```
| > mysqladmin -u root -p create blog  
| > mysql -u root -p blog < data/sql/lib.model.schema.sql
```

El código SQL generado también es útil para reconstruir la base de datos en otro entorno o para cambiar de sistema gestor de bases de datos. Si el archivo `propel.ini` define las opciones de conexión correctas con la base de datos, el comando `php symfony propel:insert-sql` se encarga de crear automáticamente las tablas.

Sugerencia La línea de comandos también incluye una tarea para volcar los contenidos de un archivo de texto a la base de datos. El Capítulo 16 incluye más información sobre la tarea `propel:data-load` y sobre los archivos en formato YAML llamados *"fixtures"*.

8.8.2. Construir un modelo de datos en formato YAML a partir de una base de datos existente

Symfony puede utilizar la capa de acceso a bases de datos proporcionada por Creole para generar un archivo `schema.yml` a partir de una base de datos existente, gracias a la introspección (que es la capacidad de las bases de datos para determinar la estructura de las tablas que la forman). Se trata de una opción muy útil cuando se hace ingeniería inversa o si se prefiere trabajar primero en la base de datos antes de trabajar con el modelo de objetos.

Para construir el modelo, el archivo `propel.ini` del proyecto debe apuntar a la base de datos correcta y debe tener todas las opciones de conexión. Después, se ejecuta el comando `propel:build-schema`:

```
| > php symfony propel:build-schema
```

Se genera un nuevo archivo `schema.yml` a partir de la estructura de la base de datos y se almacena en el directorio `config/`. Ahora se puede construir el modelo a partir de este esquema.

El comando para generar el esquema es bastante potente y es capaz de añadir diversa información relativa a la base de datos en el esquema. Como el formato YAML no soporta este tipo de información sobre la base de datos, se debe generar un esquema en formato XML para poder incluirla. Para ello, solo es necesario añadir el argumento `xml` a la tarea `build-schema`:

```
| > php symfony propel:build-schema --xml
```

En vez de generar un archivo `schema.yml`, se crea un archivo `schema.xml` que es totalmente compatible con Propel y que contiene toda la información adicional. No obstante, los esquemas XML generados suelen ser bastante profusos y difíciles de leer.

Las tareas `propel:build-sql` y `propel:build-schema` no emplean las opciones de conexión definidas en el archivo `databases.yml`. En su lugar, estos comandos utilizan las opciones de conexión de otro archivo llamado `propel.ini` que se encuentra en el directorio `config/` del proyecto:

```
| propel.database.createUrl = mysql://login:passwd@localhost
| propel.database.url       = mysql://login:passwd@localhost/blog
```

Este archivo también contiene otras opciones que se utilizan para configurar el generador de Propel de forma que las clases del modelo generadas sean compatibles con Symfony. La mayoría de opciones son de uso interno y por tanto no interesan al usuario, salvo algunas de ellas:

```
| // Base classes are autoloading in symfony
| // Set this to true to use include_once statements instead
| // (Small negative impact on performance)
| propel.builder.addIncludes = false
|
| // Generated classes are not commented by default
| // Set this to true to add comments to Base classes
| // (Small negative impact on performance)
| propel.builder.addComentarios = false
|
| // Behaviors are not handled by default
| // Set this to true to be able to handle them
| propel.builder.AddBehaviors = false
```

Después de modificar las opciones del archivo `propel.ini`, se debe reconstruir el modelo para que los cambios surjan efecto.

8.9. Resumen

Symfony utiliza Propel como ORM y Creole como la capa de abstracción de bases de datos. De esta forma, en primer lugar se debe describir el esquema relacional de la base de datos en formato YAML antes de generar las clases del modelo de objetos. Después, durante la ejecución de la aplicación, se utilizan los métodos de las clases objeto y clases *peer* para acceder a la información de un registro o conjunto de registros. Se puede redefinir y ampliar el modelo fácilmente añadiendo métodos a las clases personalizadas. Las opciones de conexión se definen en el archivo `databases.yml`, que puede definir más de una conexión. La línea de comandos contiene tareas especiales que evitan tener que definir la estructura de la base de datos más de una vez.

La capa del modelo es la más compleja del framework Symfony. Una de las razones de esta complejidad es que la manipulación de datos es una tarea bastante intrincada. Las consideraciones de seguridad relacionadas con el modelo son cruciales para un sitio web y no deberían ignorarse. Otra de las razones es que Symfony se ajusta mejor a las aplicaciones medianas y grandes en un entorno empresarial. En ese tipo de aplicaciones, las tareas

automáticas proporcionadas por el modelo de Symfony suponen un gran ahorro de tiempo, por lo que merece la pena el tiempo dedicado a aprender su funcionamiento interno.

Así que no dudes en dedicar algo de tiempo a probar los objetos y métodos del modelo para entenderlos completamente. La recompensa será la gran solidez y escalabilidad de las aplicaciones desarrolladas.

Capítulo 9. Enlaces y sistema de enrutamiento

Los enlaces y las URL requieren de un tratamiento especial en cualquier framework para aplicaciones web. El motivo es que la definición de un único punto de entrada a la aplicación (mediante el controlador frontal) y el uso de *helpers* en las plantillas, permiten separar completamente el funcionamiento y el aspecto de las URL. Este mecanismo se conoce como "enrutamiento" (del inglés "*routing*"). El enrutamiento no es solo una utilidad curiosa, sino que es una herramienta muy útil para hacer las aplicaciones web más fáciles de usar y más seguras. En este capítulo se detalla la forma de manejar las URL en las aplicaciones de Symfony:

- Qué es y como funciona el sistema de enrutamiento
- Cómo utilizar *helpers* de enlaces en las plantillas para enlazar URL salientes
- Cómo configurar las reglas de enrutamiento para modificar el aspecto de las URL

Además, se incluyen una serie de trucos para mejorar el rendimiento del sistema de enrutamiento y para añadirle algunos toques finales.

9.1. ¿Qué es el enrutamiento?

El enrutamiento es un mecanismo que reescribe las URL para simplificar su aspecto. Antes de poder comprender su importancia, es necesario dedicar unos minutos al estudio de las URL de las aplicaciones

9.1.1. URL como instrucciones de servidor

Cuando el usuario realiza una acción, las URL se encargan de enviar la información desde el navegador hasta el servidor. Las URL tradicionales incluyen la ruta hasta el script del servidor y algunos parámetros necesarios para completar la petición, como se muestra en el siguiente ejemplo:

```
| http://www.ejemplo.com/web/controlador/articulo.php?id=123456&codigo_formato=6532
```

La URL anterior incluye información sobre la arquitectura de la aplicación y sobre su base de datos. Normalmente, los programadores evitan mostrar la estructura interna de la aplicación en la interfaz (las páginas por ejemplo se titulan "Perfil personal" y no "QZ7.65"). Desvelar detalles internos de la aplicación en la URL no solo contradice esta norma, sino que tiene otras desventajas:

- Los datos técnicos que se muestran en las URL son una fuente potencial de agujeros de seguridad. En el ejemplo anterior, ¿qué sucede si un usuario malicioso modifica el valor del parámetro *id*? ¿Supone este caso que la aplicación ofrece una interfaz directa a la base de datos? ¿Qué sucedería si otro usuario probara otros nombres de script, como por

ejemplo `admin.php`? En resumen, las URL directas permiten *jugar* de forma directa y sencilla con una aplicación y es casi imposible manejar su seguridad.

- Las URL complejas son muy difíciles de leer y hoy en día las URL no solo aparecen en la barra de direcciones. También suelen aparecer cuando un usuario pasa el ratón por encima de un enlace y también en los resultados de búsqueda. Cuando los usuarios buscan información, es más útil proporcionarles URL sencillas y fáciles de entender y no URL complejas como las que se muestran en la figura 9.1

Microsoft Office Clip Art and Media Home Page

Microsoft Office Clip Art and Media - Over 140000 clip art graphics, animations, photos, and sounds for use in **Microsoft** Office products.

office.**microsoft**.com/clipart/default.aspx?lc=en-us - 56k - 30 Aug 2006 -

Cached - [Similar pages](#)

Figura 9.1. Las URL aparecen en muchos lugares, como por ejemplo los resultados de búsqueda

- Si se modifica una URL (porque cambia el nombre del script o el de alguno de sus parámetros), se deben modificar todos los enlaces a esa URL. De esta forma, las modificaciones en la estructura del controlador son muy pesadas y costosas, lo que contradice la filosofía del desarrollo ágil de aplicaciones.

La situación podría ser incluso mucho peor si Symfony no utilizara un controlador frontal; es decir, si la aplicación contiene varios scripts accesibles desde el exterior, como por ejemplo:

```
http://www.ejemplo.com/web/galeria/album.php?nombre=mis%20vacaciones
http://www.ejemplo.com/web/weblog/publico/post/listado.php
http://www.ejemplo.com/web/general/contenido/pagina.php?nombre=sobre%20nosotros
```

En este caso, los programadores deben hacer coincidir la estructura de las URL y la estructura del sistema de archivos, por lo que su mantenimiento se convierte en una pesadilla cuando cualquiera de las dos estructuras se modifica.

9.1.2. URL como parte de la interfaz

Una de las ideas del sistema de enrutamiento es utilizar las URL como parte de la interfaz. Las aplicaciones trasladan información al usuario mediante el formateo de las URL y el usuario puede utilizar las URL para acceder a los recursos de la aplicación.

Lo anterior es posible en las aplicaciones Symfony porque la URL que se muestra al usuario no tiene que guardar obligatoriamente relación con la instrucción del servidor necesaria para completar la petición. En su lugar, la URL está relacionada con el recurso solicitado, y su aspecto puede configurarse libremente. En Symfony es posible por ejemplo utilizar la siguiente URL y obtener los mismos resultados que la primera URL mostrada en este capítulo:

```
| http://www.ejemplo.com/articulos/economia/2006/sectores-actividad.html
```

Este tipo de URL tiene muchas ventajas:

- Las URL tienen significado y ayudan a los usuarios a decidir si la página que se cargará al pulsar sobre un enlace contiene lo que esperan. Un enlace puede contener detalles adicionales sobre el recurso que enlaza. Esto último es especialmente útil para los resultados de los buscadores. Además, muchas veces las URL aparecen sin que se

mencione el título de su página (por ejemplo cuando se copian las URL en un mensaje de email) por lo que en ese caso deberían contener su propio significado. La figura 9-2 muestra una URL sencilla y fácil de entender.

[symfony PHP5 framework » AJAX pagination made simple](#)
 The **AJAX pagination** demo uses two very simple actions, both passing a pager to their template: `class pagerActions extends sfActions { public function ...`
www.symfony-project.com/weblog/2006/07/17/ajax-pagination-made-simple.html - 12k -
 Cached - [Similar pages](#)

Figura 9.2. Las URL pueden incluir información adicional sobre una página, como por ejemplo su fecha de publicación

- Las URL que aparecen en los documentos impresos son más fáciles de escribir y de recordar. Si la dirección del sitio web de una empresa se muestra en una tarjeta de visita con un aspecto similar a `http://www.ejemplo.com/controlador/web/index.jsp?id=ERD4`, probablemente no reciba muchas visitas.
- La URL se puede convertir en una especie de línea de comandos, que permita realizar acciones u obtener información de forma intuitiva. Este tipo de aplicaciones son las que más rápidamente utilizan los usuarios más avanzados.

```
// Listado de resultados: se puede añadir una nueva etiqueta para restringir los resultados
http://del.icio.us/tag/symfony+ajax
// Página de perfil de usuario: se puede modificar el nombre para obtener otro perfil
http://www.askeet.com/user/francois
```

- Se puede modificar el aspecto de la URL y el del nombre de la acción o de los parámetros de forma independiente y con una sola modificación. En otras palabras, es posible empezar a programar la aplicación y después modificar el aspecto de las URL sin estropear completamente la aplicación.
- Aunque se modifique la estructura interna de la aplicación, las URL pueden mantener su mismo aspecto hacia el exterior. De esta forma, las URL se convierten en persistentes y pueden ser añadidas a los marcadores o favoritos.
- Cuando los motores de búsqueda indexan un sitio web, suelen tratar de forma diferente (incluso saltándoselas) a las páginas dinámicas (las que acaban en `.php`, `.asp`, etc.) Así que si se formatean las URL de esta forma, los buscadores creen que están indexando contenidos estáticos, por lo que generalmente se obtiene una mejor indexación de las páginas de la aplicación.
- Son más seguras. Cualquier URL no reconocida se redirige a una página especificada por el programador y los usuarios no pueden navegar por el directorio raíz del servidor mediante la prueba de diferentes URL. La razón es que no se visualiza el nombre del script utilizado o el de sus parámetros.

La relación entre las URL mostradas al usuario y el nombre del script que se ejecuta y de sus parámetros está gestionada por el sistema de enrutamiento, que utiliza patrones que se pueden modificar mediante la configuración de la aplicación.

Nota ¿Qué sucede con los contenidos estáticos? Afortunadamente, las URL de los contenidos estáticos (imágenes, hojas de estilos y archivos de JavaScript) no suelen mostrarse durante la navegación, por lo que no es necesario utilizar el sistema de enrutamiento para este tipo de contenidos. Symfony almacena todos los contenidos estáticos en el directorio `web/` y sus URL se corresponden con su localización en el sistema de archivos. No obstante, es posible gestionar dinámicamente los contenidos estáticos mediante URL generadas con un *helper* para contenidos estáticos. Por ejemplo, para mostrar una imagen generada dinámicamente, se puede utilizar el *helper* `image_tag(url_for('captcha/image?key='.$key))`.

9.1.3. Cómo funciona

Symfony desasocia las URL externas y las URI utilizadas internamente. La correspondencia entre las dos es responsabilidad del sistema de enrutamiento. Symfony simplifica este mecanismo utilizando una sintaxis para las URI internas muy similar a la de las URL habituales. El listado 9-1 muestra un ejemplo.

Listado 9-1 - URL externas y URI internas

```
// Sintaxis de las URI internas
<modulo>/<accion>[?parametro1=valor1][&parametro2=valor2][&parametro3=valor3]...

// Ejemplo de URI interna que nunca se muestra al usuario
articulo/permalink?ano=2006&tema=economia&titulo=sectores-actividad

// Ejemplo de URL externa que se muestra al usuario
http://www.ejemplo.com/articulos/economia/2006/sectores-actividad.html
```

El sistema de enrutamiento utiliza un archivo de configuración especial, llamado `routing.yml`, en el que se pueden definir las reglas de enrutamiento. Si se considera la regla mostrada en el listado 9-2, se define un patrón cuyo aspecto es `articulos/*//*` y que también define el nombre de cada pieza que forma parte de la URL.

Listado 9-2 - Ejemplo de regla de enrutamiento

```
articulo_segun_titulo:
  url:    articulos/:tema/:ano/:titulo.html
  param: { module: articulo, action: permalink }
```

Todas las peticiones realizadas a una aplicación Symfony son analizadas en primer lugar por el sistema de enrutamiento (que es muy sencillo porque todas las peticiones se gestionan mediante un único controlador frontal). El sistema de enrutamiento busca coincidencias entre la URL de la petición y los patrones definidos en las reglas de enrutamiento. Si se produce una coincidencia, las partes del patrón que tienen nombre se transforman en parámetros de la petición y se juntan a los parámetros definidos en la clave `param`. El listado 9-3 muestra su funcionamiento.

Listado 9-3 - El sistema de enrutamiento interpreta las URL de las peticiones entrantes

```
// El usuario escribe (o pulsa) sobre esta URL externa
http://www.ejemplo.com/articulos/economia/2006/sectores-actividad.html

// El controlador frontal comprueba que coincide con la regla articulo_segun_titulo
// El sistema de enrutamiento crea los siguientes parámetros de la petición
```

```
'module' => 'articulo'
'action' => 'permalink'
'tema' => 'economia'
'ano' => '2006'
'titulo' => 'sectores-actividad'
```

Sugerencia La extensión `.html` de las URL externas es solo un adorno y por ese motivo el sistema de enrutamiento la ignora. Su única función es la de hacer que las páginas dinámicas parezcan páginas estáticas. La sección "Configuración del enrutamiento" al final de este capítulo explica cómo activar esta extensión.

Después, la petición se pasa a la acción `permalink` del módulo `articulo`, que dispone de toda la información necesaria en los parámetros de la petición para obtener el artículo solicitado.

El mecanismo de enrutamiento también funciona en la otra dirección. Para mostrar las URL en los enlaces de una aplicación, se debe proporcionar al sistema de enrutamiento la información necesaria para determinar la regla que se debe aplicar a cada enlace. Además, no se deben escribir los enlaces directamente con etiquetas `<a>` (ya que de esta forma no se estaría utilizando el sistema de enrutamiento) sino con un *helper* especial, tal y como se muestra en el listado 9-4.

Listado 9-4 - El sistema de enrutamiento formatea las URL externas mostradas en las plantillas

```
// El helper url_for() transforma una URI interna en una URL externa
<a href="<?php echo url_for('articulo/
permalink?tema=economia&ano=2006&titulo=sectores-actividad') ?>">pincha aquí</a>

// El helper reconoce que la URI cumple con la regla articulo_segun_titulo
// El sistema de enrutamiento crea una URL externa a partir de el
=> <a href="http://www.ejemplo.com/articulos/economia/2006/
sectores-actividad.html">pincha aquí</a>

// El helper link_to() muestra directamente un enlace
// y evita tener que mezclar PHP y HTML
<?php echo link_to(
    'pincha aqui',
    'articulo/permalink?tema=economia&ano=2006&titulo=sectores-actividad'
) ?>

// Internamente link_to() llama a url_for(), por lo que el resultado es el mismo
=> <a href="http://www.ejemplo.com/articulos/economia/2006/
sectores-actividad.html">pincha aquí</a>
```

De forma que el enrutamiento es un mecanismo bidireccional y solo funciona cuando se utiliza el *helper* `link_to()` para mostrar todos los enlaces.

9.2. Reescritura de URL

Antes de adentrarse en el funcionamiento interno del sistema de enrutamiento, se debe aclarar una cuestión importante. En los ejemplos mostrados en las secciones anteriores, las URI internas no incluyen el controlador frontal (`index.php` o `frontend_dev.php`). Como se sabe, es el controlador frontal y no otros elementos de la aplicación, el que decide el entorno de ejecución.

Por este motivo, todos los enlaces deben ser independientes del entorno de ejecución y el nombre del controlador frontal nunca aparece en las URI internas.

Además, tampoco se muestra el nombre del script PHP en las URL generadas en los ejemplos anteriores. La razón es que, por defecto, las URL no contienen el nombre de ningún script de PHP en el entorno de producción. El parámetro `no_script_name` del archivo `settings.yml` controla la aparición del nombre del controlador frontal en las URL generadas. Si se establece su valor a `off`, como se muestra en el listado 9-5, las URL generadas por los *helpers* incluirán el nombre del script del controlador frontal en cada enlace.

Listado 9-5 - Mostrando el nombre del controlador frontal en las URL, en `apps/frontend/settings.yml`

```
prod:
  .settings
    no_script_name: off
```

Ahora, las URL generadas tienen este aspecto:

```
| http://www.ejemplo.com/index.php/articulos/economia/2006/sectores-actividad.html
```

En todos los entornos salvo en el de producción, el parámetro `no_script_name` tiene un valor igual a `off` por defecto. Si se prueba la aplicación en el entorno de desarrollo, el nombre del controlador frontal siempre aparece en las URL.

```
| http://www.ejemplo.com/frontend_dev.php/articulos/economia/2006/sectores-actividad.html
```

En el entorno de producción, la opción `no_script_name` tiene el valor de `on`, por lo que las URL solo muestran la información necesaria para el enrutamiento y son más sencillas para los usuarios. No se muestra ningún tipo de información técnica.

```
| http://www.ejemplo.com/articulos/economia/2006/sectores-actividad.html
```

¿Cómo sabe la aplicación el nombre del script del controlador frontal que tiene que ejecutar? En este punto es donde comienza la reescritura de URL. El servidor web se puede configurar para que se llame siempre a un mismo script cuando la URL no indica el nombre de ningún script.

En el servidor web Apache se debe tener activado previamente el módulo `mod_rewrite`. Todos los proyectos de Symfony incluyen un archivo llamado `.htaccess` que añade las opciones necesarias para el `mod_rewrite` de Apache en el directorio `web/`. El contenido por defecto de este archivo se muestra en el listado 9-6.

Listado 9-6 - Reglas de reescritura de URL por defecto para Apache, en `mi proyecto/web/.htaccess`

```
<IfModule mod_rewrite.c>
  RewriteEngine On

  # we skip all files with .something
  RewriteCond %{REQUEST_URI} \.+$.
  RewriteCond %{REQUEST_URI} !\.html$
  RewriteRule .* - [L]
```

```
# we check if the .html version is here (caching)
RewriteRule ^$ index.html [QSA]
RewriteRule ^([^.]+)$ $1.html [QSA]
RewriteCond %{REQUEST_FILENAME} !-f

# no, so we redirect to our front web controller
RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

El servidor web analiza la estructura de las URL entrantes. Si la URL no contiene ningún sufijo y no existe ninguna versión cacheada de la página disponible (el Capítulo 12 detalla el sistema de cache), la petición se redirige al script `index.php`.

No obstante, el directorio `web/` de un proyecto Symfony lo comparten todas las aplicaciones y todos los entornos de ejecución del proyecto. Por este motivo, es habitual que exista más de un controlador frontal en el directorio `web`. Por ejemplo, si un proyecto tiene dos aplicaciones llamadas `frontend` y `backend` y dos entornos de ejecución llamados `dev` y `prod`, el directorio `web/` contiene 4 controladores frontales:

```
index.php          // frontend en prod
frontend_dev.php   // frontend en dev
backend.php        // backend en prod
backend_dev.php    // backend en dev
```

Las opciones de `mod_rewrite` solo permiten especificar un script por defecto. Si se establece el valor `on` a la opción `no_script_name` de todas las aplicaciones y todos los entornos, todas las URL se interpretan como si fueran peticiones al controlador frontal de la aplicación `frontend` en el entorno de producción (`prod`). Esta es la razón por la que en un mismo proyecto, solo se pueden aprovechar del sistema de enrutamiento una aplicación y un entorno de ejecución concretos.

Sugerencia Existe una forma de acceder a más de una aplicación sin indicar el nombre del script. Para ello, se crean subdirectorios en el directorio `web/` y se mueven los controladores frontales a cada subdirectorio. Después, se modifica la ruta a cada archivo de configuración `ProjectConfiguration` y se crea el archivo `.htaccess` de configuración para cada aplicación.

9.3. Helpers de enlaces

Debido al sistema de enrutamiento, es conveniente utilizar los *helpers* de enlaces en las plantillas en vez de etiquetas `<a>` normales y corrientes. Más que una molestia, el uso de estos *helpers* debe verse como un método sencillo de mantener la aplicación *limpia* y muy fácil de mantener. Además, los *helpers* de enlaces incluyen una serie de utilidades y atajos que no es recomendable desaprovechar.

9.3.1. Hiperenlaces, botones y formularios

En secciones anteriores ya se ha mostrado el *helper* `link_to()`. Se utiliza para mostrar enlaces válidos según XHTML y requiere de 2 parámetros: el elemento que va a mostrar el enlace y la URI interna del recurso al que apunta el enlace. Si en vez de un enlace se necesita un botón, simplemente se utiliza el *helper* `button_to()`. Los formularios también disponen de un *helper*

para controlar el valor del atributo `action`. El siguiente capítulo explica los formularios en detalle. El listado 9-7 muestra algunos ejemplos de *helpers* de enlaces.

Listado 9-7 - Helpers de enlaces para las etiquetas `<a>`, `<input>` y `<form>`

```
// Enlace simple de texto
<?php echo link_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <a href="/url/con/enrutamiento/a/Economia_en_Francia">Mi artículo</a>

// Enlace en una imagen
<?php echo link_to(image_tag('ver.gif'), 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <a href="/url/con/enrutamiento/a/Economia_en_Francia"></a>

// Boton
<?php echo button_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <input value="Mi artículo" type="button" onclick="document.location.href='/url/con/
enrutamiento/a/Economia_en_Francia';" />

// Formulario
<?php echo form_tag('articulo/ver?titulo=Economia_en_Francia') ?>
=> <form method="post" action="/url/con/enrutamiento/a/Economia_en_Francia" />
```

Los *helpers* de enlaces aceptan URI internas y también URL absolutas (las que empiezan por `http://` y para las que no se aplica el sistema de enrutamiento) y URL internas a una página (también llamadas *anclas*). Las aplicaciones reales suelen construir sus URI internas en base a una serie de parámetros dinámicos. El listado 9-8 muestra ejemplos de todos estos casos.

Listado 9-8 - URL que admiten los *helpers* de enlaces

```
// URI interna
<?php echo link_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <a href="/url/con/enrutamiento/a/Economia_en_Francia">Mi artículo</a>

// URI interna con parámetros dinámicos
<?php echo link_to('Mi artículo', 'articulo/ver?titulo='.$articulo->getTitulo()) ?>

// URI interna con anclas (enlaces a secciones internas de la página)
<?php echo link_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia#seccion1') ?>
=> <a href="/url/con/enrutamiento/a/Economia_en_Francia#seccion1">Mi artículo</a>

// URL absolutas
<?php echo link_to('Mi artículo', 'http://www.ejemplo.com/cualquierpagina.html') ?>
=> <a href="http://www.ejemplo.com/cualquierpagina.html">Mi artículo</a>
```

9.3.2. Opciones de los helpers de enlaces

Como se explicó en el Capítulo 7, los *helpers* aceptan como argumento opciones adicionales, que se pueden indicar en forma de array asociativo o en forma de cadena de texto. Los *helpers* de enlaces también aceptan este tipo de opciones, como muestra el listado 9-9.

Listado 9-9 - Los helpers de enlaces aceptan opciones adicionales

```
// Opciones adicionales como array asociativo
<?php echo link_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia', array(
```



```

    'class' => 'miclasecss',
    'target' => '_blank'
  )) ?>

  // Opciones adicionales como cadena de texto (producen el mismo resultado)
  <?php echo link_to('Mi artículo', 'articulo/
ver?titulo=Economia_en_Francia','class=miclasecss target=_blank') ?>
  => <a href="/url/con/enrutamiento/a/Economia_en_Francia" class="miclasecss"
target="_blank">Mi artículo</a>

```

También se pueden utilizar otras opciones específicas de Symfony llamadas `confirm` y `popup`. La primera muestra una ventana JavaScript de confirmación al pinchar en el enlace y la segunda opción abre el destino del enlace en una nueva ventana, como se muestra en el listado 9-10.

Listado 9-10 - Opciones `confirm` y `popup` en los *helpers* de enlaces

```

<?php echo link_to('Borrar elemento', 'item/borrar?id=123', 'confirm=¿Estás seguro?') ?>
=> <a onclick="return confirm('¿Estás seguro?');"
    href="/url/con/enrutamiento/a/borrar/123.html">Borrar elemento</a>

<?php echo link_to('Añadir al carrito', 'carritoCompra/anadir?id=100', 'popup=true') ?>
=> <a onclick="window.open(this.href);return false;"
    href="/url/con/enrutamiento/a/carritoCompra/anadir/id/100.html">Añadir al
carrito</a>

<?php echo link_to('Añadir al carrito', 'carritoCompra/anadir?id=100', array(
    'popup' => array('popupWindow', 'width=310,height=400,left=320,top=0')
  )) ?>
=> <a
onclick="window.open(this.href, 'popupWindow', 'width=310,height=400,left=320,top=0');return
false;"
    href="/url/con/enrutamiento/a/carritoCompra/anadir/id/100.html">Añadir al
carrito</a>

```

Estas opciones también se pueden combinar entre sí.

9.3.3. Opciones GET y POST falsas

En ocasiones, los programadores web utilizan peticiones GET para realizar acciones más propias de una petición POST. Si se considera por ejemplo la siguiente URL:

```
| http://www.ejemplo.com/index.php/carritoCompra/anadir/id/100
```

Este tipo de petición modifica los datos de la aplicación, ya que añade un elemento al objeto que representa el carrito de la compra y que se almacena en la sesión del servidor o en una base de datos. Si los usuarios añaden esta URL a los favoritos de sus navegadores o si la URL se cachea o es indexada por un buscador, se pueden producir problemas en la base de datos y en las métricas del sitio web. En realidad, esta petición debería tratarse como una petición de tipo POST, ya que los robots que utilizan los buscadores no hacen peticiones POST para indexar las páginas.

Symfony permite transformar una llamada a los *helpers* `link_to()` o `button_to()` en una petición POST. Solamente es necesario añadir la opción `post=true`, tal y como se muestra en el listado 9-11.

Listado 9-11 - Convirtiendo un enlace en una petición POST

```
<?php echo link_to('Ver carrito de la compra', 'carritoCompra/anadir?id=100',
'post=true') ?>
=> <a onclick="f = document.createElement('form'); document.body.appendChild(f);
      f.method = 'POST'; f.action = this.href; f.submit();return false;"
      href="/carritoCompra/anadir/id/100.html">Ver carrito de la compra</a>
```

La etiqueta `<a>` resultante conserva el atributo `href`, por lo que los navegadores sin soporte de JavaScript, como por ejemplo los robots que utilizan los buscadores, utilizan el enlace normal con la petición GET. Así que es posible que se deba restringir la acción para que solamente responda a las peticiones de tipo POST, que se puede realizar añadiendo por ejemplo la siguiente instrucción al principio de la acción:

```
| $this->forward404Unless($this->getRequest()->isMethod('post'));
```

Esta opción no se debe utilizar en los enlaces que se encuentran dentro de los formularios, ya que genera su propia etiqueta `<form>`.

Se trata de una buena práctica definir como peticiones POST los enlaces que realizan acciones que modifican los datos.

9.3.4. Forzando los parámetros de la petición como variables de tipo GET

Las variables que se pasan como parámetro a `link_to()` se transforman en patrones según las reglas del sistema de enrutamiento. Si no existe en el archivo `routing.yml` ninguna regla que coincida con la URI interna, se aplica la regla por defecto que transforma `modulo/accion?clave=valor` en `/modulo/accion/clave/valor`, como se muestra en el listado 9-12.

Listado 9-12 - Regla de enrutamiento por defecto

```
<?php echo link_to('Mi artículo', 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <a href="/articulo/ver/titulo/Economia_en_Francia">Mi artículo</a>
```

Si quieres utilizar la sintaxis de las peticiones GET (para pasar los parámetros de la petición en la forma `?clave=valor`) se deben indicar los parámetros en la opción `query_string`.

Si se utilizan enlaces que tienen una parte correspondiente a las anclas se pueden producir conflictos. Por ello, el nombre del ancla se debe indicar en la opción `anchor` en vez de añadirlo directamente a la URL. Todos los *helpers* de enlaces aceptan estas opciones, tal y como se muestra en el listado 9-13.

Listado 9-13 - Forzando el uso de variables tipo GET con la opción `query_string`

```
<?php echo link_to('Mi artículo', 'articulo/ver', array(
'query_string' => 'titulo=Economia_en_Francia',
'anchor' => 'seccion_dentro_de_la_pagina'
)) ?>
```

```
=> <a href="/articulo/ver?titulo=Economia_en_Francia#seccion_dentro_de_la_pagina">Mi
artículo</a>
```

Las URL con los parámetros en forma de variables GET se pueden interpretar por los scripts en el lado del cliente y por las variables `$_GET` y `$_REQUEST` en el lado del servidor.

El Capítulo 7 introdujo los helpers para contenidos estáticos `image_tag()`, `stylesheet_tag()` y `javascript_include_tag()`, que permiten incluir imágenes, hojas de estilos y archivos JavaScript en la respuesta del servidor. Las rutas a los contenidos estáticos no se procesan en el sistema de enrutamiento, ya que se trata de enlaces a recursos que se guardan en el directorio web público.

Además, no es necesario indicar la extensión para los contenidos estáticos. Symfony añade de forma automática las extensiones `.png`, `.js` o `.css` cuando se llama al *helper* de una imagen, un archivo JavaScript o una hoja de estilos. Symfony también busca de forma automática estos contenidos estáticos en los directorios `web/images/`, `web/js/` y `web/css/`. Evidentemente, es posible incluir otros tipos de archivos y archivos que se encuentren en otros directorios. Para ello, solo es necesario indicar como argumento el nombre completo del archivo o la ruta completa al archivo. Tampoco es necesario definir un valor para el atributo `alt` si el nombre del archivo enlazado es suficientemente significativo, ya que Symfony utiliza por defecto el nombre como atributo `alt`.

```
<?php echo image_tag('test') ?>
<?php echo image_tag('test.gif') ?>
<?php echo image_tag('/mis_imagenes/test.gif') ?>
=> <img href="/images/test.png" alt="Test" />
    <img href="/images/test.gif" alt="Test" />
    <img href="/mis_imagenes/test.gif" alt="Test" />
```

Para indicar un tamaño personalizado a una imagen, se utiliza la opción `size`. Esta opción requiere una anchura y una altura en píxel separadas por un `x`.

```
<?php echo image_tag('test', 'size=100x20') ?>
=> <img href="/images/test.png" alt="Test" width="100" height="20"/>
```

Si los contenidos estáticos se tienen que añadir en la sección `<head>` de la página (por ejemplo para los archivos JavaScript y las hojas de estilos), se deben utilizar los helpers `use_stylesheet()` y `use_javascript()` en las plantillas, en vez de las funciones acabadas en `_tag()` utilizadas en el layout. Estos helpers añaden los contenidos estáticos a la respuesta y los añaden antes de que se envíe la etiqueta `</head>` al navegador.

9.3.5. Utilizando rutas absolutas

Los *helpers* de enlaces y de contenidos estáticos generan rutas relativas por defecto. Para forzar el uso de rutas absolutas, se debe asignar el valor `true` a la opción `absolute`, como muestra el listado 9-14. Esta técnica es muy útil cuando se deben incluir enlaces en mensajes de email, canales RSS o respuestas de una API.

Listado 9-14 - Utilizando URL absolutas en vez de relativas

```

<?php echo url_for('articulo/ver?titulo=Economia_en_Francia') ?>
=> '/url/con/enrutamiento/a/Economia_en_Francia'
<?php echo url_for('articulo/ver?titulo=Economia_en_Francia', true) ?>
=> 'http://www.ejemplo.com/url/con/enrutamiento/a/Economia_en_Francia'

<?php echo link_to('economía', 'articulo/ver?titulo=Economia_en_Francia') ?>
=> <a href="/url/con/enrutamiento/a/Economia_en_Francia">economía</a>
<?php echo link_to('economía', 'articulo/
ver?titulo=Economia_en_Francia','absolute=true') ?>
=> <a href=" http://www.ejemplo.com/url/con/enrutamiento/a/
Economia_en_Francia">economía</a>

// Lo mismo sucede con los helpers de contenidos estáticos
<?php echo image_tag('prueba', 'absolute=true') ?>
<?php echo javascript_include_tag('miscript', 'absolute=true') ?>

```

Hoy en día, existen robots que rastrean todas las páginas web en busca de direcciones de correo electrónico que puedan ser utilizadas en los envíos masivos de spam. Por este motivo, no se pueden incluir directamente las direcciones de correo electrónico en las páginas web sin acabar siendo una víctima del spam en poco tiempo. Afortunadamente, Symfony proporciona un *helper* llamado `mail_to()`.

El *helper* `mail_to()` requiere 2 parámetros: la dirección de correo electrónico real y la cadena de texto que se muestra al usuario. Como opción adicional se puede utilizar el parámetro `encode`, que produce un código HTML bastante difícil de leer, que los navegadores muestran correctamente, pero que los robots de spam no son capaces de entender.

```

<?php echo mail_to('midireccion@midominio.com', 'contacto') ?>
=> <a href="mailto:midireccion@midominio.com">contacto</a>
<?php echo mail_to('midireccion@midominio.com', 'contacto', 'encode=true') ?>
=> <a href="&#109;&#x61;... &#111;&#x64;">&#x63;&#x74;... e&#115;&#x73;</a>

```

Las direcciones de email resultantes están compuestas por caracteres transformados por un codificador aleatorio que los transforma en entidades decimales y hexadecimales aleatoriamente. Aunque este truco funciona para la mayoría de robots de spam, las técnicas que emplean este tipo de empresas evolucionan rápidamente y podrían dejar obsoleta esta técnica en poco tiempo.

9.4. Configuración del sistema de enrutamiento

El sistema de enrutamiento se encarga de 2 tareas:

- Interpreta las URL externas de las peticiones entrantes y las transforma en URI internas para determinar el módulo, la acción y los parámetros de la petición.
- Transforma las URI internas utilizadas en los enlaces en URL externas (siempre que se utilicen los *helpers* de enlaces).

La transformación se realiza en base a una serie de reglas de enrutamiento. Todas estas reglas se almacenan en un archivo de configuración llamado `routing.yml` y que se encuentra en el directorio `config/`. El listado 9-15 muestra las reglas que incluyen por defecto todos los proyectos de Symfony.

Listado 9-15 - Las reglas de enrutamiento por defecto, en frontend/config/routing.yml

```
# default rules
homepage:
  url:    /
  param: { module: default, action: index }

default_symfony:
  url:    /symfony/:action/*
  param: { module: default }

default_index:
  url:    /:module
  param: { action: index }

default:
  url:    /:module/:action/*
```

9.4.1. Reglas y patrones

Las reglas de enrutamiento son asociaciones biyectivas entre las URL externas y las URI internas. Una regla típica está formada por:

- Un identificador único en forma de texto, que se define por legibilidad y por rapidez, y que se puede utilizar en los *helpers* de enlaces
- El patrón que debe cumplirse (en la clave `url`)
- Un array de valores para los parámetros de la petición (en la clave `param`)

Los patrones pueden contener comodines (que se representan con un asterisco, `*`) y comodines con nombre (que empiezan por *2 puntos*, `:`). Si se produce una coincidencia con un comodín con nombre, ese valor que coincide se transforma en un parámetro de la petición. Por ejemplo, la regla anterior llamada `default` produce coincidencias con cualquier URL del tipo `/valor1/valor2`, en cuyo caso se ejecutará el módulo llamado `valor1` y la acción llamada `valor2`. Y en la regla llamada `default_symfony`, el valor `symfony` es una palabra clave y `action` es un comodín con nombre que se transforma en parámetro de la petición.

Nota A partir de la versión 1.1 de Symfony, los comodines con nombre pueden separarse con un guión medio o con un punto, por lo que es posible crear patrones avanzados como el siguiente:

```
mi_regla:
  url:    /ruta/:parametro1.:formato
  param: { module: mimodulo, action: miaccion }
```

Si se define la regla anterior, una URL como `/ruta/12.xml` produce una coincidencia con esa regla y provoca que se ejecute la acción `miaccion` dentro del módulo `mimodulo`. Además, a la acción se le pasa un parámetro llamado `parametro1` con valor igual a `12` y otro parámetro llamado `formato` con valor `xml`.

Si quieres utilizar otros separadores, puedes modificar la opción `segment_separators` en la configuración de la factoría `sfPatternRouting` (ver capítulo 19).

El sistema de enrutamiento procesa el archivo `routing.yml` desde la primera línea hasta la última y se detiene en la primera regla que produzca una coincidencia. Por este motivo se deben añadir las reglas personalizadas antes que las reglas por defecto. Si se consideran las reglas del listado 9-16, la URL `/valor/123` produce coincidencias con las dos reglas, pero como Symfony prueba primero la regla `mi_regla:`, y esa regla produce una coincidencia, ni siquiera se llega a probar la regla `default:`. De esta forma, la petición se procesa en la acción `mimodulo/miaccion` con el parámetro `id` inicializado con el valor 123 (no se procesa por tanto en la acción `valor/123`).

Listado 9-16 - Las reglas se procesan de principio a fin

```
mi_regla:
  url:    /valor/:id
  param: { module: mimodulo, action: miaccion }

# default rules
default:
  url:    /:module/:action/*
```

Nota No siempre que se crea una nueva acción es necesario añadir una nueva regla al sistema de enrutamiento. Si el patrón `modulo/accion` es útil para la nueva acción, no es necesario añadir más reglas al archivo `routing.yml`. Sin embargo, si se quieren personalizar las URL externas de la acción, es necesario añadir una nueva regla por encima de las reglas por defecto.

El listado 9-17 muestra el proceso de modificación del formato de la URL externa de la acción `articulo/ver`.

Listado 9-17 - Modificación del formato de las URL externas de la acción `articulo/ver`

```
<?php echo url_for('Mi artículo', 'articulo/ver?id=123') ?>
=> /articulo/ver/id/123           // Formato por defecto

// Para cambiarlo por /articulo/123, se añade una nueva regla al
// principio del archivo routing.yml
articulo_segun_id:
  url:    /articulo/:id
  param: { module: articulo, action: ver }
```

El problema es que la regla `articulo_segun_id` del listado 9-17 rompe con el enrutamiento normal de todas las otras acciones del módulo `articulo`. De hecho, ahora una URL como `articulo/borrar` produce una coincidencia en esta regla, por lo que no se ejecuta la regla `default`, sino que se ejecuta la regla `articulo_segun_id`. Por tanto, esta URL no llama a la acción `borrar`, sino que llama a la acción `ver` con el atributo `id` inicializado con el valor `borrar`. Para evitar estos problemas, se deben definir restricciones en el patrón, de forma que la regla `articulo_segun_id` solo produzca coincidencias con las URL cuyo comodín `id` sea un número entero.

9.4.2. Restricciones en los patrones

Cuando una URL puede producir coincidencias con varias reglas diferentes, se deben refinar las reglas añadiendo restricciones o requisitos a sus patrones. Un requisito es una serie de

expresiones regulares que deben cumplir los comodines para que la regla produzca una coincidencia.

Para modificar por ejemplo la regla `articulo_segun_id` anterior de forma que solo se aplique a las URL cuyo atributo `id` sea un número entero, se debe añadir una nueva línea a la regla, como muestra el listado 9-18.

Listado 9-18 - Añadiendo requisitos a las reglas de enrutamiento

```
articulo_segun_id:
  url: /articulo/:id
  param: { module: articulo, action: ver }
  requirements: { id: \d+ }
```

Ahora, una URL como `articulo/borrar` nunca producirá una coincidencia con la regla `articulo_segun_id`, porque la cadena de texto `borrar` no cumple con los requisitos de la regla. Por consiguiente, el sistema de enrutamiento continua buscando posibles coincidencias con otras reglas hasta que al final la encuentra en la regla llamada `default`.

Una buena recomendación sobre seguridad es la de no utilizar claves primarias en las URL y sustituirlas por cadenas de texto siempre que sea posible. ¿Cómo sería posible acceder a los artículos a través de su título en lugar de su ID? Las URL externas resultantes serían de esta forma:

```
| http://www.ejemplo.com/articulo/Economia_en_Francia
```

Para utilizar estas URL, se crea una nueva acción llamada `permalink` y que utiliza un parámetro llamado `slug` en vez del parámetro `id` habitual. (**Nota del traductor:** “*slug*” es un término adaptado del periodismo anglosajón y que hace referencia al título de una noticia o artículo en el que se han sustituido los espacios en blanco por guiones y se han eliminado todos los caracteres que no sean letras o números, lo que los hace ideales para utilizarse como parte de las URL) La nueva regla queda de la siguiente forma:

```
articulo_segun_id:
  url: /articulo/:id
  param: { module: articulo, action: ver }
  requirements: { id: \d+ }

articulo_segun_slug:
  url: /articulo/:slug
  param: { module: articulo, action: permalink }
```

La acción `permalink` debe buscar el artículo solicitado a partir de su título, por lo que el modelo de la aplicación debe proporcionar el método adecuado.

```
public function executePermalink($peticion)
{
    $articulo = ArticlePeer::obtieneSegunSlug($peticion->getParameter('slug'));
    $this->forward404Unless($articulo); // Muestra un error 404 si no se encuentra el
    artículo
    $this->articulo = $articulo;        // Pasar el objeto a la plantilla
}
```

También es necesario modificar los enlaces que apuntan a la acción `ver` en las plantillas por nuevos enlaces que apunten a la acción `permalink`, para que se aplique correctamente el nuevo formato de las URI internas.

```
// Se debe sustituir esta línea...
<?php echo link_to('Mi artículo', 'articulo/ver?id='.$articulo->getId()) ?>

// ...por esta otra
<?php echo link_to('Mi artículo', 'articulo/permalink?slug='.$articulo->getSlug()) ?>
```

Gracias a la definición de `requirements` en las reglas, las URL externas como `/articulo/Economia_en_Francia` se procesan en la regla `articulo_segun_slug` aunque la regla `articulo_segun_id` aparezca antes.

Por último, como ahora los artículos se buscan a partir del campo `slug`, se debería añadir un índice a esa columna del modelo para optimizar el rendimiento de la base de datos.

9.4.3. Asignando valores por defecto

Para completar las reglas, se pueden asignar valores por defecto a los comodines con nombre, incluso aunque el parámetro no esté definido. Los valores por defecto se establecen en el array `param::`.

Por ejemplo, la regla `articulo_segun_id` no se ejecuta si no se pasa el parámetro `id`. El listado 9-19 muestra como forzar la presencia de ese parámetro.

Listado 9-19 - Asignar un valor por defecto a un comodín

```
articulo_segun_id:
  url:           /articulo/:id
  param:         { module: articulo, action: ver, id: 1 }
```

Los parámetros por defecto no necesariamente tienen que ser comodines que se encuentran en el patrón de la regla de enrutamiento. En el listado 9-20, al parámetro `display` se le asigna el valor `true`, aunque ni siquiera forma parte de la URL.

Listado 9-20 - Asignar un valor por defecto a un parámetro de la petición

```
articulo_segun_id:
  url:           /articulo/:id
  param:         { module: articulo, action: ver, id: 1, display: true }
```

Si se mira con un poco de detenimiento, se puede observar que `articulo` y `ver` son también valores por defecto asignados a las variables `module` y `action` que no se encuentran en el patrón de la URL.

Sugerencia Para incluir un parámetro por defecto en todas las reglas de enrutamiento, se utiliza el método `sfRouting::addDefaultParameter()`. Si por ejemplo se necesita que todas las reglas tengan un parámetro llamado `tema` con un valor por defecto igual a `default`, se añade la instrucción `$this->context->getRouting()->setDefaultParameter('tema', 'default');` en al menos un filtro global de la aplicación.

9.4.4. Acelerando el sistema de enrutamiento mediante el uso de los nombres de las reglas

Los *helpers* de enlaces aceptan como argumento el nombre o etiqueta de la regla en vez del par modulo/acción, siempre que la etiqueta vaya precedida del signo @, como muestra el listado 9-21.

Listado 9-21 - Uso de la etiqueta de las reglas en vez de Modulo/Acción

```
<?php echo link_to('Mi artículo', 'articulo/ver?id='.$articulo->getId()) ?>

// también se puede escribir como...
<?php echo link_to('Mi artículo', '@articulo_segun_id?id='.$articulo->getId()) ?>
```

Esta técnica tiene sus ventajas e inconvenientes. En cuanto a las ventajas:

- El formateo de las URI internas es más rápido, ya que Symfony no debe recorrer todas las reglas hasta encontrar la que se corresponde con el enlace. Si la página contiene un gran número de enlaces, el ahorro de tiempo de las reglas con nombre será apreciable respecto a los pares módulo/acción.
- El uso de los nombres de las reglas permite abstraer aun más la lógica de la acción. Si se modifica el nombre de la acción pero se mantiene la URL, solo será necesario realizar un cambio en el archivo `routing.yml`. Todas las llamadas al *helper* `link_to()` funcionarán sin tener que realizar ningún cambio.
- La lógica que se ejecuta es más comprensible si se utiliza el nombre de la regla. Aunque los módulos y las acciones tengan nombres explícitos, normalmente es más comprensible llamar a la regla `@ver_articulo_segun_slug` que simplemente llamar a `articulo/ver`.

Por otra parte, la desventaja principal es que es más complicado añadir los enlaces, ya que siempre se debe consultar el archivo `routing.yml` para saber el nombre de la regla que se utiliza en la acción.

La mejor técnica de las 2 depende del proyecto en el que se trate, por lo que es el programador el que tendrá que tomar la decisión.

Sugerencia Mientras se prueba la aplicación (en el entorno dev), se puede comprobar la regla que se está aplicando para cada petición del navegador. Para ello, se debe desplegar la sección *"logs and msgs"* de la barra de depuración y se debe buscar la línea que dice *"matched route XXX"*. El Capítulo 16 contiene más información sobre el modo de depuración web.

Nota Desde la versión 1.1 de Symfony las operaciones del sistema de enrutamiento son mucho más rápidas en el entorno de producción, ya que las conversiones de URI internas a URL externas se guardan en la caché.

9.4.5. Añadiendo la extensión .html

Si se comparan estas dos URL:

```
http://frontend.ejemplo.com/articulo/Economia_en_Francia
http://frontend.ejemplo.com/articulo/Economia_en_Francia.html
```

Aunque se trata de la misma página, los usuarios (y los robots que utilizan los buscadores) las consideran como si fueran diferentes debido a sus URL. La segunda URL parece que pertenece a un directorio web de páginas estáticas correctamente organizadas, que es exactamente el tipo de sitio web que mejor saben indexar los buscadores.

Para añadir un sufijo a todas las URL externas generadas en el sistema de enrutamiento, se debe modificar el valor de la opción `suffix` en el archivo de configuración `settings.yml`, como se muestra en el listado 9-22.

Listado 9-22 - Establecer un sufijo a todas las URL, en `frontend/config/settings.yml`

```
prod:
  .settings
    suffix:          .html
```

El sufijo por defecto es un punto (`.`), lo que significa que el sistema de enrutamiento no añade ningún sufijo a menos que se especifique uno.

En ocasiones es necesario indicar un sufijo específico para una única regla de enrutamiento. En ese caso, se indica el sufijo directamente como parte del patrón definido mediante `url`: en la regla del archivo `routing.yml`, como se muestra en el listado 9-23. El sufijo global se ignora en este caso.

Listado 9-23 - Estableciendo un sufijo en una única URL, en `frontend/config/routing.yml`

```
articulo_listado:
  url:          /ultimos_articulos
  param:        { module: articulo, action: listado }

articulo_listado_rss:
  url:          /ultimos_articulos.rss
  param:        { module: articulo, action: listado, type: feed }
```

9.4.6. Creando reglas sin el archivo `routing.yml`

Como sucede con la mayoría de archivos de configuración, el archivo `routing.yml` es una buena solución para definir las reglas del sistema de enrutamiento, pero no es la única solución. Se pueden definir reglas en PHP, en el archivo `config.php` de la aplicación o en el script del controlador frontal, pero antes de llamar a la función `dispatch()`, ya que este método determina la acción que se ejecuta en función de las reglas de enrutamiento disponibles en ese momento. Definir reglas mediante PHP permite crear reglas dinámicas que dependan de la configuración o de otros parámetros.

El objeto que gestiona las reglas de enrutamiento es una factoría llamada `sfPatternRouting`. Se encuentra disponible en cualquier parte del código mediante la llamada `sfContext::getInstance()->getRouting()`. Su método `prependRoute()` añade una nueva regla por encima de las reglas definidas en el archivo `routing.yml`. El método espera 4 parámetros, que son los mismos que se utilizan para definir una regla: la etiqueta de la ruta, el patrón de la URL, el array asociativo con los valores por defecto y otro array asociativo con los requisitos. La

regla definida en el archivo `routing.yml` del listado 9-18 es equivalente por ejemplo al código PHP mostrado en el listado 9-24.

Nota A partir de la versión 1.1 de Symfony la clase que se encarga del enrutamiento se puede configurar en el archivo de configuración `factories.yml`. En este capítulo se explica el funcionamiento de la clase `sfPatternRouting`, que es la clase configurada por defecto para gestionar el sistema de enrutamiento, mientras que en el capítulo 17 se explica cómo cambiar esa clase por defecto.

Listado 9-24 - Definiendo una regla en PHP

```
sfContext::getInstance()->getRouting()->prependRoute(  
    'articulo_segun_id',                // Nombre ruta  
    '/articulo/:id',                  // Patrón de la ruta  
    array('module' => 'articulo', 'action' => 'ver'), // Valores por defecto  
    array('id' => '\d+'),              // Requisitos  
);
```

La clase `sfPatternRouting` define otros métodos muy útiles para la gestión manual de las rutas: `clearRoutes()`, `hasRoutes()`, etc. La API de Symfony (http://www.symfony-project.org/api/1_1/) dispone de mucha más información.

Sugerencia A medida que se profundiza en los conceptos presentados en este libro, se pueden ampliar los conocimientos visitando la documentación de la API disponible online o incluso, investigando el código fuente de Symfony. En este libro no se describen todas las opciones y parámetros de Symfony, pero la documentación online contiene todos los detalles posibles.

9.5. Trabajando con rutas en las acciones

En ocasiones es necesario obtener información sobre la ruta actual, por ejemplo para preparar un enlace típico de "Volver a la página XXX". En estos casos, se deben utilizar los métodos disponibles en el objeto `sfPatternRouting`. Las URI devueltas por el método `getCurrentInternalUri()` se pueden utilizar directamente en las llamadas al *helper* `link_to()`, como se muestra en el listado 9-25.

Listado 9-25 - Uso de `sfRouting` para obtener información sobre la ruta actual

```
// Si se necesita una URL como la siguiente  
http://frontend.ejemplo.com/articulo/21  
  
$enrutamiento = sfContext::getInstance()->getRouting();  
  
// Se utiliza lo siguiente en la acción articulo/ver  
$uri = $enrutamiento->getCurrentInternalUri();  
=> articulo/ver?id=21  
  
$uri = $enrutamiento->getCurrentInternalUri(true);  
=> @articulo_segun_id?id=21  
  
$regla = $enrutamiento->getCurrentRouteName();  
=> articulo_segun_id
```

```
// Si se necesitan los nombres del módulo y de la acción,  
// se pueden utilizar los parámetros de la petición  
$modulo = $peticion->getParameter('module');  
$accion = $peticion->getParameter('action');
```

Si se necesita transformar dentro de la acción una URI interna en una URL externa, como se hace en las plantillas con el *helper* `url_for()`, se utiliza el método `genUrl()` del objeto `sfController`, como se muestra en el listado 9-26.

Listado 9-26 - Uso de `sfController` para transformar una URI interna

```
$uri = 'articulo/ver?id=21';  
  
$url = $this->getController()->genUrl($uri);  
=> /articulo/21  
  
$url = $this->getController()->genUrl($uri, true);  
=> http://frontend.ejemplo.com/articulo/21
```

9.6. Resumen

El sistema de enrutamiento es un mecanismo bidireccional diseñado para formatear las URL externas de forma que sean más fáciles para los usuarios. La reescritura de URL es necesaria para omitir el nombre del controlador frontal de las aplicaciones de cada proyecto. Para que el sistema de enrutamiento funcione en ambas direcciones, es necesario utilizar los *helpers* de enlaces cada vez que se incluye un enlace en las plantillas. El archivo `routing.yml` configura las reglas del sistema de enrutamiento, su prioridad y sus requisitos. El archivo `settings.yml` controla otras opciones adicionales como la presencia del nombre del controlador frontal en las URL y el uso de sufijos en las URL generadas.

Capítulo 10. Formularios

Cuidado En este capítulo se describe el funcionamiento de los formularios de Symfony 1.0. No obstante, esta información sigue siendo válida en Symfony 1.1 por motivos de compatibilidad y porque el generador de la parte de administración de las aplicaciones todavía utiliza este tipo de formularios. No obstante, si estás desarrollando un proyecto nuevo con Symfony 1.1, deberías utilizar el nuevo mecanismo de formularios que se explica en un libro dedicado exclusivamente a los formularios y que publicaremos próximamente.

Cuando se crean las plantillas, la mayor parte del tiempo se dedica a los formularios. No obstante, los formularios normalmente se diseñan bastante mal. Como se debe prestar atención a los valores por defecto, al formato de los datos, a la validación, a la recarga de los datos introducidos y al manejo en general de los formularios, algunos programadores tienden a olvidar otros aspectos importantes. Por este motivo, Symfony presta especial atención a este tema. En este capítulo se describen las herramientas que automatizan partes de este proceso y que aceleran el desarrollo de los formularios:

- Los *helpers* de formulario proporcionan una manera más rápida de crear controles de formulario en las plantillas, sobre todo para los elementos más complejos como fechas, listas desplegables y áreas de texto con formato.
- Si un formulario se encarga de modificar las propiedades de un objeto, el uso de los *helpers* de objetos aceleran el desarrollo de las plantillas.
- Los archivos YAML de validación facilitan la validación de los formularios y la recarga de los datos introducidos.
- Los validadores encapsulan todo el código necesario para validar los datos introducidos por el usuario. Symfony incluye validadores para la mayoría de casos habituales y permite añadir validadores propios de forma sencilla.

10.1. Helpers de formularios

En las plantillas, es común mezclar las etiquetas HTML con código PHP. Los *helpers* de formularios que incluye Symfony intentan simplificar esta tarea para evitar tener que incluir continuamente etiquetas `<?php echo` en medio de las etiquetas `<input>`.

10.1.1. Etiqueta principal de los formularios

Como se explicó en el capítulo anterior, para crear un formulario se emplea el *helper* `form_tag()`, ya que se encarga de transformar la acción que se pasa como parámetro a una URL válida para el sistema de enrutamiento. El segundo argumento se emplea para indicar opciones adicionales, como por ejemplo, cambiar el valor del `method` por defecto, establecer el valor de `enctype` o especificar otros atributos. El listado 10-1 muestra algunos ejemplos.

Listado 10-1 - El *helper* `form_tag()`

```

<?php echo form_tag('prueba/guardar') ?>
=> <form method="post" action="/ruta/a/guardar">

<?php echo form_tag('prueba/guardar', 'method=get multipart=true
class=formularioSimple') ?>
=> <form method="get" enctype="multipart/form-data" class="formularioSimple"
action="/ruta/a/guardar">

```

Como no se utiliza un *helper* para cerrar el formulario, siempre debe incluirse la etiqueta HTML `</form>`, aunque no quede bien en el código fuente de la plantilla.

10.1.2. Elementos comunes de formulario

Los *helpers* de formulario asignan por defecto a cada elemento un atributo `id` cuyo valor coincide con su atributo `name`, aunque esta no es la única convención útil. El listado 10-2 muestra una lista completa de los *helpers* disponibles para los elementos comunes de formularios y sus opciones.

Listado 10-2 - Sintaxis de los *helpers* para los elementos comunes de formulario

```

// Cuadro de texto (input)
<?php echo input_tag('nombre', 'valor inicial') ?>
=> <input type="text" name="nombre" id="nombre" value="valor inicial" />

// Todos los helpers de formularios aceptan un parámetro con opciones adicionales
// De esta forma es posible añadir atributos propios a la etiqueta que se genera
<?php echo input_tag('nombre', 'valor inicial', 'maxlength=20') ?>
=> <input type="text" name="nombre" id="nombre" value="valor inicial" maxlength="20" />

// Cuadro de texto grande (área de texto)
<?php echo textarea_tag('nombre', 'valor inicial', 'size=10x20') ?>
=> <textarea name="nombre" id="nombre" cols="10" rows="20">
    valor inicial
</textarea>

// Checkbox
<?php echo checkbox_tag('soltero', 1, true) ?>
<?php echo checkbox_tag('carnet_conducir', 'B', false) ?>
=> <input type="checkbox" name="soltero" id="soltero" value="1" checked="checked" />
    <input type="checkbox" name="carnet_conducir" id="carnet_conducir" value="B" />

// Radio button
<?php echo radiobutton_tag('estado[]', 'valor1', true) ?>
<?php echo radiobutton_tag('estado[]', 'valor2', false) ?>
=> <input type="radio" name="estado[]" id="estado_valor1" value="valor1"
checked="checked" />
    <input type="radio" name="estado[]" id="estado_valor2" value="valor2" />

// Lista desplegable (select)
<?php echo select_tag('pago',
    '<option selected="selected">Visa</option>
    <option>Eurocard</option>
    <option>Mastercard</option>')
?>

```

```

=> <select name="pago" id="pago">
    <option selected="selected">Visa</option>
    <option>Eurocard</option>
    <option>Mastercard</option>
</select>

// Lista de opciones para una etiqueta select
<?php echo options_for_select(array('Visa', 'Eurocard', 'Mastercard'), 0) ?>
=> <option value="0" selected="selected">Visa</option>
    <option value="1">Eurocard</option>
    <option value="2">Mastercard</option>

// Helper de lista desplegable con una lista de opciones
<?php echo select_tag('pago', options_for_select(array(
    'Visa',
    'Eurocard',
    'Mastercard'
), 0)) ?>
=> <select name="pago" id="pago">
    <option value="0" selected="selected">Visa</option>
    <option value="1">Eurocard</option>
    <option value="2">Mastercard</option>
</select>

// Para indicar el nombre de las opciones, se utiliza un array asociativo
<?php echo select_tag('nombre', options_for_select(array(
    'Steve' => 'Steve',
    'Bob'   => 'Bob',
    'Albert' => 'Albert',
    'Ian'   => 'Ian',
    'Buck'  => 'Buck'
), 'Ian')) ?>
=> <select name="nombre" id="nombre">
    <option value="Steve">Steve</option>
    <option value="Bob">Bob</option>
    <option value="Albert">Albert</option>
    <option value="Ian" selected="selected">Ian</option>
    <option value="Buck">Buck</option>
</select>

// Lista desplegable que permite una selección múltiple
// (los valores seleccionados se pueden indicar en forma de array)
<?php echo select_tag('pago', options_for_select(
    array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' => 'Mastercard'),
    array('Visa', 'Mastercard'),
), array('multiple' => true)) ?>
=> <select name="pago[]" id="pago" multiple="multiple">
    <option value="Visa" selected="selected">Visa</option>
    <option value="Eurocard">Eurocard</option>
    <option value="Mastercard">Mastercard</option>
</select>

// Lista desplegable que permite una selección múltiple
// (los valores seleccionados se pueden indicar en forma de array)
<?php echo select_tag('pago', options_for_select(

```

```

    array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' => 'Mastercard'),
    array('Visa', 'Mastercard')
), 'multiple=multiple') ?>
=> <select name="pago" id="pago" multiple="multiple">
    <option value="Visa" selected="selected">
    <option value="Eurocard">Eurocard</option>
    <option value="Mastercard" selected="selected">Mastercard</option>
</select>

// Campo para adjuntar archivos
<?php echo input_file_tag('nombre') ?>
=> <input type="file" name="nombre" id="nombre" value="" />

// Cuadro de texto de contraseña
<?php echo input_password_tag('nombre', 'valor') ?>
=> <input type="password" name="nombre" id="nombre" value="valor" />

// Campo oculto
<?php echo input_hidden_tag('nombre', 'valor') ?>
=> <input type="hidden" name="nombre" id="nombre" value="valor" />

// Botón de envío de formulario (botón normal de texto)
<?php echo submit_tag('Guardar') ?>
=> <input type="submit" name="submit" value="Guardar" />

// Botón de envío de formulario (botón creado con la imagen indicada)
<?php echo submit_image_tag('imagen_envio') ?>
=> <input type="image" name="submit" src="/images/imagen_envio.png" />

```

El *helper* `submit_image_tag()` utiliza la misma sintaxis y tiene las mismas características que `image_tag()`.

Nota En los radio button, el valor del atributo `id` no se copia directamente del atributo de `name`, sino que se construye mediante una combinación del nombre y de cada valor. El motivo es que el atributo `name` debe tener el mismo valor para todos los radio button que se quieren definir como mutuamente excluyentes, al mismo tiempo que en una página HTML dos o más elementos no pueden disponer del mismo valor para su atributo `id`.

¿Cómo se obtienen los datos enviados por los usuarios a través de los formularios? Los datos se encuentran disponibles en los parámetros de la petición, por lo que en una acción se debe llamar a `$peticion->getParameter($nombreElemento)` para obtener el valor.

Una buena práctica consiste en utilizar la misma acción para mostrar y para procesar el formulario. En función del método de la solicitud (GET o POST) se muestra la plantilla del formulario o se procesan los datos enviados para redirigir a otra acción.

```

// En mimodulo/actions/actions.class.php
public function executeModificarAutor($peticion)
{
    if (!$this->getRequest()->isMethod('post'))
    {
        // Mostrar el formulario
        return sfView::SUCCESS;
    }
}

```



```

    else
    {
        // Procesar Los datos del formulario
        $nombre = $peticion->getParameter('nombre');
        ...
        $this->redirect('mimodulo/otraaccion');
    }
}

```

Para que esta técnica funcione, el destino del formulario tiene que ser la misma acción que la acción que muestra el formulario.

```

// En mimodulo/templates/modificarAutorSuccess.php
<?php echo form_tag('mimodulo/modificarAutor') ?>

...

```

Symfony también incluye *helpers* de formularios para realizar peticiones asíncronas en segundo plano. El siguiente capítulo se centra en Ajax y proporciona todos los detalles.

10.1.3. Campos para introducir fechas

Muchos formularios permiten al usuario introducir fechas. Uno de los principales fallos en los datos de los formularios suele ser el formato incorrecto de las fechas. El *helper* `input_date_tag()` simplifica la introducción de fechas mostrando un calendario interactivo creado con JavaScript, tal y como muestra la figura 10-1. Para ello, se indica la opción `rich` con un valor de `true`.



Figura 10.1. Etiqueta para introducir la fecha mediante un calendario

Si no se utiliza la opción `rich`, el *helper* muestra 3 listas desplegables (`<select>`) cargadas con una serie de meses, días y años. También es posible mostrar por separado cada una de estas listas utilizando sus propios *helpers* (`select_day_tag()`, `select_month_tag()` y `select_year_tag()`). Los valores iniciales de estos elementos son el día, mes y año actuales. El listado 10-3 muestra los *helpers* disponibles para introducir fechas.

Listado 10-3 - *Helpers* para introducir datos

```

<?php echo input_date_tag('fechanacimiento', '2005-05-03', 'rich=true') ?>
=> Muestra un cuadro de texto y un calendario dinámico

```

```
// Los siguientes helpers requieren incluir el grupo de helpers llamado DateForm
<?php use_helper('DateForm') ?>

<?php echo select_day_tag('dia', 1, 'include_custom=Seleccione un día') ?>
=> <select name="dia" id="dia">
    <option value="">Seleccione un día</option>
    <option value="1" selected="selected">01</option>
    <option value="2">02</option>
    ...
    <option value="31">31</option>
</select>

<?php echo select_month_tag('mes', 1, 'include_custom=Seleccione un mes
use_short_month=true') ?>
=> <select name="mes" id="mes">
    <option value="">Seleccione un mes</option>
    <option value="1" selected="selected">Jan</option>
    <option value="2">Feb</option>
    ...
    <option value="12">Dec</option>
</select>

<?php echo select_year_tag('ano', 2007, 'include_custom=Seleccione un año
year_end=2010') ?>
=> <select name="ano" id="ano">
    <option value="">Seleccione un año</option>
    <option value="2006">2006</option>
    <option value="2007" selected="selected">2007</option>
    ...
</select>
```

Los valores permitidos por el *helper* `input_date_tag()` son los mismos que admite la función `strtotime()` de PHP. El listado 10-4 muestra algunos de los listados que se pueden utilizar y el listado 10-5 muestra los que no se pueden emplear.

Listado 10-4 - Formatos de fecha válidos para los *helpers* de fecha

```
// Funcionan bien
<?php echo input_date_tag('prueba', '2006-04-01', 'rich=true') ?>
<?php echo input_date_tag('prueba', 1143884373, 'rich=true') ?>
<?php echo input_date_tag('prueba', 'now', 'rich=true') ?>
<?php echo input_date_tag('prueba', '23 October 2005', 'rich=true') ?>
<?php echo input_date_tag('prueba', 'next tuesday', 'rich=true') ?>
<?php echo input_date_tag('prueba', '1 week 2 days 4 hours 2 seconds', 'rich=true') ?>

// Devuelven un valor null
<?php echo input_date_tag('prueba', null, 'rich=true') ?>
<?php echo input_date_tag('prueba', '', 'rich=true') ?>
```

Listado 10-5 - Formatos de fecha incorrectos para los *helpers* de fecha

```
// Fecha de referencia = 01/01/1970
<?php echo input_date_tag('prueba', 0, 'rich=true') ?>
```

```
// Los formatos que no son válidos en inglés no funcionan
<?php echo input_date_tag('prueba', '01/04/2006', 'rich=true') ?>
```

10.1.4. Editor de textos avanzado

Las áreas de texto definidas mediante `<textarea>` se pueden utilizar como editor de textos avanzado gracias a la integración con las herramientas TinyMCE y FCKEditor. Estos editores muestran una interfaz similar a la de un procesador de textos, incluyendo botones para formatear el texto en negrita, cursiva y otros estilos, tal y como muestra la figura 10-2.

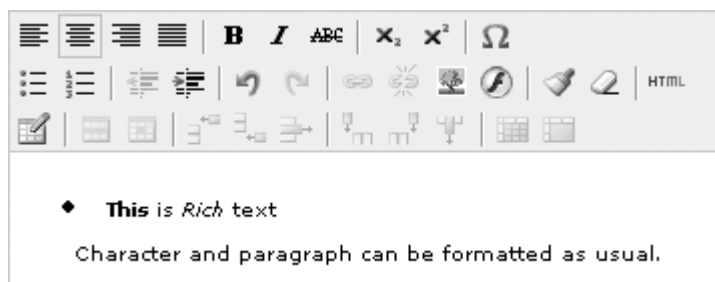


Figura 10.2. Editor de textos avanzado

Los dos editores se tienen que instalar manualmente. Como el proceso es el mismo para los dos, sólo se explica cómo instalar el editor TinyMCE. En primer lugar, se descarga el editor desde la página web del proyecto (<http://tinymce.moxiecode.com/>) y se descomprime en una carpeta temporal. A continuación, se copia el directorio `tinymce/jscripts/tiny_mce/` en la carpeta `web/js/` del proyecto y se define la ruta a la librería en el archivo `settings.yml`, como se muestra en el listado 10-6.

Listado 10-6 - Definiendo la ruta de la librería TinyMCE

```
all:
  .settings:
    rich_text_js_dir: js/tiny_mce
```

Una vez instalado, se puede activar el editor avanzado mediante la opción `rich=true`. También es posible definir opciones propias para el editor JavaScript mediante la opción `tinymce_options`. El listado 10-7 muestra algunos ejemplos.

Listado 10-7 - Editores de texto avanzado

```
<?php echo textarea_tag('nombre', 'valor inicial', 'rich=true size=10x20') ?>
=> se muestra un editor de textos avanzado creado con TinyMCE
<?php echo textarea_tag('nombre', 'valor inicial', 'rich=true size=10x20
tinymce_options=language:"fr",theme_advanced_buttons2:"separator"') ?>
=> se muestra un editor de textos avanzado creado con TinyMCE y personalizado con
opciones propias
```

10.1.5. Selección de idioma, país y moneda

En ocasiones es necesario mostrar un campo de formulario para seleccionar un país. Como el nombre de los países varía en función del idioma en el que se muestran, las opciones de una lista desplegable de países deberían cambiar en función de la cultura del usuario (el Capítulo 13 incluye más información sobre el concepto de *culturas*). Como se muestra en el listado 10-8, el

helper `select_country_tag()` automatiza este proceso: traduce el nombre de todos los países y utiliza como valor los códigos estándar definidos por el ISO.

Listado 10-8 - *Helper* para seleccionar un país

```
<?php echo select_country_tag('pais', 'AL') ?>
=> <select name="pais" id="pais">
    <option value="AF">Afghanistan</option>
    <option value="AL" selected="selected">Albania</option>
    <option value="DZ">Algeria</option>
    <option value="AS">American Samoa</option>
    ...
```

De forma similar a `select_country_tag()`, el *helper* `select_language_tag()` muestra una lista de idiomas, tal y como indica el listado 10-9.

Listado 10-9 - *Helper* para seleccionar un idioma

```
<?php echo select_language_tag('idioma', 'en') ?>
=> <select name="idioma" id="idioma">
    ...
    <option value="elx">Elamite</option>
    <option value="en" selected="selected">English</option>
    <option value="enm">English, Middle (1100-1500)</option>
    <option value="ang">English, Old (ca.450-1100)</option>
    <option value="myv">Erzya</option>
    <option value="eo">Esperanto</option>
    ...
```

El tercer *helper* es `select_currency_tag()`, que muestra una lista de monedas similar a la del listado 10-10.

Listado 10-10 - *Helper* para seleccionar una moneda

```
<?php echo select_currency_tag('moneda', 'EUR') ?>
=> <select name="moneda" id="moneda">
    ...
    <option value="ETB">Ethiopian Birr</option>
    <option value="ETD">Ethiopian Dollar</option>
    <option value="EUR" selected="selected">Euro</option>
    <option value="XBA">European Composite Unit</option>
    <option value="XEU">European Currency Unit</option>
    ...
```

Nota Los tres *helpers* anteriores aceptan un tercer parámetro opcional que corresponde a un array de opciones. Este parámetro se puede utilizar para restringir la lista de opciones que se muestran. En el *helper* de países la opción se llama `countries` y se indica de la siguiente forma: `array('countries' => array ('FR', 'DE'))`. En el *helper* de los idiomas la opción se llama `languages` y en el de las monedas la opción se llama `currencies`.

Restringir la lista completa de opciones a un grupo de valores determinados es muy recomendable porque las listas pueden contener algunos elementos desfasados.

Por último, el *helper* `select_currency_tag()` dispone de otro parámetro opcional llamado `display` que permite controlar la forma en la que se muestran las monedas. Los valores que admite son `symbol`, `code` y `name`.

10.2. Helpers de formularios para objetos

Cuando se utilizan los elementos de formulario para modificar las propiedades de un objeto, resulta tedioso utilizar los *helpers* normales. Por ejemplo, para editar el atributo `telefono` de un objeto `Cliente`, se podría escribir lo siguiente:

```
<?php echo input_tag('telefono', $cliente->getTelefono()) ?>
=> <input type="text" name="telefono" id="telefono" value="0123456789" />
```

Para no tener que repetir continuamente el nombre del atributo, Symfony define un *helper* de formulario para objetos en cada uno de los *helpers* de formularios. Los *helpers* de formularios para objetos deducen el nombre y el valor inicial del elemento a partir de un objeto y del nombre de un método. El anterior `input_tag()` es equivalente a:

```
<?php echo object_input_tag($cliente, 'getTelefono') ?>
=> <input type="text" name="telefono" id="telefono" value="0123456789" />
```

El ahorro de código no es muy significativo para el *helper* `object_input_tag()`. No obstante, todos los *helpers* estándar de formulario disponen del correspondiente *helper* para objetos y todos comparten la misma sintaxis. Utilizando estos *helpers*, es muy sencillo crear los formularios. Esta es la razón por la que los *helpers* de formulario para objetos se utilizan en el *scaffolding* y en los sistemas de gestión creados de forma automática (en el Capítulo 14 se definen los detalles). El listado 10-11 muestra una lista de todos los *helpers* de formularios para objetos.

Listado 10-11 - Sintaxis de los *helpers* de formularios para objetos

```
<?php echo object_input_tag($objeto, $metodo, $opciones) ?>
<?php echo object_input_date_tag($objeto, $metodo, $opciones) ?>
<?php echo object_input_hidden_tag($objeto, $metodo, $opciones) ?>
<?php echo object_textarea_tag($objeto, $metodo, $opciones) ?>
<?php echo object_checkbox_tag($objeto, $metodo, $opciones) ?>
<?php echo object_select_tag($objeto, $metodo, $opciones) ?>
<?php echo object_select_country_tag($objeto, $metodo, $opciones) ?>
<?php echo object_select_language_tag($objeto, $metodo, $opciones) ?>
```

No existe un *helper* llamado `object_password_tag()`, ya que no es recomendable proporcionar un valor por defecto en un campo de texto de contraseña basado en lo que escribió antes el usuario.

Cuidado Al contrario de lo que sucede con los *helpers* de formularios, los *helpers* de formularios para objetos solamente están disponibles si se incluye de forma explícita el grupo de *helpers* llamado `Object` en la plantilla, mediante `use_helper('Object')`.

De todos los *helpers* de formularios para objetos, los más interesantes son `objects_for_select()` y `object_select_tag()`, que se emplean para construir listas desplegables.

10.2.1. Llenando listas desplegables con objetos

El *helper* `options_for_select()`, descrito anteriormente junto con el resto de *helpers* estándar, transforma un array asociativo de PHP en una lista de opciones, como se muestra en el listado 10-12.

Listado 10-12 - Creando una lista de opciones a partir de un array con `options_for_select()`

```
<?php echo options_for_select(array(
    '1' => 'Steve',
    '2' => 'Bob',
    '3' => 'Albert',
    '4' => 'Ian',
    '5' => 'Buck'
), 4) ?>
=> <option value="1">Steve</option>
    <option value="2">Bob</option>
    <option value="3">Albert</option>
    <option value="4" selected="selected">Ian</option>
    <option value="5">Buck</option>
```

Imagina que se dispone de un array de objetos de tipo `Autor` que ha sido obtenido mediante una consulta realizada con `Propel`. Si se quiere mostrar una lista desplegable cuyas opciones se obtienen de ese array, es necesario recorrer el array para obtener el valor del `id` y nombre de cada objeto, tal y como muestra el listado 10-13.

Listado 10-13 - Creando una lista de opciones a partir de un array de objetos con `options_for_select()`

```
// En la acción
$opciones = array();
foreach ($autores as $autor)
{
    $opciones[$autor->getId()] = $autor->getNombre();
}
$this->opciones = $opciones;

// En la plantilla
<?php echo options_for_select($opciones, 4) ?>
```

Como esta técnica es muy habitual, Symfony incluye un *helper* que automatiza todo el proceso llamado `objects_for_select()` y que crea una lista de opciones directamente a partir de un array de objetos. El *helper* requiere 2 parámetros adicionales: los nombres de los métodos empleados para obtener el `value` y el texto de las etiquetas `<option>` que se generan. De esta forma, el listado 10-13 es equivalente a la siguiente línea de código:

```
| <?php echo objects_for_select($autores, 'getId', 'getNombre', 4) ?>
```

Aunque esta instrucción es muy rápida e *inteligente*, Symfony va más allá cuando se emplean claves externas.

10.2.2. Creando una lista desplegable a partir de una columna que es clave externa

Los valores que puede tomar una columna que es clave externa de otra son los valores de una clave primaria que corresponden a una tabla externa. Si por ejemplo se dispone de una tabla llamada `articulo` con una columna `autor_id` que es una clave externa de la tabla `autor`, los posibles valores de esta columna son los de la columna `id` de la tabla `autor`. Básicamente, una lista desplegable para editar el autor de un artículo debería tener el aspecto del listado 10-14.

Listado 10-14 - Creando una lista de opciones a partir de una clave externa con `objects_for_select()`

```
<?php echo select_tag('autor_id', objects_for_select(
    AutorPeer::doSelect(new Criteria()),
    'getId',
    '__toString',
    $articulo->getAutorId()
)) ?>
=> <select name="autor_id" id="autor_id">
    <option value="1">Steve</option>
    <option value="2">Bob</option>
    <option value="3">Albert</option>
    <option value="4" selected="selected">Ian</option>
    <option value="5">Buck</option>
</select>
```

El *helper* `object_select_tag()` automatiza todo el proceso. En el ejemplo anterior se muestra una lista desplegable con el nombre extraído de las filas de la tabla externa. El *helper* puede adivinar el nombre de la tabla y de la columna externa a partir del esquema de base de datos, por lo que su sintaxis es muy concisa. El listado 10-13 es equivalente a la siguiente línea de código:

```
| <?php echo object_select_tag($articulo, 'getAutorId') ?>
```

El *helper* `object_select_tag()` adivina el nombre de la clase *peer* relacionada (`AutorPeer` en este caso) a partir del nombre del método que se pasa como parámetro. No obstante, también es posible indicar una clase propia mediante la opción `related_class` pasada como tercer argumento. El texto que se muestra en cada etiqueta `<option>` es el nombre del registro de base de datos, que es el resultado de aplicar el método `__toString()` a la clase del objeto (si no está definido el método `$autor->__toString()`, se utiliza el valor de la clave primaria). Además, la lista de opciones se obtiene mediante un método `doSelect()` al que se pasa un objeto `Criteria` vacío, por lo que el método devuelve todas las filas de la tabla ordenadas por fecha de creación. Si se necesita mostrar solamente un subconjunto de filas o se quiere realizar un ordenamiento diferente, se crea un método en la clase *peer* que devuelve esa selección en forma de array de objetos y se indica como opción `peer_method` en el *helper*. Por último, es posible añadir una opción vacía o una opción propia como primera opción de la lista desplegable gracias a las opciones `include_blank` y `include_custom`. El listado 10-15 muestra todas estas opciones del *helper* `object_select_tag()`.

Listado 10-15 - Opciones del *helper* `object_select_tag()`

```
// Sintaxis básica
<?php echo object_select_tag($articulo, 'getAutorId') ?>
// Construye La Lista mediante AutorPeer::doSelect(new Criteria())

// Utiliza otra clase peer para obtener Los valores
<?php echo object_select_tag($articulo, 'getAutorId', 'related_class=OtracLase') ?>
// Construye La Lista mediante OtracLasePeer::doSelect(new Criteria())

// Utiliza otro método de la clase peer para obtener Los valores
<?php echo object_select_tag($articulo,
'getAutorId','peer_method=getAutoresMasFamosos') ?>
// Construye La Lista mediante AutorPeer::getAutoresMasFamosos(new Criteria())

// Añade una opción <option value="">&nbsp;</option> al principio de La Lista
<?php echo object_select_tag($articulo, 'getAutorId', 'include_blank=true') ?>

// Añade una opción <option value="">Selecione un autor</option> al principio de La
Lista
<?php echo object_select_tag($articulo, 'getAutorId',
'include_custom=Selecione un autor') ?>
```

10.2.3. Modificando objetos

Las acciones pueden procesar de forma sencilla los formularios que permiten modificar los datos de los objetos utilizando los *helpers* de formularios para objetos. El listado 10-16 muestra un ejemplo de un objeto de tipo Autor con los atributos nombre, edad y dirección.

Listado 10-16 - Un formulario construido con los *helpers* de formularios para objetos

```
<?php echo form_tag('autor/modificar') ?>
  <?php echo object_input_hidden_tag($autor, 'getId') ?>
  Nombre: <?php echo object_input_tag($autor, 'getNombre') ?><br />
  Edad: <?php echo object_input_tag($autor, 'getEdad') ?><br />
  Dirección: <br />
           <?php echo object_textarea_tag($autor, 'getDireccion') ?>
</form>
```

La acción modificar del módulo autor se ejecuta cuando se envía el formulario. Esta acción puede modificar los datos del objeto utilizando el modificador `fromArray()` generado por Propel, tal y como muestra el listado 10-17.

Listado 10-17 - Procesando un formulario realizado con *helpers* de formularios para objetos

```
public function executeModificar($peticion)
{
    $autor = AutorPeer::retrieveByPk($peticion->getParameter('id'));
    $this->forward404Unless($autor);

    $autor->fromArray($this->getRequest()->getParameterHolder()->getAll(),
BasePeer::TYPE_FIELDNAME);
    $autor->save();

    return $this->redirect('/autor/ver?id='.$autor->getId());
}
```


10.3. Validación de formularios

Nota Todas las opciones descritas en esta sección han sido declaradas obsoletas en Symfony 1.1, por lo que sólo están disponibles cuando se activa el plugin `sfCompat10`.

En el Capítulo 6 se explica cómo utilizar los métodos `validateXXX()` en las acciones para validar los parámetros de la petición. Sin embargo, si se utiliza este método para validar los datos enviados en un formulario, se acaba escribiendo una y otra vez los mismos o parecidos trozos de código. Symfony incluye un mecanismo específico de validación de formularios realizado mediante archivos YAML, en vez de utilizar código PHP en la acción.

Para mostrar el funcionamiento de la validación de formularios, se va a utilizar el formulario del listado 10-18. Se trata del típico formulario de contacto que incluye los campos `nombre`, `email`, `edad` y `mensaje`.

Listado 10-18 - Ejemplo de formulario de contacto, en `modules/contacto/templates/indexSuccess.php`

```
<?php echo form_tag('contacto/enviar') ?>
Nombre: <?php echo input_tag('nombre') ?><br />
Email:  <?php echo input_tag('email') ?><br />
Edad:   <?php echo input_tag('edad') ?><br />
Mensaje: <?php echo textarea_tag('mensaje') ?><br />
<?php echo submit_tag() ?>
</form>
```

El funcionamiento básico de la validación en un formulario es que si el usuario introduce datos no válidos y envía el formulario, la próxima página que se muestra debería contener los mensajes de error. La siguiente lista explica con palabras sencillas lo que se consideran datos válidos en el formulario de prueba:

- El campo `nombre` es obligatorio. Debe ser una cadena de texto de entre 2 y 100 caracteres.
- El campo `email` es obligatorio. Debe ser una cadena de texto de entre 2 y 100 caracteres y debe contener una dirección de email válida.
- El campo `edad` es obligatorio. Debe ser un número entero entre 0 y 120.
- El campo `mensaje` es obligatorio.

Se podrían definir reglas de validación más complejas para el formulario de contacto, pero de momento solo es un ejemplo para mostrar las posibilidades de la validación de formularios.

Nota La validación de formularios se puede realizar en el lado del servidor y/o en el lado del cliente. La validación en el servidor es obligatoria para no corromper la base de datos con datos incorrectos. La validación en el lado del cliente es opcional, pero mejora enormemente la experiencia de usuario. La validación en el lado del cliente debe realizarse de forma manual con JavaScript.

10.3.1. Validadores

Los campos nombre y email del formulario de ejemplo comparten las mismas reglas de validación. Como algunas de las reglas de validación son tan comunes que aparecen en todos los formularios, Symfony ha creado unos validadores que encapsulan todo el código PHP necesario para realizarlos. Un validador es una clase que proporciona un método llamado `execute()`. El método requiere de un parámetro que es el valor del campo de formulario y devuelve `true` si el valor es válido y `false` en otro caso.

Symfony incluye varios validadores ya contruidos (que se describen más adelante en la sección "Validadores estándar de Symfony") aunque ahora solo se va a estudiar el validador `sfStringValidator`. Este validador comprueba que el valor introducido es una cadena de texto y que su longitud se encuentra entre 2 límites indicados (definidos cuando se llama al método `initialize()`). Este validador es justo lo que se necesita para validar el campo nombre. El listado 10-19 muestra cómo utilizar este validador en un método de validación.

Listado 10-19 - Validando parámetros de la petición con validadores reutilizables, en `modules/contacto/action/actions.class.php`

```
public function validateEnviar($peticion)
{
    $nombre = $peticion->getParameter('nombre');

    // El campo 'nombre' es obligatorio
    if (!$nombre)
    {
        $this->getRequest()->setError('nombre', 'El campo nombre no se puede dejar vacío');

        return false;
    }

    // El campo nombre debe ser una cadena de texto de entre 2 y 100 caracteres
    $miValidador = new sfStringValidator($this->getContext(), array(
        'min'          => 2,
        'min_error'    => 'El nombre es muy corto (mínimo 2 caracteres)',
        'max'          => 100,
        'max_error'    => 'El nombre es muy largo (máximo 100 caracteres)',
    ));
    if (!$miValidador->execute($nombre, $error))
    {
        return false;
    }

    return true;
}
```

Si un usuario envía el formulario del listado 10-18 con el valor `a` en el campo nombre, el método `execute()` de `sfStringValidator` devuelve un valor `false` (porque la longitud de la cadena de texto es menor que el mínimo de 2 caracteres). El método `validateSend()` devolverá `false` y se ejecutará el método `handleErrorEnviar()` en vez del método `executeEnviar()`.

Sugerencia El método `setError()` del objeto `sfRequest` proporciona información a la plantilla para que se puedan mostrar los mensajes de error, como se explica más adelante en la sección "Mostrando mensajes de error en el formulario". Los validadores establecen los errores de forma interna, por lo que se pueden definir diferentes errores para los diferentes casos de error en la validación. Este es precisamente el objetivo de los parámetros `min_error` y `max_error` de inicialización de `sfStringValidator`.

Las reglas de validación definidas anteriormente se pueden traducir en validadores:

- nombre: `sfStringValidator` (`min=2, max=100`)
- email: `sfStringValidator` (`min=2, max=100`) y `sfEmailValidator`
- edad: `sfNumberValidator` (`min=0, max=120`)

El hecho de que un campo sea requerido no es algo que se controle mediante un validador.

10.3.2. Archivo de validación

Aunque se podría realizar de forma sencilla la validación del formulario de contacto mediante los validadores en el método `validateEnviar()`, esta forma de trabajo supondría repetir mucho código PHP. Symfony ofrece una alternativa mucho mejor para definir las reglas de validación de un formulario, mediante el uso de archivos YAML. El listado 10-20 muestra por ejemplo como realizar la misma validación que el listado 10-19 pero mediante un archivo de validación.

Listado 10-20 - Archivo de validación, en `modules/contacto/validate/enviar.yml`

```
fields:
  name:
    required:
      msg:      El campo nombre no se puede dejar vacío
    sfStringValidator:
      min:      2
      min_error: El nombre es muy corto (mínimo 2 caracteres)
      max:      100
      max_error: El nombre es muy largo (máximo 100 caracteres)
```

En el archivo de validación, la clave `fields` define la lista de campos que tienen que ser validados, si son requeridos o no y los validadores que deben utilizarse para comprobar su validez. Los parámetros de cada validador son los mismos que se utilizan para inicializar manualmente los validadores. Se pueden utilizar tantos validadores como sean necesarios sobre un mismo campo de formulario.

Nota El proceso de validación no termina cuando el validador falla. Symfony ejecuta todos los validadores y determina que la validación ha fallado si al menos uno de ellos falla. Incluso cuando algunas de las reglas de validación fallan, Symfony busca el método `validateXXX()` y lo ejecuta. De esta forma, las 2 técnicas de validación son complementarias. La gran ventaja es que si un formulario tiene muchos errores, se muestran todos los mensajes de error.

Los archivos de validación se encuentran en el directorio `validate/` del módulo y su nombre se corresponde con el nombre de la acción que validan. El listado 10-19 por ejemplo se debe guardar en un archivo llamado `validate/enviar.yml`.

10.3.3. Mostrando el formulario de nuevo

Cuando la validación falla, Symfony por defecto busca un método `handleErrorEnviar()` en la clase de la acción o muestra la plantilla `enviarError.php` si el método no existe.

El procedimiento habitual para informar al usuario de que la validación ha fallado es el de volver a mostrar el formulario con los mensajes de error. Para ello, se debe redefinir el método `handleErrorSend()` para finalizar con una redirección a la acción que muestra el formulario (en este caso `module/index`) tal y como muestra el listado 10-21.

Listado 10-21 - Volviendo a mostrar el formulario, en `modules/contacto/actions/actions.class.php`

```
class ContactoActions extends sfActions
{
    public function executeIndex()
    {
        // Mostrar el formulario
    }

    public function handleErrorEnviar()
    {
        $this->forward('contacto', 'index');
    }

    public function executeEnviar()
    {
        // Procesar el envío del formulario
    }
}
```

Si se utiliza la misma acción para mostrar el formulario y para procesarlo, el método `handleErrorEnviar()` puede devolver el valor `sfView::SUCCESS` para volver a mostrar el formulario, como se indica en el listado 10-22.

Listado 10-22 - Una sola acción para mostrar y procesar el formulario, en `modules/contacto/actions/actions.class.php`

```
class ContactoActions extends sfActions
{
    public function executeEnviar()
    {
        if ($this->getRequest()->getMethod() != sfRequest::POST)
        {
            // Preparar los datos para la plantilla

            // Mostrar el formulario
            return sfView::SUCCESS;
        }
    }
}
```

```

        else
        {
            // Procesar el formulario
            ...
            $this->redirect('mimodulo/otraaccion');
        }
    }
    public function handleErrorEnviar()
    {
        // Preparar los datos para la plantilla

        // Mostrar el formulario
        return sfView::SUCCESS;
    }
}

```

La lógica que se emplea para preparar los datos del formulario se puede refactorizar en un método de tipo `protected` de la clase de la acción, para evitar su repetición en los métodos `executeSend()` y `handleErrorSend()`.

Con esta nueva configuración, cuando el usuario introduce un nombre inválido, se vuelve a mostrar el formulario pero los datos introducidos se pierden y no se muestran los mensajes de error. Para arreglar este último problema, se debe modificar la plantilla que muestra el formulario para insertar los mensajes de error cerca del campo que ha provocado el error.

10.3.4. Mostrando los mensajes de error en el formulario

Cuando un campo del formulario no supera con éxito su validación, los mensajes de error definidos como parámetros del validador se añaden a la petición (de la misma forma que se añadían manualmente mediante el método `setError()` en el listado 10-19). El objeto `sfRequest` proporciona un par de métodos útiles para obtener el mensaje de error: `hasError()` y `getError()`, cada uno de los cuales espera como argumento el nombre de un campo de formulario. Además, se puede mostrar un mensaje de aviso al principio del formulario para llamar la atención del usuario e indicarle que el formulario contiene errores mediante el método `hasErrors()`. Los listados 10-23 y 10-24 muestran cómo utilizar estos métodos.

Listado 10-23 - Mostrando mensajes de error al principio del formulario, en templates/`indexSuccess.php`

```

<?php if ($sf_request->hasErrors()): ?>
    <p>Los datos introducidos no son correctos.
    Por favor, corrija los siguientes errores y vuelva a enviar el formulario:</p>
    <ul>
        <?php foreach($sf_request->getErrors() as $nombre => $error): ?>
            <li><?php echo $nombre ?>: <?php echo $error ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif; ?>

```

Listado 10-24 - Mostrando mensajes de error dentro del formulario, en templates/`indexSuccess.php`

```

<?php echo form_tag('contacto/enviar') ?>
  <?php if ($sf_request->hasError('nombre')): ?>
    <?php echo $sf_request->getError('nombre') ?> <br />
  <?php endif; ?>
  Nombre:   <?php echo input_tag('nombre') ?><br />
  ...
  <?php echo submit_tag() ?>
</form>

```

La condición utilizada antes del método `getError()` en el listado 10-23 es un poco larga de escribir. Por este motivo, Symfony incluye un *helper* llamado `form_error()` y que puede sustituirlo. Para poder utilizarlo, es necesario declarar de forma explícita el uso de este grupo de *helpers* llamado `Validation`. El listado 10-25 modifica al listado 10-24 para utilizar este *helper*.

Listado 10-25 - Mostrando mensajes de error dentro del formulario, forma abreviada

```

<?php use_helper('Validation') ?>
<?php echo form_tag('contacto/enviar') ?>

    <?php echo form_error('nombre') ?><br />
  Nombre: <?php echo input_tag('nombre') ?><br />
  ...
  <?php echo submit_tag() ?>
</form>

```

El *helper* `form_error()` añade por defecto un carácter antes y después del mensaje de error para hacerlos más visibles. Por defecto, el carácter es una flecha que apunta hacia abajo (correspondiente a la entidad `↓`), pero se puede definir otro carácter en el archivo `settings.yml`:

```

all:
  .settings:
    validation_error_prefix:  ' &darr;&nbsp;'
    validation_error_suffix:  ' &nbsp;&darr;'

```

Si ahora falla la validación, el formulario muestra correctamente los mensajes de error, pero los datos introducidos por el usuario se pierden. Para mejorar el formulario es necesario volver a mostrar los datos que introdujo anteriormente el usuario.

10.3.5. Mostrando de nuevo los datos introducidos

Como los errores se manejan mediante el método `forward()` (como se muestra en el listado 10-21), la petición original sigue siendo accesible y por tanto los datos introducidos por el usuario se encuentran en forma de parámetros de la petición. De esta forma, es posible mostrar los datos introducidos en el formulario utilizando los valores por defecto, tal y como se muestra en el listado 10-26.

Listado 10-26 - Indicando valores por defecto para mostrar los datos introducidos por el usuario anteriormente después de un fallo en la validación, en `templates/indexSuccess.php`

```

<?php use_helper('Validation') ?>
<?php echo form_tag('contacto/enviar') ?>

```

```

        <?php echo form_error('nombre') ?><br />
Nombre: <?php echo input_tag('nombre', $sf_params->get('nombre')) ?><br />
        <?php echo form_error('email') ?><br />
Email: <?php echo input_tag('email', $sf_params->get('email')) ?><br />
        <?php echo form_error('edad') ?><br />
Edad: <?php echo input_tag('edad', $sf_params->get('edad')) ?><br />
        <?php echo form_error('mensaje') ?><br />
Mensaje: <?php echo textarea_tag('mensaje', $sf_params->get('mensaje')) ?><br />
        <?php echo submit_tag() ?>
</form>

```

Una vez más, se trata de un mecanismo bastante tedioso de escribir. Symfony ofrece una alternativa para volver a mostrar los datos de todos los campos de un formulario. Esta alternativa se realiza mediante el archivo YAML de validación y no mediante la modificación de los valores por defecto de los elementos. Solamente es necesario activar la opción `fillin`: del formulario, con la sintaxis descrita en el listado 10-27.

Listado 10-27 - Activando la opción `fillin` para volver a mostrar los datos del formulario cuando la validación falla, en `validate/enviar.yml`

```

fillin:
  enabled: true # Habilita volver a mostrar Los datos
  param:
    name: prueba # Nombre del formulario (no es necesario indicarlo si solo hay 1
formulario en la página)
    skip_fields: [email] # No mostrar Los datos introducidos en estos campos
    exclude_types: [hidden, password] # No mostrar Los campos de estos tipos
    check_types: [text, checkbox, radio, select, hidden] # Muestra Los datos de estos
tipos de campos
    content_type: html # html es el formato por defecto. Las otras opciones son xml y
xhtml (esta última es igual que XML, salvo que no se incluye la declaración XML)

```

Por defecto, se vuelven a mostrar los datos de los campos de tipo cuadro de texto, checkbox, radio button, áreas de texto y listas desplegables (sencillas y múltiples). No se vuelven a mostrar los datos en los campos de tipo contraseña y en los campos ocultos. Además, la opción `fillin` no funciona para los campos utilizados para adjuntar archivos.

Nota La opción `fillin` funciona procesando el contenido XML de la respuesta antes de enviarla al usuario. Por defecto los datos se vuelven a mostrar en formato HTML.

Si necesitas mostrar los datos en formato XHTML, la opción `content-type` debe valer `xml`. Además, si la respuesta no es un documento XHTML estrictamente válido, la opción `fillin` puede que no funcione.

El tercer valor posible de la opción `content_type` es `xhtml`, que es idéntico a `xml`, salvo que no incluye la declaración de los archivos XML, lo que evita que se active el modo *quirks* en el navegador Internet Explorer 6.

Antes de volver a mostrar los datos introducidos por el usuario, puede ser necesario modificar sus valores. A los campos del formulario se les pueden aplicar mecanismos de escape, reescritura de URL, transformación de caracteres especiales en entidades y cualquier otra

transformación que se pueda llevar a cabo llamando a una función. Las conversiones se definen bajo la clave `converters:`, como muestra el listado 10-28.

Listado 10-28 - Convirtiendo los datos del usuario antes del `fillin`, en `validate/` `enviar.yml`

```
fillin:
  enabled: true
  param:
    name: prueba
    converters:      # Conversiones aplicadas
    htmlentities:   [nombre, comentarios]
    htmlspecialchars: [comentarios]
```

10.3.6. Validadores estándar de Symfony

Symfony contiene varios validadores ya definidos y que se pueden utilizar directamente en los formularios:

- `sfStringValidator`
- `sfNumberValidator`
- `sfEmailValidator`
- `sfUrlValidator`
- `sfRegexValidator`
- `sfCompareValidator`
- `sfPropelUniqueValidator`
- `sfFileValidator`
- `sfCallbackValidator`

Cada uno dispone de una serie de parámetros y de mensajes de error, pero se pueden redefinir fácilmente mediante el método `initialize()` del validador o mediante el archivo YAML. Las siguientes secciones describen los validadores y muestran ejemplos de su uso.

10.3.6.1. Validador de cadenas de texto

`sfStringValidator` permite establecer una serie de restricciones relacionadas con las cadenas de texto.

```
sfStringValidator:
  values:      [valor1, valor2]
  values_error: Los únicos valores aceptados son valor1 y valor2
  insensitive: false # Si vale true, la comparación con los valores no tiene en
                    cuenta mayúsculas y minúsculas
  min:         2
  min_error:   Por favor, introduce por lo menos 2 caracteres
  max:         100
  max_error:   Por favor, introduce menos de 100 caracteres
```


10.3.6.2. Validador de números

`sfNumberValidator` verifica si un parámetro es un número y permite establecer una serie de restricciones sobre su valor.

```
sfNumberValidator:
  nan_error:    Por favor, introduce un número entero
  min:          0
  min_error:    El valor debe ser como mínimo 0
  max:          100
  max_error:    El valor debe ser inferior o igual a 100
```

10.3.6.3. Validador de email

`sfEmailValidator` verifica si el valor de un parámetro es una dirección válida de email.

```
sfEmailValidator:
  strict:       true
  email_error:  Esta dirección de email no es válida
```

La recomendación RFC822 define el formato de las direcciones de correo electrónico. No obstante, el formato válido es mucho más permisivo que el de las direcciones habituales de email. Según la recomendación, un email como `yo@localhost` es una dirección válida, aunque es una dirección que seguramente será poco útil. Si se establece la opción `strict` a `true` (que es su valor por defecto) solo se consideran válidas las direcciones de correo electrónico con el formato `nombre@dominio.extension`. Si la opción `strict` vale `false`, se utilizan las normas de la recomendación RFC822.

10.3.6.4. Validador de URL

`sfUrlValidator` comprueba si el valor de un campo es una URL válido.

```
sfUrlValidator:
  url_error:    La URL no es válida
```

10.3.6.5. Validador de expresiones regulares

`sfRegexValidator` permite comprar el valor de un campo con una expresión regular compatible con Perl.

```
sfRegexValidator:
  match:         No
  match_error:   Los comentarios con más de una URL se consideran spam
  pattern:       /http.*http/si
```

El parámetro `match` determina si el parámetro debe cumplir el patrón establecido (cuando vale `Yes`) o no debe cumplirlo para considerarse válido (cuando vale `No`).

10.3.6.6. Validador para comparaciones

`sfCompareValidator` compara dos parámetros de petición. Su mayor utilidad es para comparar dos contraseñas.

```

fields:
  password1:
    required:
      msg:      Por favor, introduce una contraseña
  password2:
    required:
      msg:      Por favor, vuelve a introducir la contraseña
  sfCompareValidator:
    check:      password1
    compare_error: Las 2 contraseñas son diferentes

```

El parámetro `check` contiene el nombre del campo cuyo valor debe coincidir con el valor del campo actual para considerarse válido.

Por defecto el validador comprueba que los dos parámetros sean iguales. Se puede utilizar otra comparación indicándola en el parámetro `operator`. Los operadores disponibles son `>`, `>=`, `<`, `<=`, `==` y `!=`.

10.3.6.7. Validador Propel para valores únicos

`sfPropelUniqueValidator` comprueba que el valor de un parámetro de la petición no existe en la base de datos. Se trata de un validador realmente útil para las columnas que deben ser índices únicos.

```

fields:
  nombre:
    sfPropelUniqueValidator:
      class:      Usuario
      column:     login
      unique_error: Ese login ya existe. Por favor, seleccione otro login.

```

En este ejemplo, el validador busca en la base de datos los registros correspondientes a la clase `Usuario` y comprueba si alguna fila tiene en su columna `login` el mismo valor que el parámetro que se pasa al validador.

Cuidado El validador `sfPropelUniqueValidator` puede sufrir problemas de tipo "*condición de carrera*" (*race condition*). Aunque la probabilidad de que ocurra es muy baja, en un entorno multiusuario, el resultado puede cambiar justo cuando se devuelve su valor. Por este motivo, la aplicación debe estar preparada para tratar los errores que se producen con `INSERT` duplicados.

10.3.6.8. Validador de archivos

`sfFileValidator` permite restringir el tipo (mediante un array de mime-types) y el tamaño de los archivos subidos por el usuario.

```

fields:
  image:
    required:
      msg:      Por favor, sube un archivo de imagen
    file:      True
  sfFileValidator:
    mime_types:
      - 'image/jpeg'

```

```

- 'image/png'
- 'image/x-png'
- 'image/jpeg'
mime_types_error: Solo se permiten los formatos PNG y JPEG
max_size:         512000
max_size_error:   El tamaño máximo es de 512Kb

```

El atributo `file` debe valer `True` para ese campo y el formulario de la plantilla debe declararse de tipo `multipart`.

10.3.6.9. Validador de callback

`sfCallbackValidator` delega la validación en un método o función externa. El método que se invoca debe devolver `true` o `false` como resultado de la validación.

```

fields:
  numero_cuenta:
    sfCallbackValidator:
      callback:      is_numeric
      invalid_error: Por favor, introduce un número.
  numero_tarjeta_credito:
    sfCallbackValidator:
      callback:      [misUtilidades, validarTarjetaCredito]
      invalid_error: Por favor, introduce un número correcto de tarjeta de crédito.

```

El método o función que se llama recibe como primer argumento el valor que se debe comprobar. Se trata de un método muy útil cuando se quieren reutilizar los métodos o funciones existentes en vez de tener que volver a crear un código similar para la validación.

Sugerencia También es posible crear validadores propios, como se describe más adelante en la sección "Creando validadores propios".

10.3.7. Validadores con nombre

Si se utilizan de forma constante las mismas opciones para un validador, se pueden agrupar bajo un validador con nombre. En el ejemplo del formulario de contacto, el campo `email` requiere las mismas opciones en `sfStringValidator` que el campo `name`. De esta forma, es posible crear un validador con nombre `miStringValidator` para evitar tener que repetir las mismas opciones. Para ello, se añade una etiqueta `miStringValidator` bajo la clave `validators:`, y se indica la `class` y los `param` del validador que se quiere utilizar. Después, este validador ya se puede utilizar como cualquier otro validador indicando su nombre en la sección `fields`, como se muestra en el listado 10-29.

Listado 10-29 - Reutilizando validadores con nombre en un archivo de validación, en `validate/enviar.yml`

```

validators:
  miStringValidator:
    class: sfStringValidator
    param:
      min:      2
      min_error: Este campo es demasiado corto (mínimo 2 caracteres)

```

```

        max:          100
        max_error: Este campo es demasiado largo (mínimo 100 caracteres)

fields:
  nombre:
    required:
      msg:          El nombre no se puede dejar vacío
      miStringValidator:
  email:
    required:
      msg:          El email no se puede dejar vacío
      miStringValidator:
      sfEmailValidator:
      email_error:  La dirección de email no es válida

```

10.3.8. Restringiendo la validación a un método

Por defecto, los validadores indicados en el archivo de validación se ejecutan cuando la acción se llama mediante un método POST. Se puede redefinir esta opción de forma global o campo a campo especificando otro valor en la clave `methods`, de forma que se pueda utilizar una validación diferente para métodos diferentes, como muestra el listado 10-30.

Listado 10-30 - Definiendo cuando se valida un campo, en `validate/enviar.yml`

```

methods:          [post]      # Opción por defecto

fields:
  nombre:
    required:
      msg:          El nombre no se puede dejar vacío
      miStringValidator:
  email:
    methods:       [post, get] # Redefine la opción global
    required:
      msg:          El email no se puede dejar vacío
      miStringValidator:
      sfEmailValidator:
      email_error:  La dirección de email no es válida

```

10.3.9. ¿Cuál es el aspecto de un archivo de validación?

Hasta ahora solamente se han mostrado partes del archivo de validación. Cuando se juntan todas las partes, las reglas de validación se pueden definir de forma sencilla en el archivo YAML. El listado 10-31 muestra el archivo de validación completo para el formulario de contacto, incluyendo todas las reglas definidas anteriormente.

Listado 10-31 - Ejemplo de archivo de validación completo

```

fillin:
  enabled:        true

validators:
  miStringValidator:
    class: sfStringValidator

```

```

    param:
      min:      2
      min_error: Este campo es demasiado corto (mínimo 2 caracteres)
      max:      100
      max_error: Este campo es demasiado largo (máximo 100 caracteres)

    fields:
      nombre:
        required:
          msg:      El nombre no se puede dejar vacío
        miStringValidator:
      email:
        required:
          msg:      El email no se puede dejar vacío
        myStringValidator:
        sfEmailValidator:
          email_error: La dirección de email no es válida
      edad:
        sfNumberValidator:
          nan_error:  Por favor, introduce un número
          min:        0
          min_error:  "Aun no has nacido, ¿cómo vas a enviar un mensaje?"
          max:        120
          max_error:  "Abuela, ¿no es usted un poco mayor para navegar por Internet?"
      mensaje:
        required:
          msg:      El mensaje no se puede dejar vacío

```

10.4. Validaciones complejas

El archivo de validación es útil en la mayoría de los casos, aunque puede no ser suficiente cuando la validación es muy compleja. En este caso, se puede utilizar el método `validateXXX()` en la acción o se puede utilizar alguna de las soluciones que se presentan a continuación.

10.4.1. Creando un validador propio

Los validadores son clases que heredan de la clase `sfValidator`. Si las clases de validación que incluye Symfony no son suficientes, se puede crear otra clase fácilmente y si se guarda en cualquier directorio `lib/` del proyecto, se cargará automáticamente. La sintaxis es muy sencilla: cuando el validador se ejecuta, se llama al método `execute()`. El método `initialize()` se puede emplear para definir opciones por defecto.

El método `execute()` recibe como primer argumento el valor que se debe comprobar y como segundo argumento, el mensaje de error que se debe mostrar cuando falla la validación. Los dos parámetros se pasan por referencia, por lo que se pueden modificar los mensajes de error directamente en el propio método de validación.

El método `initialize()` recibe el *singleton* del contexto y el array de parámetros del archivo YAML. En primer lugar debe invocar el método `initialize()` de su clase padre `sfValidator` y después, debe establecer los valores por defecto.

Todos los validadores disponen de un contenedor de parámetros accesible mediante `$this->getParameterHolder()`.

Si por ejemplo se quiere definir un validador llamado `sfSpamValidator` para comprobar si una cadena de texto no es spam, se puede utilizar el código del listado 10-32 en un archivo llamado `sfSpamValidator.class.php`. El validador comprueba si `$valor` contiene más de `max_url` veces la cadena de texto `http`.

Listado 10-32 - Creando un validador propio, en `lib/sfSpamValidator.class.php`

```
class sfSpamValidator extends sfValidator
{
    public function execute(&$valor, &$error)
    {
        // Para max_url=2, la expresión regular es /http.*http/is
        $re = '/'.implode('.', array_fill(0, $this->getParameter('max_url') + 1, 'http')).'/is';

        if (preg_match($re, $valor))
        {
            $error = $this->getParameter('spam_error');

            return false;
        }

        return true;
    }

    public function initialize ($contexto, $parametros = null)
    {
        // Inicializar la clase padre
        parent::initialize($contexto);

        // Valores por defecto de los parámetros
        $this->setParameter('max_url', 2);
        $this->setParameter('spam_error', 'Esto es spam');

        // Establecer los parámetros
        $this->getParameterHolder()->add($parametros);

        return true;
    }
}
```

Después de incluir el validador en cualquier directorio con carga automática de clases (y después de borrar la cache de Symfony) se puede utilizar en los archivos de validación de la forma que muestra el listado 10-33.

Listado 10-33 - Utilizando un validador propio, en `validate/enviar.yml`

```
fields:
  mensaje:
    required:
      msg:          El mensaje no se puede dejar vacío
```

```
sfSpamValidator:
  max_url:      3
  spam_error:   En este sitio web no nos gusta el spam
```

10.4.2. Utilizando la sintaxis de los arrays para los campos de formulario

PHP permite utilizar la sintaxis de los arrays para los campos de formulario. Cuando se diseñan manualmente los formularios o cuando se utilizan los que genera automáticamente Propel (ver Capítulo 14) el código HTML resultante puede ser similar al del listado 10-34.

Listado 10-34 - Formulario con sintaxis de array

```
<label for="articulo_titulo">Titulo:</label>
<input type="text" name="articulo[titulo]" id="articulo_titulo" value="Valor inicial"
      size="45" />
```

Si en un archivo de validación se utiliza el nombre del campo de formulario tal y como aparece en el formulario (con los corchetes) se producirá un error al procesar el archivo YAML. La solución consiste en reemplazar los corchetes [] por llaves {} en la sección `fields`, como muestra el listado 10-35. Symfony se encarga de la conversión de los nombres que se envían después a los validadores.

Listado 10-35 - Archivo de validación para un formulario que utiliza la sintaxis de los arrays

```
fields:
  articulo{titulo}:
    required:      Yes
```

10.4.3. Ejecutando un validador en un campo vacío

En ocasiones es necesario ejecutar un validador a un campo que no es obligatorio, es decir, en un campo que puede estar vacío. El caso más habitual es el de un formulario en el que el usuario puede (pero no es obligatorio) cambiar su contraseña. Si decide cambiarla, debe escribir la nueva contraseña dos veces. El ejemplo se muestra en el listado 10-36.

Listado 10-36 - Archivo de validación para un formulario con 2 campos de contraseña

```
fields:
  password1:
  password2:
    sfCompareValidator:
      check:      password1
      compare_error: Las 2 contraseñas no coinciden
```

La validación que se ejecuta es la siguiente:

- Si `password1 == null` y `password2 == null`:
 - La comprobación `required` se cumple.
 - Los validadores no se ejecutan.
 - El formulario es válido.

- Si `password2 == null` y `password1` no es `null`:
 - La comprobación `required` se cumple.
 - Los validadores no se ejecutan.
 - El formulario es válido.

El validador para `password2` debería ejecutarse si `password1` es `not null`. Afortunadamente, los validadores de Symfony permiten controlar este caso gracias al parámetro `group`. Cuando un campo de formulario pertenece a un grupo, su validador se ejecuta si el campo no está vacío y si alguno de los campos que pertenecen al grupo no está vacío.

Así que si se modifica la configuración del proceso de validación por lo que se muestra en el listado 10-37, la validación se ejecuta correctamente.

Listado 10-37 - Archivo de validación para un formulario con 2 campos de contraseña y un grupo

```
fields:
  password1:
    group:          grupo_password
  password2:
    group:          grupo_password
    sfCompareValidator:
      check:         password1
      compare_error: Las 2 contraseñas no coinciden
```

El proceso de validación ahora se ejecuta de la siguiente manera:

- Si `password1 == null` y `password2 == null`:
 - La comprobación `required` se cumple.
 - Los validadores no se ejecutan.
 - El formulario es válido.
- Si `password1 == null` and `password2 == lo_que_sea`:
 - La comprobación `required` se cumple.
 - `password2` es `not null`, por lo que se ejecuta su validador y falla.
 - Se muestra un mensaje de error para `password2`.
- Si `password1 == lo_que_sea` y `password2 == null`:
 - La comprobación `required` se cumple.
 - `password1` es `not null`, por lo que se ejecuta también el validador para `password2` por pertenecer al mismo grupo y la validación falla.
 - Se muestra un mensaje de error para `password2`.
- Si `password1 == lo_que_sea` y `password2 == lo_que_sea`:
 - La comprobación `required` se cumple.

- `password2` es `not null`, por lo que se ejecuta su validador y no se producen errores.
- El formulario es válido.

10.5. Resumen

Incluir formularios en las plantillas es muy sencillo gracias a los *helpers* de formularios que incluye Symfony y a sus opciones avanzadas. Si se definen formularios para modificar las propiedades de un objeto, los *helpers* de formularios para objetos simplifican enormemente su desarrollo. Los archivos de validación, los *helpers* de validación y la opción de volver a mostrar los datos en un formulario, permiten reducir el esfuerzo necesario para crear un control estricto de los formularios que sea robusto y a la vez fácil de utilizar por parte de los usuarios. Además, cualquier validación por muy compleja que sea se puede realizar escribiendo un validador propio o utilizando un método `validateXXX()` en la clase de la acción.

Capítulo 11. Integración con Ajax

Las aplicaciones de la denominada Web 2.0 incluyen numerosas interacciones en el lado del cliente, efectos visuales complejos y comunicaciones asíncronas con los servidores. Todo lo anterior se realiza con JavaScript, pero programarlo manualmente es una tarea tediosa y que requiere de mucho tiempo para corregir los posibles errores. Afortunadamente, Symfony incluye una serie de *helpers* que automatizan muchos de los usos comunes de JavaScript en las plantillas. La mayoría de comportamientos en el lado del cliente se pueden programar sin necesidad de escribir ni una sola línea de JavaScript. Los programadores solo tienen que ocuparse del efecto que quieren incluir y Symfony se encarga de lidiar con la sintaxis necesaria y con las posibles incompatibilidades entre navegadores.

En este capítulo se describen las herramientas proporcionadas por Symfony para facilitar la programación en el lado del cliente:

- Los *helpers* básicos de JavaScript producen etiquetas `<script>` válidas según los estándares XHTML, para actualizar elementos DOM (Document Object Model) o para ejecutar un script mediante un enlace.
- Prototype es una librería de JavaScript completamente integrada en Symfony y que simplifica el desarrollo de scripts mediante la definición de nuevas funciones y métodos de JavaScript.
- Los *helpers* de Ajax permiten al usuario actualizar partes de la página web pinchando sobre un enlace, enviando un formulario o modificando un elemento de formulario.
- Todos estos *helpers* disponen de múltiples opciones que proporcionan una mayor flexibilidad, sobre todo mediante el uso de las funciones de tipo *callback*.
- Script.aculo.us es otra librería de JavaScript que también está integrada en Symfony y que añade efectos visuales dinámicos que permiten mejorar la interfaz y la experiencia de usuario.
- JSON (JavaScript Object Notation) es un estándar utilizado para que un script de cliente se comunique con un servidor.
- Las aplicaciones Symfony también permiten definir interacciones complejas en el lado del cliente, combinando todos los elementos anteriores. Mediante una sola línea de código PHP (la llamada al *helper* de Symfony) es posible incluir las opciones de autocompletado, arrastrar y soltar, listas ordenables dinámicamente y texto editable.

11.1. Helpers básicos de JavaScript

JavaScript siempre se había considerado como poco útil en el desarrollo de aplicaciones web profesionales debido a sus problemas de incompatibilidad entre distintos navegadores. Hoy en día, se han resuelto la mayoría de incompatibilidades y se han creado librerías muy completas que permiten programar interacciones complejas de JavaScript sin necesidad de programar

cientos de líneas de código y sin perder cientos de horas corrigiendo problemas. El avance más popular se llama Ajax, como se explica más adelante en la sección "Helpers de Ajax".

Sorprendentemente, en este capítulo casi no se incluye código JavaScript. La razón es que Symfony permite la programación de scripts del lado del cliente de forma diferente: encapsula y abstrae toda la lógica JavaScript en *helpers*, por lo que las plantillas no incluyen código JavaScript. Para el programador, añadir cierta lógica a un elemento de la página solo requiere de una línea de código PHP, pero la llamada a este *helper* produce código JavaScript, cuya complejidad se puede comprobar al ver el código fuente de la página generada como respuesta. Los *helpers* se encargan de resolver los problemas de incompatibilidades entre navegadores por lo que la cantidad de código JavaScript que generan puede ser muy importante. Por tanto, en este capítulo se muestra como realizar los efectos que normalmente se programan manualmente con JavaScript sin necesidad de utilizar JavaScript.

Todos los *helpers* descritos se encuentran disponibles en las plantillas siempre que se declare de forma explícita el uso del *helper* llamado Javascript.

```
| <?php use_helper('Javascript') ?>
```

Algunos de estos *helpers* generan código HTML y otros generan directamente código JavaScript.

11.1.1. JavaScript en las plantillas

En XHTML, los bloques de código JavaScript deben encerrarse en secciones CDATA. Por eso es tedioso crear páginas que tienen muchos bloques de código JavaScript. Symfony incluye un *helper* llamado `javascript_tag()` y que transforma una cadena de texto en una etiqueta `<script>` válida según los estándares XHTML. El listado 11-1 muestra el uso de este *helper*.

Listado 11-1 - Incluyendo JavaScript con el *helper* `javascript_tag()`

```
<?php echo javascript_tag("
    function mifuncion()
    {
        ...
    }
") ?>
=> <script type="text/javascript">
    //
        function mifuncion()
        {
            ...
        }
    //]]&gt;
&lt;/script&gt;</pre></div><div data-bbox="138 793 907 846" data-label="Text"><p>El uso habitual de JavaScript, más que sus bloques de código, es la definición de enlaces que ejecutan un determinado script cuando se pincha en ellos. El <i>helper</i> <code>link_to_function()</code> se encarga exactamente de eso, como muestra el listado 11-2.</p></div><div data-bbox="138 858 901 874" data-label="Section-Header"><h5>Listado 11-2 - Ejecutando JavaScript mediante un enlace con el <i>helper</i> <code>link_to_function()</code></h5></div><div data-bbox="138 924 270 938" data-label="Page-Footer">www.librosweb.es</div><div data-bbox="866 927 907 942" data-label="Page-Footer">219</div>
```

```
<?php echo link_to_function('¡Pínchame!', "alert('Me has pinchado')") ?>
=> <a href="#" onClick="alert('Me has pinchado'); return none;">¡Pínchame!</a>
```

Como sucede con el *helper* `link_to()`, se pueden añadir opciones a la etiqueta `<a>` generada mediante un tercer argumento de la función.

Nota De la misma forma que el *helper* `link_to()` tiene una función relacionada llamada `button_to()`, también es posible ejecutar un script al pulsar un botón (`<input type="button">`) utilizando el *helper* `button_to_function()`. Si se necesita una imagen pinchable, se puede llamar a `link_to_function(image_tag('mi_imagen'), "alert('Me has pinchado')")`.

11.1.2. Actualizando un elemento DOM

Una de las tareas habituales de las interfaces dinámicas es la actualización de algunos elementos de la página. Normalmente se realiza como se muestra en el listado 11-3.

Listado 11-3 - Actualizando un elemento con JavaScript

```
<div id="indicador">Comienza el procesamiento de datos</div>
<?php echo javascript_tag(
    document.getElementById("indicador").innerHTML =
        "<strong>El procesamiento de datos ha concluido</strong>";
    "> ?>
```

Symfony incluye un *helper* que realiza esta tarea y que genera código JavaScript (no HTML). El *helper* se denomina `update_element_function()` y el listado 11-4 muestra su uso.

Listado 11-4 - Actualizar un elemento mediante JavaScript con el *helper* `update_element_function()`

```
<div id="indicador">Comienza el procesamiento de datos</div>
<?php echo javascript_tag(
    update_element_function('indicador', array(
        'content' => "<strong>El procesamiento de datos ha concluido</strong>",
    ))
) ?>
```

A primera vista parece que este *helper* no es muy útil, ya que el código necesario es tan largo como el código JavaScript original. En realidad su ventaja es la facilidad de lectura del código. Si lo que se necesita es insertar el contenido antes o después de un elemento, eliminarlo en vez de actualizarlo o no hacer nada si no se cumple una condición, el código JavaScript resultante es muy complicado. Sin embargo, el *helper* `update_element_function()` permite mantener la facilidad de lectura del código de la plantilla, tal y como se muestra en el listado 11-5.

Listado 11-5 - Opciones del *helper* `update_element_function()`

```
// Insertar el contenido después del elemento 'indicador'
update_element_function('indicador', array(
    'position' => 'after',
    'content' => "<strong>El procesamiento de datos ha concluido</strong>",
));

// Eliminar el elemento anterior a 'indicador', solo si $condicion vale true
update_element_function('indicador', array(
```

```

    'action'    => $condicion ? 'remove' : 'empty',
    'position' => 'before',
  ))

```

El *helper* permite que el código de las plantillas sea más fácil de entender que el código JavaScript, además de proporcionar una sintaxis unificada para efectos similares. También esa es la razón por la que el nombre del *helper* es tan largo: su nombre es tan explícito que no hace falta añadir comentarios que lo expliquen.

11.1.3. Aplicaciones que se degradan correctamente

La etiqueta `<noscript>` permite especificar cierto código HTML que muestran los navegadores que no tienen soporte de JavaScript. Symfony complementa esta etiqueta con un *helper* que funciona de forma inversa: asegura que cierto código solo se ejecuta en los navegadores que soportan JavaScript. Los *helpers* `if_javascript()` y `end_if_javascript()` permiten crear aplicaciones que se degradan correctamente en los navegadores que no soportan JavaScript, como muestra el listado 11-6.

Listado 11-6 - Uso del *helper* `if_javascript()` para que la aplicación se degrade correctamente

```

<?php if_javascript(); ?>
  <p>Tienes activado JavaScript.</p>
<?php end_if_javascript(); ?>

<noscript>
  <p>No tienes activado JavaScript.</p>
</noscript>

```

Nota No es necesario incluir instrucciones `echo` cuando se llama a los *helpers* `if_javascript()` y `end_if_javascript()`.

11.2. Prototype

Prototype es una librería de JavaScript muy completa que amplía las posibilidades del lenguaje de programación, añade todas esas funciones que faltaban y con las que los programadores soñaban y ofrece nuevos mecanismos para la manipulación de los elementos DOM. El sitio web del proyecto es <http://prototypejs.org/>.

Los archivos de Prototype se incluyen con el framework Symfony y son accesibles en cualquier nuevo proyecto, en la carpeta `web/sf/prototype/`. Por tanto, se puede utilizar Prototype añadiendo el siguiente código a la acción:

```

$directorioPrototype = sfConfig::get('sf_prototype_web_dir');
$this->getResponse()->addJavascript($directorioPrototype.'/js/prototype');

```

También se puede añadir con el siguiente cambio en el archivo `view.yml`:

```

all:
  javascripts: [%SF_PROTOTYPE_WEB_DIR%/js/prototype]

```

Nota Como los *helpers* de Ajax de Symfony, que se describen en la siguiente sección, dependen de Prototype, la librería Prototype se incluye automáticamente cuando se utiliza cualquiera de

ellos. Por tanto, no es necesario añadir los archivos JavaScript de Prototype a la respuesta si la plantilla hace uso de cualquier *helper* cuyo nombre acaba en `_remote`.

Una vez que la librería Prototype se ha cargado, se pueden utilizar todas las funciones nuevas que añade al lenguaje JavaScript. El objetivo de este libro no es describir esas nuevas funciones, pero es fácil encontrar buena documentación de Prototype en la web, como por ejemplo:

- Particletree (<http://particletree.com/features/quick-guide-to-prototype/>)
- Sergio Pereira (<http://www.sergiopereira.com/articles/prototype.js.html>)
- Script.aculo.us (<http://wiki.script.aculo.us/scriptaculous/show/Prototype>)

Una de las funciones que Prototype añade a JavaScript es la función *dólar*, `$()`. Básicamente se trata de un atajo de la función `document.getElementById()`, pero tiene más posibilidades. El listado 7-11 muestra un ejemplo de su uso.

Listado 11-7 - Uso de la función `$()` para obtener un elemento a partir de su ID con JavaScript

```
nodo = $('elementoID');  
  
// Es equivalente a...  
nodo = document.getElementById('elementoID');  
  
// Puede obtener más de un elemento a la vez  
// En este caso, el resultado es un array de elementos DOM  
nodos = $('primerDiv', 'segundoDiv');
```

Prototype también incluye una función que no dispone JavaScript y que devuelve un array de todos los elementos DOM que tienen un valor del atributo `class` igual al indicado como argumento:

```
| nodos = document.getElementsByClassName('miclass');
```

No obstante, no se suele utilizar la función anterior, ya que Prototype incluye una función mucho más poderosa llamada *doble dólar*, `$$()`. Esta función devuelve un array con todos los elementos DOM seleccionados mediante un selector de CSS. La función anterior es equivalente por tanto a la siguiente:

```
| nodos = $$('.miclass');
```

Gracias al poder de los selectores CSS, se pueden procesar los nodos DOM mediante su `class`, su `id` y mediante selectores avanzados como el descendiente (padre-hijo) y el relacional (anterior-siguiente), mucho más fácilmente que como se haría mediante Xpath. Incluso es posible combinar todos los selectores CSS para seleccionar los elementos DOM mediante esta función:

```
| nodos = $$('body div#principal ul li.ultimo img > span.leyenda');
```

Un último ejemplo de las mejoras en la sintaxis de JavaScript proporcionadas por Prototype es el iterador de arrays llamado `each`. Permite un código tan conciso como PHP y con la posibilidad

añadida de definir funciones anónimas y *closures* de JavaScript. Se trata de un truco muy útil si se programa JavaScript manualmente.

```
var verduras = ['Zanahorias', 'Lechuga', 'Ajo'];  
verduras.each(function(comida) { alert('Me encanta ' + comida); });
```

Como programar JavaScript con Prototype es mucho más divertido que hacerlo sin su ayuda y como Prototype es parte de Symfony, es conveniente dedicar el tiempo necesario para leer su documentación antes de continuar.

11.3. Helpers de Ajax

¿Qué sucede si se quiere actualizar un elemento de la página no con JavaScript como en el listado 11-5, sino mediante un script de PHP que se encuentra en el servidor? De esta forma, sería posible modificar parte de la página en función de una respuesta del servidor. El *helper* `remote_function()` realiza exactamente esa tarea, como se demuestra en el listado 11-8.

Listado 11-8 - Uso del *helper* `remote_function()`

```
<div id="mizona"></div>  
<?php echo javascript_tag(  
    remote_function(array(  
        'update' => 'mizona',  
        'url'    => 'mimodulo/miaccion',  
    ))  
) ?>
```

Nota El parámetro `url` puede contener una URI interna (`modulo/accion?clave1=valor1&...`) o el nombre de una regla del sistema de enrutamiento, al igual que sucede con el *helper* `url_for()`.

Cuando se ejecuta, el script anterior actualiza el contenido del elemento cuyo `id` es igual a `mizona` con la respuesta de la acción `mimodulo/miaccion`. Este tipo de interacción se llama Ajax, y es el núcleo de las aplicaciones web más interactivas. La versión en inglés de la Wikipedia (<http://en.wikipedia.org/wiki/AJAX>) lo describe de la siguiente manera:

Ajax permite que las páginas web respondan de forma más rápida mediante el intercambio en segundo plano de pequeñas cantidades de datos con el servidor, por lo que no es necesario recargar la página entera cada vez que el usuario realiza un cambio. El objetivo es aumentar la interactividad, la rapidez y la usabilidad de la página.

Ajax depende de `XMLHttpRequest`, un objeto JavaScript cuyo comportamiento es similar a un *frame* oculto, cuyo contenido se puede actualizar realizando una petición al servidor y se puede utilizar para manipular el resto de la página web. Se trata de un objeto a muy bajo nivel, por lo que los navegadores lo tratan de forma diferente y el resultado es que se necesitan muchas líneas de código para realizar peticiones Ajax a mano. Afortunadamente, Prototype encapsula todo el código necesario para trabajar con Ajax y proporciona un objeto Ajax mucho más simple y que también utiliza Symfony. Este es el motivo por el que la librería Prototype se carga automáticamente cuando se utiliza un *helper* de Ajax en la plantilla.

Sugerencia Los *helpers* de Ajax no funcionan si la URL de la acción remota no pertenece al mismo dominio que la página web que la llama. Se trata de una restricción por motivos de seguridad que imponen los navegadores y que no puede saltarse.

Las interacciones de Ajax están formadas por tres partes: el elemento que la invoca (un enlace, un formulario, un botón, un contador de tiempo o cualquier otro elemento que el usuario manipula e invoca la acción), la acción del servidor y una zona de la página en la que mostrar la respuesta de la acción. Se pueden crear interacciones más complejas si por ejemplo la acción remota devuelve datos que se procesan en una función JavaScript en el navegador del cliente. Symfony incluye numerosos *helpers* para insertar interacciones Ajax en las plantillas y todos contienen la palabra *remote* en su nombre. Además, todos comparten la misma sintaxis, un array asociativo con todos los parámetros de Ajax. Debe tenerse en cuenta que los *helpers* de Ajax generan código HTML, no código JavaScript.

Las acciones que se invocan de forma remota no dejan de ser acciones normales y corrientes. Se les aplica el sistema de enrutamiento, determinan la vista que deben generar en función del valor que devuelven, pasan variables a sus plantillas y pueden modificar el modelo como cualquier otra acción.

Sin embargo, cuando se invocan mediante Ajax, las acciones devuelven el valor `true` a la siguiente función:

```
| $esAjax = $this->getRequest()->isXmlHttpRequest();
```

Symfony es capaz de darse cuenta de que una acción se está ejecutando en un contexto Ajax y puede adaptar la respuesta de forma adecuada. Por tanto, y por defecto, las acciones Ajax no incluyen la barra de depuración de aplicaciones ni siquiera en el entorno de desarrollo. Además, no aplican el proceso de decoración (es decir, sus plantillas no se insertan por defecto en el layout correspondiente). Si se necesita decorar la vista de una acción Ajax, se debe indicar explícitamente la opción `has_layout: true` para su vista en el archivo `view.yml`.

Como el tiempo de respuesta es crucial en las interacciones Ajax, si la respuesta es sencilla, es una buena idea no crear la vista completa y devolver la respuesta directamente en forma de texto. Se puede utilizar por tanto el método `renderText()` en la acción para no utilizar la plantilla y mejorar el tiempo de respuesta de las peticiones Ajax.

La mayoría de acciones Ajax finalizan con una plantilla que simplemente incluye un elemento parcial, porque el código de la respuesta Ajax ya se ha utilizado para mostrar la página inicial. Para evitar tener que crear una plantilla sólo para una línea de código, a partir de Symfony 1.1 la acción puede utilizar el método `renderPartial()`. Este método se aprovecha de las ventajas de la reutilización de los elementos parciales, sus posibilidades de cache y la velocidad de ejecución del método `renderText()`.

```
| public function executeMiAccion()
| {
|     // Código PHP de la acción
|     return $this->renderPartial('mimodulo/miparcial');
| }
```


11.3.1. Enlaces Ajax

Los enlaces Ajax constituyen una de las partes más importantes de las interacciones Ajax realizadas en las aplicaciones de la Web 2.0. El *helper* `link_to_remote()` muestra un enlace que llama a una función remota. La sintaxis es muy similar a `link_to()`, excepto que el segundo parámetro es el array asociativo con las opciones Ajax, como muestra el listado 11-9.

Listado 11-9 - Enlace Ajax realizado con el *helper* `link_to_remote()`

```
<div id="respuesta"></div>
<?php echo link_to_remote('Borrar este post', array(
    'update' => 'respuesta',
    'url'     => 'post/borrar?id='.$post->getId(),
)) ?>
```

En el ejemplo anterior, al pulsar sobre el enlace "Borrar este post" se realiza una llamada en segundo plano a la acción `post/borrar`. La respuesta devuelta por el servidor se muestra automáticamente en el elemento de la página cuyo atributo `id` sea igual a `respuesta`. La figura 11-1 ilustra el proceso completo.



Figura 11.1. Ejecutando una actualización remota mediante un enlace

También es posible utilizar una imagen en vez de texto para mostrar el enlace, utilizar el nombre de una regla de enrutamiento en vez de `modulo/accion` y añadir opciones a la etiqueta `<a>` como tercer argumento, tal y como muestra el listado 11-10.

Listado 11-10 - Opciones del *helper* `link_to_remote()`

```
<div id="emails"></div>
<?php echo link_to_remote(image_tag('refresh'), array(
    'update' => 'emails',
    'url'     => '@listado_emails',
), array(
    'class' => 'enlace_ajax',
)) ?>
```

11.3.2. Formularios Ajax

Los formularios web normalmente realizan una llamada a una acción que provoca que se deba recargar la página completa. El *helper* equivalente a `link_to_function()` para un formulario sería un *helper* que enviara los datos del formulario al servidor y que actualizara un elemento de la página con la respuesta del servidor. Eso es precisamente lo que hace el *helper* `form_remote_tag()`, y su sintaxis se muestra en el listado 11-11.

Listado 11-11 - Formulario Ajax con el *helper* `form_remote_tag()`

```

<div id="lista_elementos"></div>
<?php echo form_remote_tag(array(
    'update'    => 'lista_elementos',
    'url'       => 'elemento/anadir',
)) ?>
<label for="elemento">Elemento:</label>
<?php echo input_tag('elemento') ?>
<?php echo submit_tag('Añadir') ?>
</form>

```

El *helper* `form_remote_tag()` crea una etiqueta `<form>` de apertura, como sucede con el *helper* `form_tag()`. El envío del formulario consiste en el envío en segundo plano de una petición de tipo POST a la acción `elemento/anadir` y con la variable `elemento` como parámetro de la petición. La respuesta del servidor reemplaza los contenidos del elemento cuyo atributo `id` sea igual a `lista_elementos`, como se muestra en la figura 11-2. Los formularios Ajax se cierran con una etiqueta `</form>` de cierre de formularios.

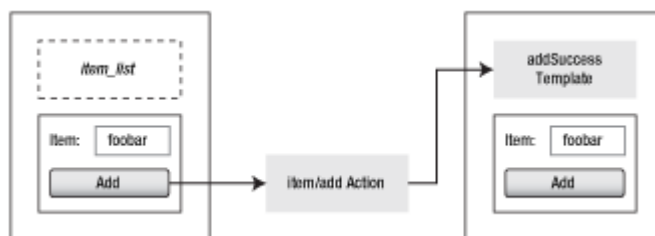


Figura 11.2. Ejecutando una actualización remota mediante un formulario

Cuidado Los formularios Ajax no pueden ser multipart, debido a una limitación del objeto `XMLHttpRequest`. En otras palabras, no es posible enviar archivos mediante formularios Ajax. Existen algunas técnicas para saltarse esta limitación, como por ejemplo utilizar un `iframe` oculto en vez del objeto `XMLHttpRequest`.

Si es necesario incluir un formulario que sea *normal* y Ajax a la vez, lo mejor es definirlo como formulario *normal* y añadir, además del botón de envío tradicional, un segundo botón (`<input type="button" />`) para enviar el formulario mediante Ajax. Symfony define este botón mediante el *helper* `submit_to_remote()`. De esta forma, es posible definir interacciones Ajax que se degradan correctamente en los navegadores que no las soportan. El listado 11-12 muestra un ejemplo.

Listado 11-12 - Formulario con envío de datos tradicional y Ajax

```

<div id="lista_elementos"></div>
<?php echo form_tag('@elemento_anadir_normal') ?>
<label for="elemento">Elemento:</label>
<?php echo input_tag('elemento') ?>
<?php if_javascript(); ?>
    <?php echo submit_to_remote('envio_ajax', 'Anadir con Ajax', array(
        'update'    => 'lista_elementos',
        'url'       => '@elemento_anadir',
    )) ?>
<?php end_if_javascript(); ?>
<noscript>
    <?php echo submit_tag('Anadir') ?>
</noscript>

```

```

    </noscript>
  </form>

```

Otro ejemplo en el que se podría utilizar la combinación de botones normales y botones Ajax es el de un formulario que edita un artículo o noticia. Podría incluir un botón realizado con Ajax para previsualizar los contenidos y un botón normal para publicar los contenidos directamente.

Nota Si el usuario envía el formulario pulsando la tecla Enter, el formulario se envía utilizando la acción definida en la etiqueta `<form>` principal, es decir, la acción *normal* y no la acción Ajax.

Los formularios más modernos no solo se encargan de enviar sus datos cuando el usuario pulsa sobre el botón de envío, sino que también pueden reaccionar a los cambios producidos por el usuario sobre alguno de sus campos. Symfony proporciona el *helper* `observe_field()` para realizar esa tarea. El listado 11-13 muestra un ejemplo de uso de este *helper* para crear un sistema que sugiere valores a medida que el usuario escribe sobre un campo: cada carácter escrito en el campo elemento lanza una petición Ajax que actualiza el valor del elemento `sugerencias_elemento` de la página.

Listado 11-13 - Ejecutando una función remota cada vez que cambia el valor de un campo de formulario mediante `observe_field()`

```

<?php echo form_tag('@elemento_anadir_normal') ?>
  <label for="elemento">Elemento:</label>
  <?php echo input_tag('elemento') ?>
  <div id="sugerencias_elemento"></div>
  <?php echo observe_field('elemento', array(
    'update'    => 'sugerencias_elemento',
    'url'       => '@elemento_escrito',
  )) ?>
  <?php echo submit_tag('Anadir') ?>
</form>

```

El par `modulo/accion` correspondiente a la regla `@elemento_escrito` se ejecuta cada vez que el usuario modifica el valor del campo de formulario que se está observando (en este caso, `elemento`) sin necesidad de enviar el formulario. La acción puede acceder a los caracteres escritos en cada momento por el usuario mediante el parámetro `elemento` de la petición. Si se necesita enviar otro valor en vez del contenido del campo de formulario que se está observando, se puede especificar en forma de expresión JavaScript en el parámetro `with`. Si por ejemplo es necesario que la acción disponga de un parámetro llamado `param`, se puede utilizar el *helper* `observe_field()` como muestra el listado 11-14.

Listado 11-14 - Pasando parámetros personalizados a la acción remota con la opción `with`

```

<?php echo observe_field('elemento', array(
  'update'    => 'sugerencias_elemento',
  'url'       => '@elemento_escrito',
  'with'      => "'param=' + value",
)) ?>

```

Este *helper* no genera un elemento HTML, sino que añade un *comportamiento* (del inglés, *"behavior"*) al elemento que se pasa como parámetro. Más adelante en este capítulo se describen más ejemplos de *helpers* de JavaScript que añaden comportamientos.

Si se quieren observar todos los campos de un formulario, se puede utilizar el *helper* `observe_form()`, que llama a una función remota cada vez que se modifica uno de los campos del formulario.

11.3.3. Ejecución periódica de funciones remotas

Por último, el *helper* `periodically_call_remote()` permite crear una interacción de Ajax que se repite cada pocos segundos. No está asociado con ningún elemento HTML de la página, sino que se ejecuta de forma transparente en segundo plano como una especie de comportamiento de la página entera. Se puede utilizar para seguir la posición del puntero del ratón, autoguardar el contenido de un área de texto grande, etc. El listado 11-15 muestra un ejemplo de uso de este *helper*.

Listado 11-15 - Ejecutando periódicamente una función remota mediante `periodically_call_remote()`

```
<div id="notificacion"></div>
<?php echo periodically_call_remote(array(
    'frequency' => 60,
    'update'     => 'notificacion',
    'url'        => '@observa',
    'with'       => "'param=' + \${'micontenido'}",
)) ?>
```

Si no se especifica el número de segundos (mediante el parámetro `frequency`) que se esperan después de cada repetición, se tiene en cuenta el valor por defecto que son 10 segundos. El parámetro `with` se evalúa con JavaScript, así que se puede utilizar cualquier función de Prototype, como por ejemplo la función `$F()`.

11.4. Parámetros para la ejecución remota

Todos los *helpers* de Ajax descritos anteriormente pueden utilizar otros parámetros, además de los parámetros `update` y `url`. El array asociativo con los parámetros de Ajax puede modificar el comportamiento de la ejecución remota y del procesamiento de las respuestas.

11.4.1. Actualizar elementos diferentes en función del estado de la respuesta

Si la ejecución remota no devuelve un resultado, los *helpers* pueden actualizar otro elemento distinto al elemento que se actualizaría en caso de una respuesta satisfactoria. Para conseguirlo, solo es necesario indicar como valor del parámetro `update` un array asociativo que establezca los diferentes elementos que se actualizan en caso de respuesta correcta (`success`) y respuesta incorrecta (`failure`). Se trata de una técnica eficaz cuando una página contiene muchas interacciones de Ajax y una única zona de notificación de errores. El listado 11-16 muestra el uso de esta técnica.

Listado 11-16 - Actualización condicional en función de la respuesta

```
<div id="error"></div>
<div id="respuesta"></div>
<p>ⓂHola Mundo!</p>
```

```
<?php echo link_to_remote('Borrar este artículo', array(
    'update' => array('success' => 'respuesta', 'failure' => 'error'),
    'url'     => 'articulo/borrar?id='.$articulo->getId(),
)) ?>
```

Sugerencia Solo las respuestas de servidor cuyo código de estado HTTP sea de tipo error (500, 404 y todos los códigos diferentes de 2XX) provocan la actualización del elemento preparado para las respuestas erróneas. Las acciones que devuelven el valor `sfView::ERROR` no se consideran como erróneas. De esta forma, si se requiere que una acción de tipo Ajax devuelva una respuesta errónea, se debe ejecutar `$this->getResponse()->setStatusCode(404)` con cualquier código HTTP de error.

11.4.2. Actualizar un elemento según su posición

Al igual que sucede con el *helper* `update_element_function()`, se puede especificar el elemento a actualizar de forma relativa respecto de otro elemento mediante el parámetro `position`. El listado 11-17 muestra un ejemplo.

Listado 11-17 - Uso del parámetro `position` para modificar el lugar donde se muestra la respuesta

```
<div id="respuesta"></div>
<p>ⓂHola Mundo!</p>
<?php echo link_to_remote('Borrar este artículo', array(
    'update' => 'respuesta',
    'url'     => 'articulo/borrar?id='.$articulo->getId(),
    'position' => 'after',
)) ?>
```

En esta ocasión, la respuesta de la petición Ajax se muestra después (*after*) del elemento cuyo atributo `id` es igual a `respuesta`; es decir, se muestra después del `<div>` y antes del `<p>`. De esta forma, se pueden realizar varias peticiones Ajax y ver como se acumulan todas las respuestas después del elemento que se actualiza.

El parámetro `position` puede tomar uno de los siguientes valores:

- `before`: antes del elemento
- `after`: después del elemento
- `top`: antes que cualquier otro contenido del elemento
- `bottom`: después de todos los contenidos del elemento

11.4.3. Actualizar un elemento en función de una condición

Las peticiones Ajax pueden tomar un parámetro adicional que permite que el usuario de su consentimiento antes de ejecutar la petición con el objeto `XMLHttpRequest`, como muestra el listado 11-18.

Listado 11-18 - Uso del parámetro `confirm` para solicitar el consentimiento del usuario antes de realizar la petición remota

```
<div id="respuesta"></div>
<?php echo link_to_remote('Borrar este artículo', array(
    'update'    => 'respuesta',
    'url'       => 'articulo/borrar?id='.$articulo->getId(),
    'confirm'   => '¿Estás seguro?',
)) ?>
```

En este caso, se muestra al usuario un cuadro de diálogo de JavaScript con el mensaje "¿Estás seguro?" cuando pincha sobre el enlace. La acción `articulo/borrar` solo se ejecuta si el usuario da su consentimiento a esta petición pulsando sobre el botón de "Aceptar".

La ejecución de la petición remota también se puede condicionar a que se cumpla una condición JavaScript evaluada en el navegador del usuario, mediante el parámetro `condition`, tal y como se muestra en el listado 11-19.

Listado 11-19 - Ejecución de petición remota condicionada a que se cumpla una condición probada en el lado del cliente

```
<div id="respuesta"></div>
<?php echo link_to_remote('Borrar este artículo', array(
    'update'    => 'respuesta',
    'url'       => 'articulo/borrar?id='.$articulo->getId(),
    'condition' => "$('IDelemento') == true",
)) ?>
```

11.4.4. Determinando el método de una petición Ajax

Las peticiones Ajax se realizan por defecto mediante un método POST. Si se quiere realizar una petición Ajax que no modifica los datos o si se quiere mostrar un formulario que incluye validación como resultado de una petición Ajax, se puede utilizar el método GET. La opción `method` modifica el método de la petición Ajax, como muestra el listado 11-20.

Listado 11-20 - Modificando el método de una petición Ajax

```
<div id="respuesta"></div>
<?php echo link_to_remote('Borrar este artículo', array(
    'update'    => 'respuesta',
    'url'       => 'articulo/borrar?id='.$articulo->getId(),
    'method'    => 'get',
)) ?>
```

11.4.5. Permitiendo la ejecución de un script

Si la respuesta de una petición Ajax incluye código JavaScript (el código es la respuesta del servidor y se incluye en el elemento indicado por el parámetro `update`) por defecto no se ejecuta ese código. El motivo es el de reducir la posibilidad de ataques remotos y para permitir al programador autorizar la ejecución del código de la respuesta después de comprobar el contenido del código.

Para permitir la ejecución de los scripts de la respuesta del servidor, se debe utilizar la opción `script`. El listado 11-21 muestra un ejemplo de una petición Ajax remota que autoriza la ejecución del código JavaScript que forme parte de la respuesta.

Listado 11-21 - Permitiendo la ejecución de un script en una respuesta Ajax

```
<div id="respuesta"></div>
<?php
    // Si la respuesta de la acción articulo/borrar contiene código
    // JavaScript, se ejecuta en el navegador del usuario
    echo link_to_remote('Borrar este artículo', array(
        'update'    => 'respuesta',
        'url'       => 'articulo/borrar?id='.$articulo->getId(),
        'script'    => 'true',
    )) ?>
```

Si la plantilla remota contiene *helpers* de Ajax (como por ejemplo `remote_function()`), estas funciones PHP generan código JavaScript, que no se ejecuta a menos que se indique la opción `script => true`.

Nota Cuando se permite la ejecución de los scripts de la respuesta remota, el código fuente del código remoto no se puede ver ni siquiera con una herramienta para visualizar el código generado. Los scripts se ejecutan pero su código no se muestra. Se trata de un comportamiento poco habitual, pero completamente normal.

11.4.6. Creando callbacks

Una desventaja importante de las interacciones creadas con Ajax es que son invisibles al usuario hasta que se actualiza la zona preparada para las notificaciones. Por tanto, si se produce un error de servidor o la red está congestionada, los usuarios pueden pensar que su acción se ha realizado correctamente cuando en realidad aun no ha sido procesada. Este es el motivo por el que es muy importante notificar al usuario sobre los eventos que se producen a lo largo de una interacción creada con Ajax.

Por defecto, cada petición remota es un proceso asíncrono durante el que se pueden ejecutar varias funciones JavaScript de tipo callback (por ejemplo para indicar el progreso de la petición). Todas las funciones de callback tienen acceso directo al objeto `request`, que contiene a su vez el objeto `XMLHttpRequest`. Los callback que se pueden definir se corresponden con los eventos que se producen durante una interacción de Ajax:

- **before:** antes de que se inicie la petición
- **after:** justo después de que se inicie la petición y antes de que se cargue
- **loading:** cuando se está cargando la respuesta remota en el navegador
- **loaded:** cuando el navegador ha terminado de cargar la respuesta remota
- **interactive:** cuando el usuario puede interaccionar con la respuesta remota, incluso si no se ha terminado de cargar
- **success:** cuando `XMLHttpRequest` está completo y el código HTTP de estado corresponde al rango 2XX
- **failure:** cuando `XMLHttpRequest` está completo y el código HTTP de estado no corresponde al rango 2XX

- 404: cuando la petición devuelve un error de tipo 404
- complete: cuando XMLHttpRequest está completo (se ejecuta después de success o failure, si alguno de los 2 está definido)

El ejemplo más habitual es el de mostrar un indicador de tipo Cargando... mientras la petición remota se está ejecutando y ocultarlo cuando se recibe la respuesta. Para incluir este comportamiento, solo es necesario añadir los parámetros loading y complete a la petición Ajax, tal y como muestra el listado 11-22.

Listado 11-22 - Uso de callbacks en Ajax para mostrar y ocultar un indicador de actividad

```
<div id="respuesta"></div>
<div id="indicador">Cargando...</div>
<?php echo link_to_remote('Borrar este artículo', array(
    'update'    => 'respuesta',
    'url'       => 'articulo/borrar?id='.$articulo->getId(),
    'loading'   => "Element.show('indicador')",
    'complete'  => "Element.hide('indicador')",
)) ?>
```

Los métodos show(), hide() y el objeto Element son otras de las utilidades proporcionadas por la librería Prototype.

11.5. Creando efectos visuales

Symfony integra los efectos visuales de la librería script.aculo.us, para poder incluir efectos más avanzados que simplemente mostrar y ocultar elementos <div> en las páginas. La mejor documentación sobre la sintaxis que se puede emplear en los efectos se encuentra en el wiki de la librería en <http://script.aculo.us/>. Básicamente, la librería se encarga de proporcionar objetos y funciones JavaScript que manipulan el DOM de la página para conseguir efectos visuales complejos. El listado 11-23 incluye algunos ejemplos. Como el resultado es una animación o efecto visual de ciertas partes de la página, es recomendable que pruebes los efectos para entender bien en qué consiste cada efecto. El sitio web de script.aculo.us incluye una galería donde se pueden ver sus efectos visuales.

Listado 11-23 - Efectos visuales en JavaScript con Script.aculo.us

```
// Resalta el elemento 'mi_elemento'
Effect.Highlight('mi_elemento', { startcolor:'#ff99ff', endcolor:'#999999' })

// Oculta un elemento
Effect.BlindDown('mi_elemento');

// Hace desaparecer un elemento
Effect.Fade('mi_elemento', { transition: Effect.Transitions.wobble })
```

Symfony encapsula el objeto Effect de JavaScript en un *helper* llamado visual_effect(), que forma parte del *helper* Javascript. El código generado es JavaScript y puede utilizarse en un enlace normal, como muestra el listado 11-24.

Listado 11-24 - Efectos visuales en las plantillas con el *helper* visual_effect()


```

<div id="div_oculto" style="display:none">Aquí estaba!</div>
<?php echo link_to_function(
    'Mostrar el DIV oculto',
    visual_effect('appear', 'div_oculto')
) ?>
// Equivalente a llamar a Effect.Appear('div_oculto')

```

El *helper* `visual_effects()` también se puede utilizar en los callbacks de Ajax, como en el listado 11-22, que muestra un indicador de actividad de forma más elegante que en el listado 11-22. El elemento indicador aparece de forma progresiva cuando comienza la petición Ajax y se desaparece también progresivamente cuando se recibe la respuesta del servidor. Además, el elemento respuesta se resalta después de ser actualizado por la petición remota, de forma que esa parte de la página capte la atención del usuario.

Listado 11-25 - Efectos visuales en los callbacks de Ajax

```

<div id="respuesta"></div>
<div id="indicador" style="display: none">Cargando...</div>
<?php echo link_to_remote('Borrar este artículo', array(
    'update' => 'respuesta',
    'url' => 'articulo/borrar?id='.$articulo->getId(),
    'loading' => visual_effect('appear', 'indicador'),
    'complete' => visual_effect('fade', 'indicador').
                    visual_effect('highlight', 'feedback'),
)) ?>

```

Los efectos visuales se pueden combinar de forma muy sencilla concatenando sus llamadas (mediante el `.`) dentro de un callback.

11.6. JSON

JSON (JavaScript Object Notation) es un formato sencillo para intercambiar datos. Consiste básicamente en un array asociativo de JavaScript (ver ejemplo en el listado 11-26) que se utilizar para incluir información del objeto. JSON ofrece 2 grandes ventajas para las interacciones Ajax: es muy fácil de leer en JavaScript y puede reducir el tamaño en bytes de la respuesta del servidor.

Listado 11-26 - Ejemplo de objeto JSON en JavaScript

```

var misDatosJson = {"menu": {
    "id": "archivo",
    "valor": "Archivo",
    "popup": {
        "menuitem": [
            {"value": "Nuevo", "onclick": "CrearNuevoDocumento()"},
            {"value": "Abrir", "onclick": "AbrirDocumento()"},
            {"value": "Cerrar", "onclick": "CerrarDocumento()"}
        ]
    }
}}

```

El formato JSON es el más adecuado para la respuesta del servidor cuando la acción Ajax debe devolver una estructura de datos a la página que realizó la llamada de forma que se pueda

procesar con JavaScript. Este mecanismo es útil por ejemplo cuando una sola petición Ajax debe actualizar varios elementos en la página.

En el listado 11-27 se muestra un ejemplo de página que contiene 2 elementos que deben ser actualizados. Un *helper* remoto solo puede actualizar uno de los elementos de la página (o título o nombre) pero no los 2 a la vez.

Listado 11-27 - Plantilla de ejemplo para actualizaciones Ajax múltiples

```
<h1 id="titulo">Carta normal</h1>
<p>Estimado <span id="nombre">el_nombre</span>,</p>
<p>Hemos recibido su email y le contestaremos en el menor plazo de tiempo.</p>
<p>Reciba un saludo cordial,</p>
```

Para actualizar los dos elementos, la respuesta Ajax podría estar formada por el siguiente array en formato JSON:

```
|  [{"titulo", "Mi carta normal"}, {"nombre", "Sr. Pérez"}]
```

Mediante algunas pocas instrucciones de JavaScript se puede interpretar la respuesta del servidor y actualizar varios elementos de la página de forma seguida. El listado 11-28 muestra el código que se podría añadir a la plantilla del listado 11-27 para conseguir este efecto.

Listado 11-28 - Actualizando más de un elemento mediante una respuesta remota

```
<?php echo link_to_remote('Actualizar la carta', array(
    'url'      => 'publicaciones/actualizar',
    'complete' => 'actualizaJSON.ajax')
)) ?>

<?php echo javascript_tag("
function actualizaJSON.ajax()
{
    json = ajax.responseJSON;
    for (var i = 0; i < json.length; i++)
    {
        Element.update(json[i][0], json[i][1]);
    }
}
") ?>
```

Dentro de la opción *complete* se tiene acceso directo a la respuesta ajax y por tanto se puede enviar el objeto de la respuesta del servidor a una función externa. La función *actualizaJSON()* recorre los datos JSON obtenidos mediante la propiedad *responseJSON* y para cada elemento del array actualiza el elemento cuyo atributo *id* coincide con el primer parámetro del array y muestra el contenido incluido en el segundo parámetro del array.

El listado 11-29 muestra como devuelve la acción *publicaciones/actualizar* una respuesta de tipo JSON.

Listado 11-29 - Ejemplo de acción que devuelve datos JSON

```
class publicacionesActions extends sfActions
{
    public function executeActualizar()
```

```
{
    $this->getResponse()->setHttpHeader('Content-Type', 'application/json;
charset=utf-8');
    $resultado = '["titulo", "Mi carta normal"], ["nombre", "Sr. Pérez"]';
    return $this->renderText('('.$resultado.')');
}
```

Si se utiliza la cabecera que establece el tipo de contenido a `application/json`, las librerías como Prototype pueden evaluar automáticamente el código JSON de la respuesta. Además, este método es preferible a enviar los datos JSON en la propia cabecera HTTP, ya que estas cabeceras están limitadas en tamaño y algunos navegadores pueden sufrir problemas con las respuestas de servidor que sólo tienen cabeceras y ningún contenido. Por último, el método `->renderText()` hace que no se utilice ninguna plantilla, por lo que mejora el rendimiento de la aplicación.

JSON se ha convertido en un estándar en el desarrollo de aplicaciones web. Los servicios web proponen la utilización de JSON en vez de XML para permitir la integración de servicios en el navegador del usuario en vez de en el servidor. El formato JSON es seguramente la mejor opción para el intercambio de información entre el servidor y las funciones JavaScript.

Sugerencia Desde la versión 5.2 de PHP existen dos funciones, `json_encode()` y `json_decode()`, que permiten convertir un array PHP en un array JSON y viceversa (<http://www.php.net/manual/es/ref.json.php>). Estas funciones facilitan la integración de los arrays JSON y de Ajax en general.

Utilizando estas funciones, el código del ejemplo anterior se puede rehacer de la siguiente manera:

```
$resultado = array("titulo" => "Mi carta normal", "nombre" => "Sr. Pérez");
return $this->renderText(json_encode($resultado));
```

11.7. Interacciones complejas con Ajax

Entre los *helpers* de Ajax de Symfony, también existen utilidades que permiten construir interacciones complejas con una sola llamada a una función. Estas utilidades permiten mejorar la experiencia de usuario añadiendo características propias de las aplicaciones de escritorio (arrastrar y soltar, autocompletar, edición directa de contenidos, etc.) sin necesidad de escribir código JavaScript. En las siguientes secciones se describen los *helpers* de las interacciones complejas mediante ejemplos sencillos. Los parámetros adicionales y otras configuraciones se pueden consultar en la documentación de script.aculo.us.

Cuidado Aunque es sencillo incluir interacciones complejas, lo más complicado es configurarlas de forma que el usuario las perciba como algo natural en la página. Por tanto, solo se deben utilizar cuando se está seguro de que va a mejorar la experiencia de usuario. No deberían incluirse cuando su efecto es el de confundir al usuario.

11.7.1. Autocompletar

La interacción denominada "autocompletar" consiste en un cuadro de texto que muestra una lista de valores relacionados con los caracteres que teclea el usuario. Este efecto se puede conseguir con una única llamada al *helper* `input_auto_complete_tag()`, siempre que la acción

remota devuelva una respuesta formateada como una lista de elementos HTML (y) similar a la mostrada en el ejemplo 11-30.

Listado 11-30 - Ejemplo de respuesta compatible con el *helper* de autocompletar

```
<ul>
  <li>sugerencia 1</li>
  <li>sugerencia 2</li>
  ...
</ul>
```

El *helper* se puede incluir en cualquier plantilla de la misma forma que se incluiría cualquier cuadro de texto, como se muestra en el ejemplo 11-31.

Listado 11-31 - Uso del *helper* de autocompletar en una plantilla

```
<?php echo form_tag('mimodulo/miaccion') ?>
  Buscar un autor en función de su nombre:
  <?php echo input_auto_complete_tag('autor', 'nombre por defecto',
    'autor/autocompletar',
    array('autocomplete' => 'off'),
    array('use_style' => true)
  ) ?>
  <?php echo submit_tag('Buscar') ?>
</form>
```

Cada vez que el usuario teclee un carácter en el cuadro de texto autor, se realiza una llamada a la acción remota autor/autocompletar. El código de esa acción depende de cada caso y aplicación, pero debe obtener una lista de valores sugeridos en forma de lista de elementos HTML como la mostrada en el listado 11-30. El *helper* muestra la lista devuelta debajo del cuadro de texto autor y si el usuario pincha sobre un valor o lo selecciona mediante el teclado, el cuadro de texto se completa con ese valor, tal y como muestra la figura 11-3.

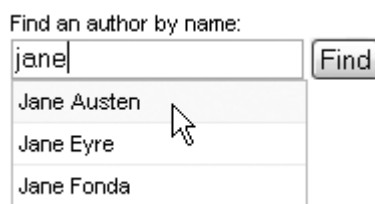


Figura 11.3. Ejemplo de autocompletar

El tercer argumento del *helper* `input_auto_complete_tag()` puede tomar uno de los siguientes parámetros:

- `use_style`: aplica estilos CSS de forma automática a la lista que se muestra.
- `frequency`: frecuencia con la que se realizan peticiones remotas (por defecto son 0.4 segundos).
- `indicator`: el valor del atributo `id` de un elemento que se muestra cuando comienza la carga de las sugerencias y se oculta cuando se ha completado.

- **tokens:** permite autocompletar por partes. Si el valor de este parámetro es , y el usuario introduce pedro, juan a la acción solo se le pasa el valor juan (siempre se le pasa el último valor después de trocear el cuadro de texto según el carácter definido por tokens).

11.7.2. Arrastrar y soltar

En las aplicaciones de escritorio suele ser normal coger un elemento con el ratón, moverlo y soltarlo en otro lugar. Sin embargo, en las aplicaciones web es mucho más raro de ver esta técnica, ya que es bastante difícil de programarla a mano con JavaScript. Afortunadamente, en Symfony se puede incluir esta técnica solo con una línea de código.

El framework incluye 2 *helpers*, `draggable_element()` y `drop_receiving_element()`, que se encargan de modificar el comportamiento de los elementos; estos *helpers* "observan" a los elementos y les añaden nuevas habilidades. Se utilizan para declarar a los elementos como "arrastrable" o como "elemento en el que se pueden soltar los elementos arrastrables". Un elemento arrastrable se activa cuando se pulsa con el ratón sobre él. Mientras no se suelte el ratón, el elemento se mueve siguiendo la posición del ratón. Los elementos en los que se pueden soltar los elementos arrastrables llaman a una función remota cuando el elemento arrastrable se suelta sobre esa zona. El listado 11-32 muestra un ejemplo de esta interacción mediante un elemento que hace de carrito de la compra.

Listado 11-32 - Elementos de arrastrar y soltar en un carrito de la compra

```
<ul id="elementos">
  <li id="elemento1" class="comida">Zanahoria</li>
  <?php echo draggable_element('elemento1', array('revert' => true)) ?>
  <li id="elemento2" class="comida">Manzana</li>
  <?php echo draggable_element('elemento2', array('revert' => true)) ?>
  <li id="elemento3" class="comida">Naranja</li>
  <?php echo draggable_element('elemento3', array('revert' => true)) ?>
</ul>
<div id="carrito">
  <p>El carrito está vacío</p>
  <p>Arrastra y suelta elementos aquí para añadirlos al carrito</p>
</div>
<?php echo drop_receiving_element('carrito', array(
  'url'      => 'carrito/anadir',
  'accept'   => 'comida',
  'update'   => 'carrito',
)) ?>
```

Cada uno de los elementos de la lista se pueden coger con el ratón y moverlos por la ventana del navegador. Cuando se suelta el ratón, el elemento vuelve a su posición original. Si el elemento se suelta sobre el elemento cuyo atributo `id` es `carrito`, se realiza una llamada a la acción remota `carrito/anadir`. La acción puede determinar el elemento que se ha añadido mediante el parámetro de petición `id`. De esta forma, el listado 11-32 es una aproximación muy realista al proceso físico de compra de productos: se cogen los productos, se sueltan en el carrito y después se realiza el pago.

Sugerencia En el listado 11-32, los *helpers* aparecen justo después del elemento que modifican, aunque no es obligatorio. Si se quiere, se pueden agrupar todos los *helpers* `draggable_element()` y `drop_receiving_element()` al final de la plantilla. Lo único importante es el primer argumento que se pasa al *helper* y que indica el elemento al que se aplica.

El *helper* `draggable_element()` acepta los siguientes parámetros:

- **revert**: si vale `true`, el elemento vuelve a su posición original cuando se suelta el ratón. También se puede indicar el nombre de una función que se ejecuta cuando finaliza el arrastre del elemento.
- **ghosting**: realiza una copia del elemento original y el usuario mueve la copia, quedando inmóvil el elemento original.
- **snap**: si vale `false`, el movimiento del elemento es libre. En otro caso, el elemento solo se puede desplazar de forma escalonada como si estuviera una gran rejilla a la que se ajusta el elemento. El valor del desplazamiento horizontal (x) y vertical (y) del elemento se puede definir como `xy`, `[x,y]` o `function(x,y){ return [x,y] }`.

El *helper* `drop_receiving_element()` acepta los siguientes parámetros:

- **accept**: una cadena de texto o un array de cadenas de texto que representan a valores de clases CSS. Este elemento solo permitirá que se suelten sobre el los elementos cuyas clases CSS contengan al menos uno de los valores indicado.
- **hoverclass**: clase CSS que se añade al elemento cuando el usuario arrastra (sin soltarlo) un elemento sobre esta zona.

11.7.3. Listas ordenables

Otra posibilidad que brindan los elementos arrastrables es la de ordenar una lista moviendo sus elementos con el ratón. El *helper* `sortable_element()` añade este comportamiento a los elementos de la lista, como se muestra en el ejemplo del listado 11-33.

Listado 11-33 - Ejemplo de lista ordenable

```
<p>What do you like most?</p>
<ul id="ordenar">
  <li id="elemento_1" class="ordenable">Zanahorias</li>
  <li id="elemento_2" class="ordenable">Manzanas</li>
  <li id="elemento_3" class="ordenable">Naranjas</li>
  // A nadie le gustan las coles de Bruselas
  <li id="elemento_4">Coles de Bruselas</li>
</ul>
<div id="respuesta"></div>
<?php echo sortable_element('ordenar', array(
  'url'      => 'elemento/ordenar',
  'update'   => 'respuesta',
  'only'     => 'ordenable',
)) ?>
```

Gracias a la magia del *helper* `sortable_element()`, la lista `` se transforma en una lista ordenable dinámicamente, de forma que sus elementos se pueden reordenar mediante la técnica de arrastras y soltar. Cada vez que el usuario mueve un elemento y lo suelta para reordenar la lista, se realiza una petición Ajax con los siguientes parámetros:

```
POST /sf_sandbox/web/frontend_dev.php/elemento/ordenar HTTP/1.1
ordenar[]=1&ordenar[]=3&ordenar[]=2&_ =
```

La lista completa se pasa como un array con el formato `ordenar[$rank]=$id`, el `$rank` empieza en 0 y el `$id` es el valor que se indica después del guión bajo (`_`) en el valor del atributo `id` de cada elemento de la lista. El atributo `id` de la lista completa (ordenar en este caso) se utiliza para el nombre del array de parámetros que se pasan al servidor.

El *helper* `sortable_element()` acepta los siguientes parámetros:

- **only:** una cadena de texto o un array de cadenas de texto que representan a valores de clases CSS. Solamente se podrán mover los elementos de la lista que tengan este valor en su atributo `class`.
- **hoverclass:** clase CSS que se añade a la lista cuando el usuario posiciona el puntero del ratón encima de ella.
- **overlap:** su valor debería ser `horizontal` si los elementos de la lista se muestran de forma horizontal y su valor debería ser `vertical` (que es el valor por defecto) cuando los elementos se muestran cada uno en una línea (como se muestran por defecto las listas en HTML).
- **tag:** si la lista reordenable no contiene elemento ``, se debe indicar la etiqueta que define los elementos que se van a hacer reordenables (por ejemplo `div` o `dl`).

Sugerencia A partir de Symfony 1.1 también se puede utilizar el *helper* `sortable_element()` sin la opción `url`. De esta forma, no se realiza ninguna petición AJAX después de cada reordenación. El uso más común es el de realizar todas las peticiones AJAX cuando el usuario pulsa sobre el botón de Guardar o similar.

11.7.4. Edición directa de contenidos

Cada vez más aplicaciones web permiten editar los contenidos de sus páginas sin necesidad de utilizar formularios que incluyen el contenido de la página. El funcionamiento de esta interacción es muy sencillo. Cuando el usuario pasa el ratón por encima de un bloque de texto, este se resalta. Si el usuario pincha sobre el bloque, el texto se convierte en un control de formulario llamado área de texto (`textarea`) que muestra el texto original. Además, se muestra un botón para guardar los cambios. El usuario realiza los cambios en el texto original y pulsa sobre el botón de guardar los cambios. Una vez guardado, el área de texto desaparece y el texto modificado se vuelve a mostrar de forma normal. Con Symfony, toda esta interacción se puede realizar aplicando el *helper* `input_in_place_editor_tag()` al elemento. El listado 11-34 muestra el uso de este *helper*.

Listado 11-34 - Ejemplo de texto editable

```
<div id="modificame">Puedes modificar este texto</div>
<?php echo input_in_place_editor_tag('modificame', 'mimodulo/miaccion', array(
    'cols'      => 40,
    'rows'      => 10,
)) ?>
```

Cuando el usuario pincha sobre el texto editable, se reemplaza por un cuadro de texto que contiene el texto original y que se puede modificar. Al guardar los cambios, se llama mediante Ajax a la acción `mimodulo/miaccion` con el contenido modificado como valor del parámetro `value`. El resultado de la acción actualiza el elemento editable. Se trata de una interacción muy rápida de incluir y muy poderosa.

El *helper* `input_in_place_editor_tag()` acepta los siguientes parámetros:

- `cols` y `rows`: el tamaño (en filas y columnas) del área de texto que se muestra para editar el contenido original (si el valor de `rows` es mayor que 1, se muestra un `<textarea>`; en otro caso, se muestra un `<input type="text">`).
- `loadTextURL`: la URI de la acción que se llama para obtener el texto que se debe editar. Se trata de una opción útil cuando el contenido del elemento tiene un formato especial y se quiere que el usuario edite el texto sin ese formato aplicado.
- `save_text` y `cancel_text`: el texto del enlace para guardar los cambios (el valor por defecto es "ok") y el del enlace para cancelar los cambios (el valor por defecto es "cancel").

Existen muchas otras opciones que se pueden consultar en la documentación de script.aculo.us sobre la edición directa de contenidos (<http://github.com/madrobby/scriptaculous/wikis/ajax-inplaceeditor>).

11.8. Resumen

Si estás cansado de escribir código JavaScript en las plantillas para incluir efectos en el navegador del usuario, los *helpers* de JavaScript de Symfony son una alternativa más sencilla. No solo automatizan los enlaces JavaScript tradicionales y la actualización de los elementos, sino que también permiten incluir interacciones Ajax de forma muy sencilla. Gracias a las mejoras que Prototype proporciona a la sintaxis de JavaScript y gracias a los efectos visuales de la librería `script.aculo.us`, hasta las interacciones más complejas se pueden realizar con unas pocas líneas de código.

Y como en Symfony es igual de fácil hacer una página estática que una página completamente interactiva y dinámica, las aplicaciones web pueden incluir todas las interacciones tradicionales de las aplicaciones de escritorio.

Capítulo 12. Uso de la cache

Una de las técnicas disponibles para mejorar el rendimiento de una aplicación consiste en almacenar trozos de código HTML o incluso páginas enteras para poder servirlos en futuras peticiones. Esta técnica se denomina "utilizar caches" y se pueden definir tanto en el lado del servidor como en el del cliente.

Symfony incluye un sistema de cache en el servidor muy flexible. Con este sistema es muy sencillo guardar en un archivo una página entera, el resultado de una acción, un elemento parcial o un trozo de plantilla. La configuración del sistema de cache se realiza de forma intuitiva mediante archivos de tipo YAML. Cuando los datos se modifican, se pueden borrar partes de la cache de forma selectiva mediante la línea de comandos o mediante algunos métodos especiales en las acciones. Symfony también permite controlar la cache en el lado del cliente mediante las cabeceras de HTTP 1.1. En este capítulo se presentan todas estas técnicas y se dan pistas para determinar las mejoras que las caches confieren a las aplicaciones.

12.1. Guardando la respuesta en la cache

El principio básico de las caches de HTML es muy sencillo: parte o todo el código HTML que se envía al usuario como respuesta a su petición se puede reutilizar en peticiones similares. El código HTML se almacena en un directorio especial (el directorio `cache/`) donde el controlador frontal lo busca antes de ejecutar la acción. Si se encuentra el código en la cache, se envía sin ejecutar la acción, por lo que se consigue un gran ahorro de tiempo de ejecución. Si no se encuentra el código, se ejecuta la acción y su respuesta (la vista) se guarda en el directorio de la cache para las futuras peticiones.

Como todas las páginas pueden contener información dinámica, la cache HTML está deshabilitada por defecto. El administrador del sitio web debe activarla para mejorar el rendimiento de la aplicación.

Symfony permite gestionar tres tipos diferentes de cache HTML:

- Cache de una acción (con o sin layout)
- Cache de un elemento parcial, de un componente o de un slot de componentes
- Cache de un trozo de plantilla

Los dos primeros tipos de cache se controlan mediante archivos YAML de configuración. La cache de trozos de plantillas se controla mediante llamadas a *helpers* dentro de las propias plantillas.

12.1.1. Opciones de la cache global

La cache HTML se puede habilitar y deshabilitar (su valor por defecto) para cada aplicación de un proyecto y para cada entorno mediante la opción `cache` del archivo `settings.yml`. El listado 12-1 muestra como habilitar la cache.

Listado 12-1 - Activando la cache, en frontend/config/settings.yml

```
dev:
  .settings:
    cache: on
```

12.1.2. Guardando una acción en la cache

Las acciones que muestran información estática (que no depende de bases de datos ni de información guardada en la sesión) y las acciones que leen información de una base de datos pero no la modifican (acciones típicas del método GET) son el tipo de acción ideal para almacenar su resultado en la cache. La figura 12-1 muestra los elementos de la página que se guardan en la cache en este caso: o el resultado de la acción (su plantilla) o el resultado de la acción junto con el layout.

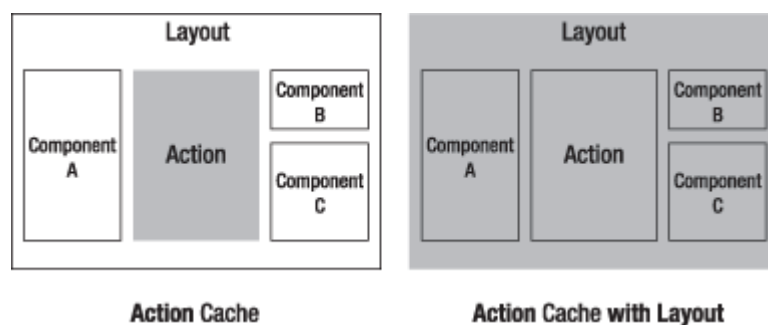


Figura 12.1. Guardando una acción en la cache

Si se dispone por ejemplo de una acción usuario/listado que devuelve un listado de todos los usuarios de un sitio web, a no ser que se modifique, añada o elimine un usuario (que se verá más adelante en la sección "Eliminar elementos de la cache") la lista contiene siempre la misma información, por lo que esta acción es ideal para guardarla en la cache.

La activación de la cache y las opciones para cada acción se definen en el archivo cache.yml del directorio config/ del módulo. El listado 12-2 muestra un ejemplo de este archivo.

Listado 12-2 - Activando la cache de una acción, en frontend/modules/usuario/config/cache.yml

```
listado:
  enabled: on
  with_layout: false # Valor por defecto
  lifetime: 86400 # Valor por defecto
```

La anterior configuración activa la cache para la acción listado y el layout no se guarda junto con el resultado de la acción (que además, es el comportamiento por defecto). Por tanto, aunque exista en la cache el resultado de la acción, el layout completo (junto con sus elementos parciales y componentes) se sigue ejecutando. Si la opción with_layout vale true, en la cache se guarda el resultado de la acción junto con el layout, por lo que este último no se vuelve a ejecutar.

Para probar las opciones de la cache, se accede con el navegador a la acción en el entorno de desarrollo.

```
| http://miaplicacion.ejemplo.com/frontend_dev.php/usuario/listado
```

Ahora se puede apreciar un borde que encierra la zona del área en la página. La primera vez, el área tiene una cabecera azul, lo que indica que no se ha obtenido de la cache. Si se recarga la página, el área de la acción muestra una cabecera amarilla, indicando que esta vez sí se ha obtenido directamente de la cache (resultando en una gran reducción en el tiempo de respuesta de la acción). Más adelante en este capítulo se detallan las formas de probar y monitorizar el funcionamiento de la cache.

Nota Los slots son parte de la plantilla, por lo que si se guarda el resultado de una acción en la cache, también se guarda el valor de los slots definidos en la plantilla de la acción. De esta forma, la cache funciona de forma nativa para los slots.

El sistema de cache también funciona para las páginas que utilizan parámetros. El módulo usuario anterior podría disponer de una acción llamada `ver` y a la que se pasa como parámetro una variable llamada `id` para poder mostrar los detalles de un usuario. El listado 12-3 muestra como modificar los cambios necesarios en el archivo `cache.yml` para habilitar la cache también en esta acción.

Se puede organizar de forma más clara el archivo `cache.yml` reagrupando las opciones comunes a todas las acciones del módulo bajo la clave `all:`, como también muestra el listado 12-3.

Listado 12-3 - Ejemplo de `cache.yml` completo, en `frontend/modules/usuario/config/cache.yml`

```
listado:
  enabled:    on
ver:
  enabled:    on

all:
  with_layout: false    # Valor por defecto
  lifetime:    86400    # Valor por defecto
```

Ahora, cada llamada a la acción `usuario/ver` que tenga un valor del parámetro `id` diferente, crea un nuevo archivo en la cache. De esta forma, la cache para la petición:

```
| http://frontend.ejemplo.com/usuario/ver/id/12
```

es completamente diferente de la cache de la petición:

```
| http://frontend.ejemplo.com/usuario/ver/id/25
```

Cuidado Las acciones que se ejecutan mediante el método `POST` o que tienen parámetros `GET` no se guardan en la cache.

La opción `with_layout` merece una explicación más detallada. Esta opción determina el tipo de información que se guarda en la cache. Si vale `true`, solo se almacenan en la cache el resultado de la ejecución de la plantilla y las variables de la acción. Si la opción vale `false`, se guarda el objeto `response` entero. Por tanto, la cache en la que se guarda el layout (valor `true`) es mucho más rápido que la cache sin el layout.

Si es posible, es decir, si el layout no depende por ejemplo de datos de sesión, es conveniente optar por la opción que guarda el layout en la cache. Desgraciadamente, el layout normalmente

contiene elementos dinámicos (como por ejemplo el nombre del usuario que está conectado), por lo que la opción habitual es la de no almacenar el layout en la cache. No obstante, las páginas que no depende de cookies, los canales RSS, las ventanas emergentes, etc. se pueden guardar en la cache incluyendo su layout.

12.1.3. Guardando un elemento parcial, un componente o un slot de componentes en la cache

En el Capítulo 7 se explicó la forma de reutilizar trozos de código en varias plantillas mediante el *helper* `include_partial()`. Guardar un elemento parcial en la cache es tan sencillo como hacerlo en una acción y se activa de la misma forma, tal y como muestra la figura 12-2.

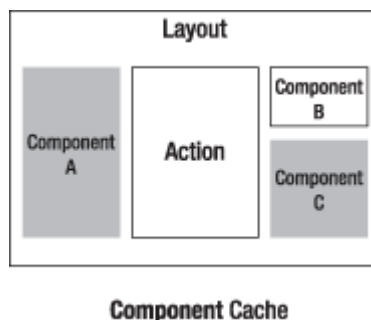


Figura 12.2. Guardando un elemento parcial, un componente o un slot de componentes en la cache

El listado 12-4 por ejemplo muestra los cambios necesarios en el archivo `cache.yml` para activar la cache en el elemento parcial `_mi_parcial.php` que pertenece al módulo usuario. La opción `with_layout` no tiene sentido en este caso.

Listado 12-4 - Guardando un elemento parcial en la cache, en `frontend/modules/usuario/config/cache.yml`

```
_mi_parcial:
  enabled:    on
  listado:
    enabled:  on
  ...
```

Ahora todas las plantillas que incluyen este elemento parcial no ejecutan su código PHP, sino que utilizan la versión almacenada en la cache.

```
| <?php include_partial('usuario/mi_parcial') ?>
```

Al igual que sucede en las acciones, la información que se guarda en la cache depende de los parámetros que se pasan al elemento parcial. El sistema de cache almacena tantas versiones diferentes como valores diferentes de parámetros se pasen al elemento parcial.

```
| <?php include_partial('usuario/mi_otro_parcial', array('parametro' => 'valor')) ?>
```

Sugerencia Guardar la acción en la cache es más avanzado que guardar elementos parciales, ya que cuando una acción se encuentra en la cache, la plantilla ni siquiera se ejecuta; si la plantilla incluye elementos parciales, no se realizan las llamadas a esos elementos parciales. Por tanto, guardar elementos parciales en la cache solo es útil cuando no se está guardando en la cache la acción que se ejecuta o para los elementos parciales incluidos en el layout.

Recordando lo que se explicó en el Capítulo 7: un componente es una pequeña acción que utiliza como vista un elemento parcial y un slot de componentes es un componente para el que la acción varía en función de las acciones que se ejecuten. Estos dos elementos son similares a los elementos parciales, por lo que el funcionamiento de su cache es muy parecido. Si el layout global incluye un componente llamado `dia` mediante `include_component('general/dia')` para mostrar la fecha, el archivo `cache.yml` del módulo `general` debería activar la cache de ese componente de la siguiente forma:

```
_dia:
  enabled: on
```

Cuando se guarda un componente o un elemento parcial en la cache, se debe decidir si se almacena solo una versión para todas las plantillas o una versión para cada plantilla. Por defecto, los componentes se guardan independientemente de la plantilla que lo incluye. No obstante, los componentes contextuales, como por ejemplo los componentes que muestran una zona lateral diferente en cada acción, deben almacenarse tantas veces como el número de plantillas diferentes que los incluyan. El sistema de cache se encarga automáticamente de este último caso, siempre que se establezca el valor `true` a la opción contextual:

```
_dia:
  contextual: true
  enabled: on
```

Nota Los componentes globales (los que se guardan en el directorio `templates/` de la aplicación) también se pueden guardar en la cache, siempre que se configuren sus opciones de cache en el archivo `cache.yml` de la aplicación.

12.1.4. Guardando un fragmento de plantilla en la cache

Guardar en la cache el resultado completo de una acción solamente es posible para algunas acciones. Para el resto de acciones, las que actualizan información y las que muestran en la plantilla información que depende de la sesión, todavía es posible mejorar su rendimiento mediante la cache, pero de forma muy diferente. Symfony incluye un tercer tipo de cache, que se utiliza para los fragmentos de las plantillas y que se activa directamente en la propia plantilla, como se muestra en la figura 12-3.

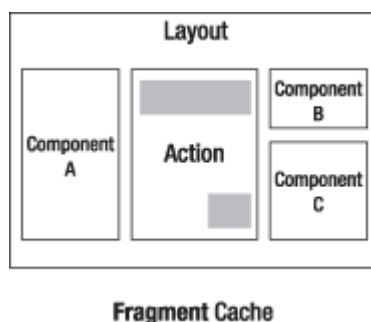


Figura 12.3. Guardando un fragmento de plantilla en la cache

Si por ejemplo se dispone de un listado de usuarios que muestra un enlace al último usuario que se ha accedido, esta última información es dinámica. El `helper` `cache()` define las partes de la plantilla que se pueden guardar en la cache. El listado 12-5 muestra los detalles sobre su sintaxis.

Listado 12-5 - Uso del *helper* `cache()`, en `frontend/modules/usuario/templates/listadoSuccess.php`

```

<!-- Código que se ejecuta cada vez -->
<?php echo link_to('Último usuario accedido', 'usuario/
ver?id='.$id_ultimo_usuario_accedido) ?>

<!-- Código guardado en la cache -->
<?php if (!cache('usuarios')): ?>
    <?php foreach ($usuarios as $usuario): ?>
        <?php echo $usuario->getNombre() ?>
    <?php endforeach; ?>
    <?php cache_save() ?>
<?php endif; ?>

```

Así es como funciona esta cache:

- Si se encuentra en la cache una versión del fragmento llamado 'usuarios', se utiliza para reemplazar todo el código existente entre `<?php if (!cache('usuarios')): ?>` y `<?php endif; ?>`.
- Si no se encuentra, se ejecuta el código definido entre esas 2 líneas y el resultado se guarda en la cache identificado con el nombre indicando en la llamada al *helper* `cache()`.

Todo el código que no se incluye entre esas dos líneas, se ejecuta siempre y por tanto nunca se guarda en la cache.

Cuidado La acción (listado en este ejemplo) no puede tener activada la cache, ya que en ese caso, no se ejecutaría la plantilla y se ignoraría por completo la declaración de la cache de los fragmentos.

La mejora en la velocidad de la aplicación cuando se utiliza esta cache no es tan significativa como cuando se guarda en la cache la acción entera, ya que en este caso siempre se ejecuta la acción, la plantilla se procesa al menos de forma parcial y siempre se utiliza el layout para decorar la plantilla.

Se pueden guardar otros fragmentos de la misma plantilla en la cache; sin embargo, en este caso se debe indicar un nombre único a cada fragmento, de forma que el sistema de cache de Symfony pueda encontrarlos cuando sea necesario.

Como sucede con las acciones y los componentes, los fragmentos que se guardan en la cache pueden tener definido un tiempo de vida en segundos como segundo argumento de la llamada al *helper* `cache()`.

```

| <?php if (!cache('usuarios', 43200)): ?>

```

Si no se indica explícitamente en el *helper*, se utiliza el valor por defecto para el tiempo de vida de la cache (que son 86400 segundos, equivalentes a 1 día).

Sugerencia Otra forma de hacer que una acción se pueda guardar en la cache es pasar las variables que modifican su comportamiento en el patrón del sistema de enrutamiento de la acción. Si la página principal muestra el nombre del usuario que está conectado, no se puede cachear la página a menos que la URL contenga el nombre del usuario. Otro caso es el de las

aplicaciones multi-idioma: si se quiere activar la cache para una página que tiene varias traducciones, el código del idioma debería incluirse dentro del patrón de la URL. Aunque este truco aumenta el número de páginas que se guardan en la cache, puede ser muy útil para acelerar las aplicaciones que son muy interactivas.

12.1.5. Configuración dinámica de la cache

El archivo `cache.yml` es uno de los métodos disponibles para definir las opciones de la cache, pero tiene el inconveniente de que no se puede modificar de forma dinámica. No obstante, como sucede habitualmente en Symfony, se puede utilizar código PHP en vez de archivos YAML, por lo que se puede configurar de forma dinámica la cache.

¿Para qué puede ser útil modificar dinámicamente las opciones de la cache? Un ejemplo práctico puede ser el de una página que es diferente para los usuarios autenticados y para los usuarios anónimos, aunque la URL sea la misma. Si se dispone de una página creada por la acción `articulo/ver` y que contiene un sistema de puntuación para los artículos, el sistema de puntuación podría estar deshabilitado para los usuarios anónimos. Para este tipo de usuarios, se muestra el formulario para registrarse cuando pinchan en el sistema de puntuación. Esta versión de la página se puede guardar tranquilamente en la cache. Por otra parte, los usuarios autenticados que pinchan sobre el sistema de puntuación, generan una petición POST que se emplea para calcular la nueva puntuación del artículo. En esta ocasión, la cache se debería deshabilitar para que Symfony cree la página de forma dinámica.

El sitio adecuado para definir las opciones dinámicas de la cache es en un filtro que se ejecute antes de `sfCacheFilter`. De hecho, todo el sistema de cache es un filtro de Symfony, como también lo son las opciones de seguridad. Para habilitar la cache en la acción `articulo/ver` solo cuando el usuario no está autenticado, se crea el archivo `conditionalCacheFilter` en el directorio `lib/` de la aplicación, tal y como se muestra en el listado 12-6.

Listado 12-6 - Configurando la cache mediante PHP, en `frontend/lib/conditionalCacheFilter.class.php`

```
class conditionalCacheFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $contexto = $this->getContext();
        if (!$contexto->getUser()->isAuthenticated())
        {
            foreach ($this->getParameter('pages') as $pagina)
            {
                $contexto->getViewCacheManager()->addCache($pagina['module'],
                $pagina['action'], array('lifeTime' => 86400));
            }
        }

        // Ejecutar el siguiente filtro
        $filterChain->execute();
    }
}
```

Este filtro se debe registrar en el archivo `filters.yml` antes de `sfCacheFilter`, como se muestra en el listado 12-7.

Listado 12-7 - Registrando un filtro propio, en `frontend/config/filters.yml`

```
...
security: ~

conditionalCache:
  class: conditionalCacheFilter
  param:
    pages:
      - { module: articulo, action: ver }

cache: ~
...
```

Para que la cache condicional pueda utilizarse, solo es necesario borrar la cache de Symfony para que se autocargue la clase del nuevo filtro. La cache solo se habilitará para las páginas definidas en el parámetro `pages` y solo para los usuarios que no están autenticados.

El método `addCache()` del objeto `sfViewCacheManager` requiere como parámetros el nombre de un módulo, el nombre de una acción y un array asociativo con las mismas opciones que se definen en el archivo `cache.yml`. Si por ejemplo se necesita guardar en la cache la acción `articulo/ver` con el layout y con un tiempo de vida de 300 segundos, se puede utilizar el siguiente código:

```
$contexto->getViewCacheManager()->addCache('articulo', 'ver', array(
  'withLayout' => true,
  'lifeTime'   => 3600,
));
```

Por defecto, la cache de Symfony guarda sus datos en archivos almacenados en el disco duro del servidor. No obstante, existen métodos alternativos como almacenar los contenidos en la memoria (utilizando `memcached` (<http://www.danga.com/memcached/>) por ejemplo) o en una base de datos (útil si se quiere compartir la cache entre varios servidores o si se quiere poder borrar rápidamente la cache). En cualquier caso, es muy sencillo modificar el modo de almacenamiento de la cache de Symfony porque la clase PHP que utiliza el gestor de la cache está definida en el archivo `factories.yml`.

La clase `sfFileCache` es la factoría que emplea por defecto la cache:

```
view_cache:
  class: sfFileCache
  param:
    automaticCleaningFactor: 0
    cacheDir:                %SF_TEMPLATE_CACHE_DIR%
```

Se puede reemplazar el valor de la opción `class` con una clase propia de almacenamiento de la cache o con una de las alternativas disponibles en Symfony: `sfAPCCache`, `sfEAcceleratorCache`, `sfMemcacheCache`, `sfSQLiteCache` y `sfXCacheCache`. Los parámetros definidos en la clave `param` se pasan al constructor de la clase de la cache en forma de array asociativo. Cualquier clase

definida para controlar el almacenamiento de la cache debe implementar todos los métodos de la clase abstracta `sfCache`. El capítulo 19 explica más en detalle esta característica.

12.1.6. Uso de la cache super rápida

Todas las páginas guardadas en la cache que se han explicado anteriormente implican la ejecución de algo de código PHP. En este tipo de páginas, Symfony carga toda la configuración, crea la respuesta, etc. Si se está completamente seguro de que una página no va a cambiar durante un periodo de tiempo, se puede saltar completamente Symfony si se guarda en la carpeta `web/` el código HTML completo de la página. Este funcionamiento es posible gracias a las opciones del módulo `mod_rewrite` de Apache, siempre que la regla de enrutamiento defina un patrón que no termine en ningún sufijo o en `.html`.

Para guardar las páginas completas en la cache, se puede acceder manualmente a todas las páginas mediante la siguiente instrucción ejecutada en la línea de comandos:

```
| > curl http://frontend.ejemplo.com/usuario/listado.html > web/usuario/listado.html
```

Una vez ejecutado el anterior comando, cada vez que se realice una petición a la acción `usuario/listado`, Apache encuentra la página `listado.html` y la sirve directamente sin llegar a ejecutar Symfony. Aunque la desventaja es que no se puede controlar mediante Symfony las opciones de esa cache (tiempo de vida, borrado automático, etc.) la gran ventaja es el increíble aumento del rendimiento de la aplicación.

Una forma más cómoda de generar estas páginas estáticas es la de utilizar el plugin `sfSuperCache`, que automatiza todo el proceso, permite definir el tiempo de vida de la cache e incluso permite el borrado de las páginas guardadas en la cache. El Capítulo 17 incluye más información sobre los plugins.

Además de la cache de HTML, Symfony dispone de otros dos mecanismos de cache, que son completamente automáticos y transparentes para el programador. En el entorno de producción, la configuración y las traducciones de las plantillas se guardan automáticamente en la cache en los directorios `miproyecto/cache/config/` y `miproyecto/cache/i18n/`.

Los aceleradores PHP (eAccelerator, APC, XCache, etc.), también llamados módulos que guardan los *opcodes* en la cache, mejoran el rendimiento de los scripts PHP al guardar en una cache la versión compilada de los scripts, por lo que se elimina el procesamiento y compilación de los scripts cada vez que se ejecutan. Las clases de Propel contienen muchísimo código PHP, por lo que son las que más se benefician de esta técnica. Todos estos aceleradores son compatibles con Symfony y pueden fácilmente triplicar el rendimiento de cualquier aplicación. Se recomienda su uso en los servidores de producción de las aplicaciones utilizadas por muchos usuarios.

Con un acelerador PHP, se pueden almacenar datos en la memoria mediante la clase `sfProcessCache`, para no tener que realizar el mismo procesamiento en cada petición. Además, si se quiere almacenar el resultado de una función que consume una gran cantidad de CPU para su reutilización posterior, es posible utilizar el objeto `sfFunctionCache`. El Capítulo 18 muestra los detalles sobre estos dos mecanismos.

12.2. Eliminando elementos de la cache

Si se modifican los scripts o los datos de la aplicación, la información de la cache estará desfasada. Para evitar incoherencias y posibles errores, se pueden eliminar partes de la cache de varias formas en función de las necesidades de cada caso.

12.2.1. Borrando toda la cache

La tarea `cache:clear` del comando `symfony` se emplea para borrar la cache (la cache de HTML, de configuración, del sistema de enrutamiento y de la internacionalización). Para borrar solo una parte de la cache, se pueden pasar parámetros, tal y como se muestra en el listado 12-8. Este comando solo se puede ejecutar desde el directorio raíz del proyecto.

Listado 12-8 - Borrando la cache

```
// Borrar toda la cache
> php symfony cache:clear

// Atajo para borrar toda la cache
> php symfony cc

// Borrar sólo la cache de la aplicación frontend
> php symfony cache:clear --app=frontend

// Borrar sólo la cache HTML de la aplicación frontend
> php symfony cache:clear --app=frontend --type=template

// Borrar sólo la cache de configuración de la aplicación frontend
// Los valores permitidos de la opción type son: config, i18n, routing y template
> php symfony cache:clear --app=frontend --type=config

// Borrar sólo la cache de configuración de la aplicación frontend en el entorno de
producción
> php symfony cache:clear --app=frontend --type=config --env=prod
```

12.2.2. Borrando partes de la cache

Cuando se modifican los datos de la base de datos, debería borrarse la cache de las acciones que tienen relación con los datos modificados. Aunque se podría borrar la cache entera, en este caso se borraría también la cache de todas las acciones que no tienen relación con los datos modificados. Por este motivo, Symfony proporciona el método `remove()` del objeto `sfViewCacheManager`. El argumento que se le pasa es una URI interna (tal y como se utilizan por ejemplo en la función `link_to()`) y se elimina la cache de la acción relacionada con esa URI.

Si se dispone de una acción llamada `modificar` en el módulo `usuario`, esta acción modifica el valor de los datos de los objetos `Usuario`. Las páginas de las acciones `listado` y `ver` de este módulo que se guardan en la cache deberían borrarse, ya que en otro caso, se mostrarían datos desfasados. Para borrar estas páginas de la cache, se utiliza el método `remove()` tal y como muestra el listado 12-9.

Listado 12-9 - Borrando la cache de una acción, en modules/usuario/actions/actions.class.php

```

public function executeModificar($peticion)
{
    // Modificar un usuario
    $id_usuario = $peticion->getParameter('id');
    $usuario = UsuarioPeer::retrieveByPk($id_usuario);
    $this->forward404Unless($usuario);
    $usuario->setNombre($peticion->getParameter('nombre'));
    ...
    $usuario->save();

    // Borrar la cache de las acciones relacionadas con este usuario
    $cacheManager = $this->getContext()->getViewCacheManager();
    $cacheManager->remove('usuario/listado');
    $cacheManager->remove('usuario/ver?id='.$id_usuario);
    ...
}

```

Eliminar de la cache los elementos parciales, los componentes y los slots de componentes es un poco más complicado. Como se les puede pasar cualquier tipo de parámetro (incluso objetos), es casi imposible identificar la versión guardada en la cache en cada caso. Como la explicación es idéntica para los tres tipos de elementos, solo se va a explicar el proceso para los elementos parciales. Symfony identifica los elementos parciales almacenados en la cache mediante un prefijo especial (`sf_cache_partial`), el nombre del módulo, el nombre del elemento parcial y una clave única o *hash* generada a partir de todos los parámetros utilizados en la llamada a la función:

```

// Un elemento parcial que se llama así
<?php include_partial('usuario/mi_parcial', array('user' => $user) ?>

// Se identifica en la cache de la siguiente manera
@sf_cache_partial?module=usuario&action=_mi_parcial&sf_cache_key=bf41dd9c84d59f3574a5da244626dcc8

```

En teoría, es posible eliminar un elemento parcial guardado en la cache mediante el método `remove()` siempre que se conozca el valor de todos los parámetros utilizados en ese elemento, aunque en la práctica es casi imposible conseguirlo. Afortunadamente, si se añade un parámetro denominado `sf_cache_key` en la llamada del *helper* `include_partial()`, se puede definir un identificador propio para ese elemento parcial. De esta forma, y como muestra el listado 12-10, es fácil borrar un elemento parcial (como por ejemplo borrar de la cache un elemento parcial que depende de un usuario que ha sido modificado):

Listado 12-10 - Borrando elementos parciales de la cache

```

<?php include_partial('usuario/mi_parcial', array(
    'user'          => $user,
    'sf_cache_key' => $user->getId()
) ?>

// Se identifica en la cache de la siguiente forma
@sf_cache_partial?module=usuario&action=_mi_parcial&sf_cache_key=12

```

```
// Se puede borrar la cache de _mi_parcial para un usuario específico
$cacheManager->remove('@sf_cache_partial?module=usuario&action=_mi_parcial&sf_cache_key='.$user->get
```

Este método no se puede utilizar para borrar todas las versiones de un elemento parcial guardadas en la cache. Más adelante, en la sección "Borrando la cache a mano" se detalla como conseguirlo.

El método `remove()` también se emplea para borrar fragmentos de plantillas. El nombre que identifica a cada fragmento en la cache se compone del prefijo `sf_cache_partial`, el nombre del módulo, el nombre de la acción y el valor de `sf_cache_key` (el identificador único utilizado en la llamada al *helper* `cache()`). El listado 12-11 muestra un ejemplo.

Listado 12-11 - Borrando fragmentos de plantilla en la cache

```
<!-- Código guardado en la cache -->
<?php if (!cache('usuarios')): ?>
    // Lo que sea...
    <?php cache_save() ?>
<?php endif; ?>

// Se identifica en la cache de la siguiente forma
@sf_cache_partial?module=usuario&action=listado&sf_cache_key=usuarios

// Se puede borrar con el siguiente método
$cacheManager->remove('@sf_cache_partial?module=usuario&action=listado&sf_cache_key=usuarios');
```

La parte más complicada del borrado de la cache es la de determinar que acciones se ven afectadas por la modificación de los datos.

Imagina que dispones de una aplicación con un módulo llamado `publicacion` y las acciones `listado` y `ver`, además de estar relacionada con un autor (representado por la clase `Usuario`). Si se modifican los datos de un `Usuario`, se verán afectadas todas las publicaciones de ese autor y el listado de las publicaciones. Por tanto, en la acción `modificar` del módulo `usuario` se debería añadir lo siguiente:

```
$c = new Criteria();
$c->add(PublicacionPeer::AUTOR_ID, $peticion->getParameter('id'));
$publicaciones = PublicacionPeer::doSelect($c);

$cacheManager = sfContext::getInstance()->getViewCacheManager();
foreach ($publicaciones as $publicacion)
{
    $cacheManager->remove('publicacion/ver?id='.$publicacion->getId());
}
$cacheManager->remove('publicacion/listado');
```

Si se utiliza la cache HTML, es necesario disponer de una visión clara de las dependencias y relaciones entre el modelo y las acciones, de forma que no se produzcan errores por no comprender completamente esas relaciones. Debe tenerse en cuenta que todas las acciones que modifican el modelo seguramente deben incluir una serie de llamadas al método `remove()` si se utiliza la cache HTML.

Cuando la situación sea realmente complicada, siempre se puede borrar la cache entera cada vez que se actualiza la base de datos.

12.2.3. Borrando simultáneamente varias partes de la cache

El método `remove()` también acepta comodines en el nombre de las claves. De esta forma, es posible borrar varias partes de la cache en una única llamada:

```
// Borra de la cache Las páginas de todos los usuarios  
$cacheManager->remove('usuario/ver?id=*');
```

Otro ejemplo de uso de comodines es el de las aplicaciones disponibles en varios idiomas, donde el código del idioma aparece en todas las URL. En este caso, la URL de la página del perfil de un usuario será similar a la siguiente:

```
| http://www.miaplicacion.com/en/usuario/ver/id/12
```

Para eliminar de la cache las páginas en cualquier idioma del usuario cuyo id es 12, se puede utilizar la siguiente instrucción:

```
| $cache->remove('usuario/ver?sf_culture=*&id=12');
```

Lo anterior también funciona en los elementos parciales:

```
// Utiliza un comodín en el nombre de la clave para borrar todas las claves  
$cacheManager->remove('@sf_cache_partial?module=usuario&action=_mi_parcial&sf_cache_key=*');
```

El método `remove()` acepta otros dos argumentos opcionales, que permiten definir las cabeceras `host` y `vary` para las que quieres borrar elementos de la cache. Symfony guarda en la cache una versión de la página para cada valor diferente de las cabeceras `host` y `vary`, por lo que si dos aplicaciones tienen el mismo código pero diferente `hostname`, las dos utilizan diferentes caches. La mayor utilidad de esta característica se da en las aplicaciones que interpretan los subdominios como parámetros de la petición (como php en la dirección <http://php.askeet.com> o life en <http://life.askeet.com>). Si no se indican los últimos dos parámetros, Symfony borra la cache para el `host` actual y para el valor `all` de la cabecera `vary`. A continuación se muestran ejemplos de cómo borrar la cache para diferentes `host` utilizando el método `remove()`:

```
// Borra de la cache Las páginas de todos los usuarios para el host actual  
$cacheManager->remove('usuario/ver?id=*');  
// Borra de la cache Las páginas de todos los usuarios para el host life-askeet.com  
$cacheManager->remove('usuario/ver?id=*', 'life.askeet.com');  
// Borra de la cache Las páginas de todos los usuarios para todos los hosts  
$cacheManager->remove('usuario/ver?id=*', '*');
```

El método `remove()` funciona con todos los métodos de cache que se pueden definir en el archivo de configuración `factories.yml` (no sólo con `sfFileCache` sino también con `sfAPCCache`, `sfEAcceleratorCache`, `sfMemcacheCache`, `sfSQLiteCache` y `sfXCacheCache`).

12.2.4. Borrado de la cache de otras aplicaciones

El borrado de la cache de otras aplicaciones no es una tarea sencilla. Imagina que un administrador modifica un registro en la tabla `usuario` de la aplicación backend. Tras la modificación, todas las

acciones que dependen de ese usuario en la aplicación frontend deben ser borradas de la cache. Sin embargo, el gestor de la cache de la aplicación backend no conoce las reglas de enrutamiento de la aplicación frontend porque las aplicaciones se encuentran aisladas entre sí. Por lo tanto, no es posible utilizar código similar al siguiente:

```
// Primero se obtiene el gestor de la cache del backend
$cacheManager = sfContext::getInstance()->getViewCacheManager();

// El patrón no se encuentra porque la plantilla está en la cache del frontend
$cacheManager->remove('usuario/ver?id=12');
```

La solución consiste en inicializar manualmente un objeto de tipo `sfCache` con las mismas opciones que el gestor de la cache del frontend. Afortunadamente, todas las clases de la cache en Symfony incluyen un método llamado `removePattern()` con la misma funcionalidad que el método `remove()` del gestor de la cache.

Si por ejemplo la aplicación backend tiene que borrar la cache de la acción `usuario/ver` en la aplicación frontend para el usuario cuyo atributo `id` es 12, se puede utilizar la siguiente instrucción:

```
$directorio_cache_frontend =
sfConfig::get('sf_root_cache_dir').DIRECTORY_SEPARATOR.'frontend'.DIRECTORY_SEPARATOR.SF_ENV.DIRECTO
// Utiliza las mismas opciones que el archivo factories.yml de la aplicación frontend
$cache = new sfFileCache(array('cache_dir' => $directorio_cache_frontend));
$cache->removePattern('usuario/ver?id=12');
```

Si utilizas otros mecanismos de cache, sólo es preciso cambiar la inicialización del objeto de la cache, ya que el proceso de borrado de la cache es idéntico:

```
$cache = new sfMemcacheCache(array('prefix' => 'frontend'));
$cache->removePattern('usuario/ver?id=12');
```

12.3. Probando y monitorizando la cache

La cache de HTML puede provocar incoherencias en los datos mostrados si no se gestiona correctamente. Cada vez que se activa la cache para un elemento, se debe probar y monitorizar la mejora obtenida en el rendimiento de su ejecución.

12.3.1. Creando un entorno de ejecución intermedio

El sistema de cache es propenso a crear errores en el entorno de producción que no se pueden detectar en el entorno de desarrollo, ya que en este último entorno la cache HTML está deshabilitada por defecto. Si se habilita la cache de HTML para algunas acciones, se debería crear un nuevo entorno de ejecución llamado `staging` en este capítulo y con las mismas opciones que el entorno `prod` (por lo tanto con la cache activada) pero con la opción `web_debug` activada (valor `on`).

Para crear el nuevo entorno, se deben añadir las líneas mostradas en el listado 12-12 al archivo `settings.yml` de la aplicación.

Listado 12-12 - Opciones del entorno `staging`, en `frontend/config/settings.yml`

```
staging:
  .settings:
    web_debug: on
    cache:     on
```

Además, se debe crear un nuevo controlador frontal copiando el de producción (que seguramente se llamará `miproyecto/web/index.php`) en un archivo llamado `frontend_staging.php`. En este archivo copiado es necesario modificar los argumentos que se pasan al método `getApplicationConfiguration()`, tal y como se muestra a continuación:

```
$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'staging', true);
```

Y sólo con esos cambios ya se dispone de un nuevo entorno de ejecución. Para probarlo, se añade el nombre del controlador frontal a la URL después del nombre de dominio:

```
| http://miaplicacion.ejemplo.com/frontend_staging.php/usuario/listado
```

12.3.2. Monitorizando el rendimiento

El Capítulo 16 describe en detalle la barra de depuración de aplicaciones y sus contenidos. No obstante, como esa barra también contiene información relacionada con los elementos guardados en la cache, se incluye ahora una breve descripción de sus características relacionadas con la cache.

Cuando se accede a una página que contiene elementos susceptibles de estar en la cache (acciones, elementos parciales, fragmentos, etc.) la barra de depuración web (que aparece en la esquina superior izquierda) muestra un botón para ignorar la cache (una flecha curvada verde), como se puede ver en la figura 12-4. Este botón se emplea para recargar la página y forzar a que se procesen todos los elementos que estaban en la cache. Se debe tener en cuenta que este botón no borra la cache.

El último número que se muestra en la derecha de la barra es el tiempo que ha durado la ejecución de la petición. Si se habilita la cache en una página, este número debería ser muy inferior al recargar la página, ya que Symfony utilizará los datos de la cache en vez de volver a ejecutar por completo los scripts. Este indicador se puede utilizar para monitorizar fácilmente las mejoras introducidas por la cache.



Figura 12.4. Barra de depuración web en las páginas que utilizan la cache

La barra de depuración también muestra el número de consultas de base de datos que se han ejecutado para la petición, el detalle del tiempo de ejecución de cada categoría (se muestra al pulsar sobre el tiempo de ejecución total). Monitorizando esta información es sencillo medir las mejoras en el rendimiento que son debidas a la cache.

12.3.3. Pruebas de rendimiento (benchmarking)

La depuración de las aplicaciones reduce notablemente la velocidad de ejecución de la aplicación, ya que se genera mucha información para que esté disponible en la barra de

depuración web. De esta forma, el tiempo total de ejecución que se muestra cuando se accede a la aplicación en el entorno `staging` no es representativo del tiempo que se empleará en producción, donde la depuración está deshabilitada.

Para obtener información sobre el tiempo de ejecución de cada petición, deberían utilizarse herramientas para realizar pruebas de rendimiento, como Apache Bench o JMeter. Estas herramientas permiten realizar pruebas de carga y calculan dos parámetros muy importantes: el tiempo de carga medio de una página y la capacidad máxima del servidor. El tiempo medio de carga es esencial para monitorizar las mejoras de rendimiento introducidas por la activación de la cache.

12.3.4. Identificando elementos de la cache

Cuando la barra de depuración web está activada, los elementos de la página que se encuentran en la cache se identifican mediante un recuadro rojo, además de que cada uno dispone de una caja de información sobre la cache en la esquina superior izquierda del elemento, como muestra la figura 12-5. La caja muestra un fondo azul si el elemento se ha ejecutado y un fondo de color amarillo si se ha obtenido directamente de la cache. Al pulsar sobre el enlace de información de la cache se muestra el identificador del elemento en la cache, su tiempo de vida y el tiempo que ha transcurrido desde su última modificación. Esta información es útil para resolver problemas con elementos fuera de contexto, para ver cuando se crearon los elementos y para visualizar las partes de la plantilla que se pueden guardar en la cache.

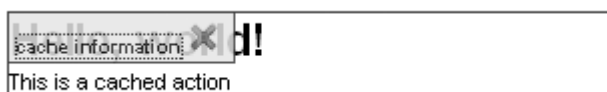


Figura 12.5. Identificación de los elementos de la página que se guardan en la cache

12.4. HTTP 1.1 y la cache del lado del cliente

El protocolo HTTP 1.1 define una serie de cabeceras que se pueden utilizar para acelerar una aplicación controlando la cache del navegador del usuario.

La especificación del protocolo HTTP 1.1 publicada por el W3C (World Wide Web Consortium) define todas las cabeceras con gran detalle (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>). Si una acción tiene habilitada la cache y utiliza la opción `with_layout`, entonces puede hacer uso de los mecanismos que se describen en las siguientes secciones.

Aunque algunos de los navegadores de los usuarios no soporten HTTP 1.1, no existe ningún riesgo en utilizar las opciones de cache de HTTP 1.1. Los navegadores que reciben cabeceras que no entienden, simplemente las ignoran, por lo que se aconseja utilizar los mecanismos de cache de HTTP 1.1.

Además, las cabeceras de HTTP 1.1 también las interpretan los servidores proxy y servidores cache. De esta forma, aunque el navegador del usuario no soporte HTTP 1.1, puede haber un proxy en la ruta de la petición que pueda aprovechar esas características.

12.4.1. Uso de la cabecera ETag para evitar el envío de contenidos no modificados

Cuando se habilita la característica de ETag, el servidor web añade a la respuesta una cabecera especial que contiene una *firma* de la respuesta enviada.

```
| ETag: "1A2Z3E4R5T6Y7U"
```

El navegador del usuario almacena esta firma y la envía junto con la petición la próxima vez que el usuario acceda a la misma página. Si la firma demuestra que la página no se ha modificado desde la primera petición, el servidor no envía de nuevo la página de respuesta. En su lugar, envía una cabecera de tipo 304: `Not modified`. Esta técnica ahorra tiempo de CPU (si se está utilizando la compresión de contenidos) y ancho de banda (ya que la página no se vuelve a enviar) en el servidor, y tiempo de carga (porque la página no se envía de nuevo) en el cliente. En resumen, las páginas que se guardan en la cache con la cabecera ETag son todavía más rápidas de cargar que las páginas que están en la cache y no tienen ETag.

Symfony permite activar la característica ETag para toda la aplicación en el archivo `settings.yml`. El valor por defecto de la opción ETag se muestra a continuación:

```
| all:
|   .settings:
|     etag: on
```

En las acciones que se guardan en la cache junto con el layout, la respuesta se obtiene directamente del directorio `cache/`, por lo que el proceso es todavía más rápido.

12.4.2. Añadiendo la cabecera Last-Modified para evitar el envío de contenidos todavía válidos

Cuando el servidor envía la respuesta al navegador, puede añadir una cabecera especial que indica cuando se modificaron por última vez los datos contenidos en la página:

```
| Last-Modified: Sat, 23 Nov 2006 13:27:31 GMT
```

Los navegadores interpretan esta cabecera y la próxima vez que solicitan la misma página, añaden una cabecera `If-Modified` apropiada:

```
| If-Modified-Since: Sat, 23 Nov 2006 13:27:31 GMT
```

El servidor entonces puede comparar el valor enviado por el cliente y el valor devuelto por la aplicación. Si coinciden, el servidor devuelve una cabecera 304: `Not modified`, ahorrando ancho de banda y tiempo de CPU, al igual que sucedía con la cabecera ETag.

Symfony permite establecer la cabecera `Last-Modified` de la misma forma que se establece cualquier otra cabecera. En una acción se puede añadir de la siguiente manera:

```
| $this->getResponse()->setHttpHeader('Last-Modified',
| $this->getResponse()->getDate($timestamp));
```

La fecha puede ser la fecha actual o la fecha de la última actualización de los datos de la página, obtenida a partir de la base de datos o del sistema de archivos. El método `getDate()` del objeto

`sfResponse` convierte un *timestamp* en una fecha formateada según el estándar requerido por la cabecera `Last-Modified` (RFC1123).

12.4.3. Añadiendo cabeceras `Vary` para permitir varias versiones de la página en la cache

Otra de las cabeceras de HTTP 1.1 es `Vary`, que define los parámetros de los que depende una página y que utilizan los navegadores y los servidores proxy para organizar la cache de las páginas. Si por ejemplo el contenido de una página depende de las cookies, se puede utilizar la siguiente cabecera `Vary`:

```
| Vary: Cookie
```

En la mayoría de ocasiones, es difícil habilitar la cache para las acciones porque la página puede variar en función de las cookies, el idioma del usuario o cualquier otro parámetro. Si no es un inconveniente aumentar el tamaño de la cache, se puede utilizar en este caso la cabecera `Vary`. Además, se puede emplear esta cabecera para toda la aplicación o solo para algunas acciones, definiéndolo en el archivo de configuración `cache.yml` o mediante el método disponible en `sfResponse`, como se muestra a continuación:

```
| $this->getResponse()->addVaryHTTPHeader('Cookie');  
| $this->getResponse()->addVaryHTTPHeader('User-Agent');  
| $this->getResponse()->addVaryHTTPHeader('Accept-Language');
```

Symfony guarda en la cache versiones diferentes de la página en función de cada uno de estos parámetros. Aunque el tamaño de la cache aumenta, la ventaja es que cuando el servidor recibe una petición que coincide con estas cabeceras, la respuesta se obtiene directamente de la cache en vez de tener que procesarla. Se trata de un mecanismo muy útil para mejorar el rendimiento de las páginas que solo varían en función de las cabeceras de la petición.

12.4.4. Añadiendo la cabecera `Cache-Control` para permitir la cache en el lado del cliente

Hasta ahora, aunque se hayan añadido las cabeceras, el navegador sigue enviando peticiones al servidor a pesar de disponer de una versión de la página en su cache. Para evitar estas peticiones, se pueden añadir las cabeceras `Cache-Control` y `Expires` a la respuesta. PHP deshabilita por defecto estas cabeceras, pero Symfony puede saltarse este comportamiento para evitar las peticiones innecesarias al servidor.

Como es habitual, esta opción se activa mediante un método del objeto `sfResponse`. En una acción se puede definir el tiempo máximo que una página debería permanecer en la cache (en segundos):

```
| $this->getResponse()->addCacheControlHTTPHeader('max_age=60');
```

Además, se pueden especificar las condiciones bajo las cuales se guarda la página en la cache, de forma que la cache del proveedor no almacene por ejemplo datos privados (como números de cuenta y contraseñas):

```
| $this->getResponse()->addCacheControlHTTPHeader('private=True');
```

Mediante el uso de las directivas HTTP de Cache-Control es posible controlar los diversos mecanismos de cache existentes entre el servidor y el navegador del cliente. La especificación del W3C de Cache-Control contiene la explicación detallada de todas estas directivas.

Symfony permite añadir otra cabecera llamada Expires:

```
$this->getResponse()->setHTTPHeader('Expires',  
$this->getResponse()->getDate($timestamp));
```

Sugerencia La consecuencia más importante de activar el mecanismo Cache-Control es que los logs del servidor no muestran todas las peticiones realizadas por los usuarios, sino solamente las que recibe realmente el servidor. De esta forma, si mejora el rendimiento de un sitio web, su popularidad descenderá de forma aparente en las estadísticas de acceso al sitio.

12.5. Resumen

El sistema de cache permite mejorar el rendimiento de la aplicación de forma variable en función del tipo de cache utilizado. La siguiente lista muestra los tipos de cache disponibles en Symfony ordenados de mayor a menor mejora en el rendimiento de la aplicación:

- Super cache
- Cache de una acción con layout
- Cache de una acción sin layout
- Cache de fragmentos de plantillas

Además, también se pueden guardar en la cache los elementos parciales y los componentes.

Si la modificación de los datos del modelo o de la sesión obliga a borrar la cache para mantener la coherencia de la información, se puede realizar un borrado muy selectivo para no penalizar el rendimiento, ya que es posible borrar solamente los elementos modificados manteniendo todos los demás.

Una recomendación muy importante es la de probar cuidadosamente todas las páginas para las que se ha habilitado la cache, ya que suele ser habitual que se produzcan errores por haber guardado en la cache elementos inadecuados o por no haber borrado de la cache los elementos modificados. Una buena técnica es la de crear un entorno intermedio llamado *staging* dedicado a probar la cache y las mejoras en el rendimiento de la aplicación.

Por último, es posible exprimir al máximo algunas características del protocolo HTTP 1.1 gracias a las opciones que proporciona Symfony para controlar la cache y que permite aprovechar las ventajas de la cache en el navegador de los clientes, de forma que se aumente aun más el rendimiento de la aplicación.

Capítulo 13. Internacionalización y localización

Cuando se desarrollan aplicaciones con soporte para varios idiomas, es fácil que la traducción de todos los contenidos, el soporte de los estándares de cada país y la traducción de la interfaz se conviertan en una pesadilla. Afortunadamente, Symfony automatiza de forma nativa todos los aspectos del proceso de internacionalización.

Como la palabra "internacionalización" es demasiado larga, los programadores normalmente se refieren a ella como `i18n` (18 es el número de letras que existen entre la letra "i" y la letra "n" de la palabra "internacionalización"). La "localización" normalmente se abrevia como `l10n`. Estas 2 palabras se refieren a 2 aspectos diferentes de las aplicaciones web multidioma.

Una aplicación *internacionalizada* dispone de varias versiones de un mismo contenido en diferentes idiomas o formatos. La interfaz de una aplicación web de correo electrónico, puede ofrecer por ejemplo el mismo servicio en diferentes idiomas, cambiando solamente la interfaz.

Una aplicación *localizada* dispone de información diferente en función del país desde el que se accede. El caso más sencillo es el de los contenidos de un portal de noticias: si el usuario accede desde Estados Unidos, se muestran las últimas noticias de Estados Unidos, pero si el usuario accede desde Francia, se mostrarán las noticias de Francia. Por tanto, una aplicación con `l10n` no solo proporciona los contenidos traducidos, sino que todo el contenido puede cambiar de una versión a otra.

En cualquier caso, el soporte de `i18n` y `l10n` en una aplicación comprende los siguientes aspectos:

- Traducción de texto (interfaz, contenidos estáticos y contenido)
- Estándares y formatos (fechas, cantidades, números, etc.)
- Contenido localizado (varias versiones de un mismo objeto en función del país del usuario)

En este capítulo se muestra la forma en la que Symfony trata cada uno de estos elementos y la forma en la que se pueden desarrollar aplicaciones con `i18n` y `l10n`.

13.1. Cultura del usuario

Todas las opciones relacionadas con `i18n` en Symfony se basan en un parámetro de la sesión de usuario llamado `culture` (cultura). La cultura está formada por la combinación del país e idioma del usuario y determina la forma en la que se muestra el texto y la información que depende de la cultura. Como su valor se serializa en la sesión de usuario, la cultura se almacena de forma persistente entre páginas diferentes.

13.1.1. Indicando la cultura por defecto

Por defecto, la cultura de los nuevos usuarios toma el valor de la opción `default_culture`. Se puede modificar su valor en el archivo de configuración `settings.yml`, como se muestra en el listado 13-1.

Listado 13-1 - Indicando la cultura por defecto, en `frontend/config/settings.yml`

```
all:
  .settings:
    default_culture:    fr_FR
```

Nota Durante el desarrollo de la aplicación, es posible que los cambios en el archivo `settings.yml` no se reflejen en la aplicación accedida mediante el navegador. La razón es que la sesión guarda la información de la cultura de las páginas anteriores. Para acceder a la aplicación con el nuevo valor de la cultura, se deben borrar las cookies del dominio de la aplicación o se debe reiniciar el navegador.

La cultura debe indicar el país y el idioma ya que, por ejemplo, se puede disponer de una traducción al francés diferente para los usuarios de Francia, Bélgica y Canadá, como también se pueden ofrecer traducciones diferentes al español para los usuarios de España y México. El idioma se codifica mediante 2 caracteres en minúscula siguiendo el estándar ISO 639-1 (en para inglés, por ejemplo). El país se codifica en forma de 2 caracteres en mayúscula siguiendo el estándar ISO 3166-1 (GB para Reino Unido, por ejemplo).

13.1.2. Modificando la cultura de un usuario

La cultura de un usuario se puede modificar mientras accede a la aplicación (por ejemplo cuando un usuario decide cambiar la versión en inglés por la versión en francés) o cuando el usuario accede a la aplicación y se utiliza el idioma que ha seleccionado en sus preferencias. Por este motivo la clase `sfUser` ofrece métodos *getter* y *setter* para la cultura del usuario. El listado 13-2 muestra cómo utilizar estos métodos en la acción.

Listado 13-2 - Modificando y obteniendo la cultura en una acción

```
// Modificando la cultura
$this->getUser()->setCulture('en_US');

// Obteniendo la cultura
$cultura = $this->getUser()->getCulture();
=> en_US
```

Cuando se utilizan las opciones de localización e internacionalización de Symfony, parece que existen varias versiones diferentes de una página para una misma URL, ya que la versión que se muestra depende de la sesión de usuario. Este comportamiento hace difícil guardar las páginas en la cache o que las páginas se indexen correctamente en los buscadores.

Una solución es hacer que la cultura se muestre en todas las URL, de forma que las páginas traducidas se muestran como si fueran URL diferentes. Para conseguirlo, se añade la opción `:sf_culture` en todas las reglas del archivo `routing.yml` de la aplicación:

```
pagina:
  url: /:sf_culture/:pagina
```

```
requirements: { sf_culture: (? :fr|en|de) }
params: ...

articulo:
  url: /:sf_culture/:ano/:mes/:dia/:slug
  requirements: { sf_culture: (? :fr|en|de) }
  params: ...
```

Para no tener que añadir manualmente el parámetro de petición `sf_culture` en todas las llamadas a `link_to()`, Symfony añade automáticamente la cultura del usuario a los parámetros de enrutamiento por defecto. También funciona de forma inversa, ya que Symfony modifica automáticamente la cultura del usuario si encuentra el parámetro `sf_culture` en la URL.

13.1.3. Determinando la cultura de forma automática

En muchas aplicaciones, la cultura del usuario se define durante la primera petición, en función de las preferencias de su navegador. Los usuarios pueden definir en el navegador una serie de idiomas que están dispuestos a aceptar. Esta información se envía al servidor en cada petición, mediante la cabecera HTTP `Accept-Language`. En Symfony esta cabecera se puede acceder a través del objeto `sfWebRequest`. Si por ejemplo se quiere obtener la lista de idiomas preferidos del usuario en una acción, se utiliza la siguiente instrucción:

```
| $idiomas = $peticion->getLanguages();
```

Aunque la cabecera HTTP es una cadena de texto, Symfony la procesa y la convierte automáticamente en un array. Por tanto, el idioma preferido del usuario se puede obtener en el ejemplo anterior mediante `$idiomas[0]`.

En la página principal de un sitio web y en un filtro utilizado en varias páginas, puede ser útil establecer automáticamente la cultura del usuario al idioma preferido del navegador del usuario. Sin embargo, como tu sitio web sólo estará disponible en un número limitado de idiomas, se mejor utilizar el método `getPreferredCulture()`. Este método compara los idiomas preferidos por el usuario y los idiomas disponibles en la aplicación, devolviendo el mejor valor posible:

```
| // El sitio web está disponible sólo en inglés y francés
| $idioma = $peticion->getPreferredCulture(array('en', 'fr'));
```

Si no se producen coincidencias entre los idiomas preferidos por el usuario y los idiomas disponibles en la aplicación, el método anterior simplemente devuelve el primer idioma disponible (en el ejemplo anterior, el método devolvería el valor `en`).

Cuidado La cabecera HTTP `Accept-Language` no es una información muy fiable, ya que casi ningún usuario sabe cómo modificar su valor en el navegador. En la mayoría de los casos, el idioma preferido del navegador es el idioma de la propia interfaz del navegador y los usuarios no están disponibles en todos los idiomas. Si se decide establecer de forma automática el valor de la cultura según el idioma preferido del navegador, es una buena idea proporcionar una forma sencilla de seleccionar otro idioma.

13.2. Estándares y formatos

Las partes internas de una aplicación web no deben preocuparse por las diferencias culturales entre países. Las bases de datos por ejemplo almacenan las fechas y cantidades siguiendo estándares internacionales. Pero cuando los datos se envían o se reciben del usuario, es necesario realizar una conversión. Los usuarios normales no entienden lo que es un timestamp y prefieren llamar a su idioma en su propio idioma (por ejemplo *"Français"* en vez de *"French"*). Así que se debe aprovechar la posibilidad de realizar estas conversiones de forma automática en función de la cultura del usuario.

13.2.1. Mostrando datos según la cultura del usuario

Una vez que se define la cultura del usuario, los *helpers* que dependen de la cultura muestran automáticamente los datos de forma correcta. El *helper* `format_number()` por ejemplo, muestra un número en un formato familiar para el usuario, en función de su cultura, tal y como muestra el listado 13-3.

Listado 13-3 - Mostrando un número según la cultura del usuario

```
<?php use_helper('Number') ?>

<?php $sf_user->setCulture('en_US') ?>
<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php $sf_user->setCulture('fr_FR') ?>
<?php echo format_number(12000.10) ?>
=> '12 000,10'
```

No es necesario indicar a los *helpers* la cultura de forma explícita. Los *helpers* la buscan automáticamente en el objeto sesión. El listado 13-4 muestra todos los *helpers* que tienen en cuenta la cultura para mostrar sus datos.

Listado 13-4 - *Helpers* dependientes de la cultura

```
<?php use_helper('Date') ?>

<?php echo format_date(time()) ?>
=> '9/14/06'

<?php echo format_datetime(time()) ?>
=> 'September 14, 2006 6:11:07 PM CEST'

<?php use_helper('Number') ?>

<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php echo format_currency(1350, 'USD') ?>
=> '$1,350.00'

<?php use_helper('I18N') ?>
```

```

<?php echo format_country('US') ?>
=> 'United States'

<?php format_language('en') ?>
=> 'English'

<?php use_helper('Form') ?>

<?php echo input_date_tag('fecha_nacimiento', mktime(0, 0, 0, 9, 14, 2006)) ?>
=> input type="text" name="fecha_nacimiento" id="fecha_nacimiento" value="9/14/06"
size="11" />

<?php echo select_country_tag('pais', 'US') ?>
=> <select name="pais" id="pais"><option value="AF">Afghanistan</option>
    ...
    <option value="GB">United Kingdom</option>
    <option value="US" selected="selected">United States</option>
    <option value="UM">United States Minor Outlying Islands</option>
    <option value="UY">Uruguay</option>
    ...
</select>

```

Los *helpers* de fechas aceptan un parámetro opcional para indicar su formato, de modo que se pueda mostrar una fecha independiente de la cultura del usuario, pero no debería utilizarse en las aplicaciones con soporte de i18n.

13.2.2. Obteniendo información en una aplicación localizada

Si es necesario obtener información del usuario, se debería obligar al usuario, si es posible, a introducir datos que ya estén internacionalizados. Esta técnica evita tener que adivinar el formato en el que ha introducido el usuario sus datos. Por ejemplo, es complicado que un usuario introduzca una cantidad monetaria con la separación de los miles.

Se pueden restringir las posibilidades del usuario ocultando los datos realmente enviados al servidor (como por ejemplo mediante `select_country_tag()`) o separando las partes de un dato complejo en varias partes individuales sencillas.

No obstante, para datos como fechas esta técnica no siempre es posible. Los usuarios están acostumbrados a introducir las fechas en el formato propio de su país, por lo que se deben convertir a un formato internacional. Para ello se puede utilizar la clase `SfI18N`. El listado 13-5 muestra cómo utilizar esta clase.

Listado 13-5 - Obteniendo una fecha a partir de un formato propio del usuario en una acción

```

$fecha= $peticion->getParameter('fecha_nacimiento');
$cultura_usuario = $this->getUser()->getCulture();

// Obtener un timestamp
$timestamp = $this->getContext()->getI18N()->getTimestampForCulture($fecha,
$cultura_usuario);

// Obtener las partes de una fecha

```



```
list($dia, $mes, $ano) = $this->getContext()->getI18N()->getDateForCulture($fecha,
$cultura_usuario);
```

13.3. Información textual en la base de datos

Una aplicación que soporta la localización ofrece diferentes contenidos en función de la cultura del usuario. Una tienda online podría ofrecer los mismos productos al mismo precio en todo el mundo, pero con una descripción personalizada para cada país. De esta forma, la base de datos tiene que ser capaz de almacenar diferentes versiones de una misma información y el esquema de la base de datos debe diseñarse de una forma especial, además de indicar la cultura cada vez que se manipulan los objetos del modelo.

13.3.1. Creando un esquema para una aplicación localizada

Cada una de las tablas que contiene información localizada, se debe dividir en 2 partes: una tabla que no contiene ninguna información relativa a la i18n y otra tabla con todas las columnas relacionadas con la i18n. Las dos tablas tienen una relación de tipo *"uno a muchos"*. De esta forma, es posible añadir más idiomas sin tener que modificar el modelo. Como ejemplo se va a considerar una tabla llamada `Producto`.

En primer lugar, se crean las tablas en el archivo `schema.yml`, tal y como muestra el listado 13-6.

Listado 13-6 - Esquema de ejemplo para datos i18n, en `config/schema.yml`

```
mi_conexion:
  mi_producto:
    _attributes: { phpName: Producto, isI18N: true, i18nTable: mi_producto_i18n }
    id:          { type: integer, required: true, primaryKey: true, autoincrement: true }
  }
  precio:       { type: float }

  mi_producto_i18n:
    _attributes: { phpName: ProductoI18n }
    id:          { type: integer, required: true, primaryKey: true, foreignTable:
mi_producto, foreignReference: id }
    culture:     { isCulture: true, type: varchar, size: 7, required: true, primaryKey:
true }
    nombre:     { type: varchar, size: 50 }
```

Lo más importante del listado anterior son los atributos `isI18N` y `i18nTable` de la primera tabla y la columna especial `culture` en la segunda. Todos estos atributos son mejoras de Propel creadas por Symfony.

Symfony puede automatizar aun más este proceso. Si la tabla que contiene los datos internacionalizados tiene el mismo nombre que la tabla principal seguido de un sufijo `_i18n` y ambas están relacionadas con una columna llamada `id`, se pueden omitir las columnas `id` y `culture` en la tabla `_i18n` y los atributos específicos para i18n en la tabla principal. Si se siguen estas convenciones, Symfony es capaz de inferir toda esta información. Así, para Symfony es equivalente el esquema del listado 13-7 al listado 13-6 mostrado anteriormente.

Listado 13-7 - Versión abreviada del esquema de ejemplo para datos i18n, en config/schema.yml

```
mi_conexion:
  mi_producto:
    _attributes: { phpName: Producto }
    id:
    precio:      float
  mi_producto_i18n:
    _attributes: { phpName: ProductoI18n }
    nombre:      varchar(50)
```

13.3.2. Usando los objetos i18n generados

Una vez construido el modelo de objetos (ejecutando la tarea `propel:build-model` después de cada modificación del archivo `schema.yml`), se puede utilizar la clase `Producto` con soporte de i18n como si fuera una sola tabla, tal y como muestra el listado 13-8.

Listado 13-8 - Trabajando con objetos i18n

```
$producto = ProductoPeer::retrieveByPk(1);
$producto->setName('Nom du produit'); // La cultura por defecto es la del usuario actual
$producto->save();

echo $producto->getName();
=> 'Nom du produit'

$producto->setName('Product name', 'en'); // Modificamos el valor para la cultura 'en'
(inglés)
$producto->save();

echo $producto->getName('en');
=> 'Product name'
```

Si no se quiere modificar la cultura cada vez que se utiliza un objeto i18n, es posible modificar el método `hydrate()` en la clase del objeto. El listado 13-9 muestra un ejemplo.

Listado 13-9 - Redefiniendo el método `hydrate()` para establecer la cultura, en `miproyecto/lib/model/Producto.php`

```
public function hydrate(ResultSet $rs, $startcol = 1)
{
    parent::hydrate($rs, $startcol);
    $this->setCulture(sfContext::getInstance()->getUser()->getCulture());
}
```

Las consultas realizadas mediante los objetos *peer* se pueden restringir para que solo obtengan los objetos que disponen de una traducción para la cultura actual, mediante el método `doSelectWithI18n` en lugar del habitual `doSelect`, como muestra el listado 13-10. Además, crea los objetos i18n relacionados a la vez que los objetos normales, de forma que se reduce el número de consultas necesarias para obtener el contenido completo (el Capítulo 18 incluye más información sobre las ventajas de este método sobre el rendimiento de la aplicación).

Listado 13-10 - Obteniendo objetos con un Criterio de tipo i18n

```
$c = new Criteria();
$c->add(ProductoPeer::PRECIO, 100, Criteria::LESS_THAN);
$productos = ProductoPeer::doSelectWithI18n($c, $cultura);
// El argumento $cultura es opcional
// Si no se indica, se utiliza la cultura actual
```

Así que no es necesario trabajar directamente con los objetos i18n, sino que se pasa la cultura al modelo (o se deja que el modelo la obtenga automáticamente) cada vez que se quiere realizar una consulta con un objeto normal.

13.4. Traducción de la interfaz

La interfaz de usuario es otro de los elementos que se deben adaptar en las aplicaciones i18n. Las plantillas tienen que poder mostrar las etiquetas, los mensajes y la navegación en diferentes idiomas pero manteniendo la misma presentación. Symfony recomienda que las plantillas se construyan con el lenguaje por defecto de la aplicación y que se defina la traducción de las frases en un archivo de diccionario. De esta forma, no es necesario modificar las plantillas cada vez que se añade, modifica o elimina una traducción.

13.4.1. Configurando la traducción

Las plantillas no se traducen automáticamente, lo que significa que antes que nada, se debe activar la opción de traducción de las plantillas en el archivo `settings.yml`, como se muestra en el listado 13-11.

Listado 13-11 - Activando la traducción de la interfaz, en `frontend/config/settings.yml`

```
all:
  .settings:
    i18n: on
```

13.4.2. Usando el helper de traducción

En esta sección se va a considerar que se quiere construir un sitio web en inglés y en francés, siendo el inglés el idioma por defecto. Antes de empezar con la traducción del sitio web, una de las plantillas del sitio podría ser similar a la del listado 13-12.

Listado 13-12 - Plantilla con un único idioma

```
| Welcome to our website. Today's date is <?php echo format_date(date()) ?>
```

Para que Symfony pueda traducir las frases de una plantilla, estas deben identificarse como *"texto traducible"*. Para ello se ha definido el *helper* `__()` (2 guiones bajos seguidos), que es parte del grupo de *helpers* llamado `I18N`. De esa forma, todas las plantillas deben encerrar las frases que se van a traducir en llamadas a ese *helper*. El listado 13-12 por ejemplo se puede modificar para que tenga el aspecto del listado 13-13 (como se verá más adelante en la sección "Cómo realizar traducciones complejas", existe una forma mejor para llamar al *helper* de traducción).

Listado 13-13 - Plantilla preparada para múltiples idiomas

```
<?php use_helper('I18N') ?>

<?php echo __('Welcome to our website.') ?>
<?php echo __('Today's date is ') ?>
<?php echo format_date(date()) ?>
```

Sugerencia Si la aplicación hace uso del grupo de *helpers* I18N en todas sus páginas, puede ser una buena idea incluirlo en la opción `standard_helpers` del archivo `settings.yml`, de forma que no sea necesario incluir `use_helper('I18N')` en cada plantilla.

13.4.3. Utilizando un archivo de diccionario

Cuando se invoca la función `__()`, Symfony busca la traducción del argumento que se le pasa en el diccionario correspondiente a la cultura del usuario. Si se encuentra una frase equivalente, la función devuelve la traducción y se muestra en la respuesta. De esta forma, la traducción de la interfaz se basa en los archivos de diccionario.

Los archivos de diccionario se crean siguiendo el formato XLIFF (XML Localization Interchange File Format), sus nombres siguen el patrón `messages.[codigo de idioma].xml` y se guardan en el directorio `i18n/` de la aplicación.

XLIFF es un formato estándar basado en XML. Como se trata de un formato muy utilizado, se pueden emplear herramientas externas que facilitan la traducción del sitio web entero. Las empresas que se encargan de realizar traducciones manejan este tipo de archivos y saben cómo traducir un sitio web entero añadiendo un nuevo archivo XLIFF.

Sugerencia Además del estándar XLIFF, Symfony también permite utilizar otros sistemas para guardar los diccionarios: gettext, MySQL, SQLite y Creole. La documentación de la API contiene toda la información sobre la configuración de estos métodos alternativos.

El listado 13-14 muestra un ejemplo de la sintaxis XLIFF necesaria para crear el archivo `messages.fr.xml` que traduce al francés los contenidos del listado 13-13.

Listado 13-14 - Diccionario en formato XLIFF, en `frontend/i18n/messages.fr.xml`

```
<?xml version="1.0" ?>
<xliff version="1.0">
  <file original="global" source-language="en_US" datatype="plaintext">
    <body>
      <trans-unit id="1">
        <source>Welcome to our website.</source>
        <target>Bienvenue sur notre site web.</target>
      </trans-unit>
      <trans-unit id="2">
        <source>Today's date is </source>
        <target>La date d'aujourd'hui est </target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

El atributo `source-language` siempre contiene el código ISO completo correspondiente a la cultura por defecto. Cada frase o elemento que se traduce, se indica en una etiqueta `trans-unit` con un atributo `id` único.

Si en la aplicación se utiliza la cultura por defecto (en este ejemplo `en_US`), las frases no se traducen y por tanto se muestran directamente los argumentos indicados en las llamadas a `__()`. El resultado del listado 13-13 es similar al listado 13-12. Sin embargo, si se modifica la cultura a `fr_FR` o `fr_BE`, se muestran las traducciones del archivo `messages.fr.xml`, y el resultado es el que se muestra en el listado 13-15.

Listado 13-15 - Una plantilla traducida

```
| Bienvenue sur notre site web. La date d'aujourd'hui est  
| <?php echo format_date(date()) ?>
```

Si se necesita añadir una nueva traducción, solamente es preciso crear un nuevo archivo `messages.XX.xml` de traducción en el mismo directorio que el resto de traducciones.

Sugerencia Como procesar los archivos de los diccionarios y buscar las traducciones es un proceso que consume un tiempo apreciable, Symfony utiliza una cache interna para mejorar el rendimiento. Por defecto esta cache utiliza archivos, pero es posible configurarla en el archivo de configuración `factories.yml` (ver capítulo 19). De esta forma es posible por ejemplo compartir una misma cache entre diferentes servidores.

13.4.4. Trabajando con diccionarios

Si el archivo `messages.XX.xml` aumenta tanto de tamaño como para hacerlo difícil de manejar, se pueden dividir sus contenidos en varios archivos de diccionarios ordenados por temas. De esta forma, es posible por ejemplo dividir el archivo `messages.fr.xml` en los siguientes tres archivos dentro del directorio `i18n/`:

- `navegacion.fr.xml`
- `terminos_de_servicio.fr.xml`
- `busqueda.fr.xml`

Siempre que una traducción no se encuentre en el archivo `messages.XX.xml` por defecto, se debe indicar como tercer argumento en la llamada al *helper* `__()` el archivo de diccionario que debe utilizarse. Para traducir una cadena de texto que se encuentra en el archivo `navegacion.fr.xml`, se utilizaría la siguiente instrucción:

```
| <?php echo __('Welcome to our website', null, 'navegacion') ?>
```

Otra forma de organizar los diccionarios es mediante su división en módulos. En vez de crear un solo archivo `messages.XX.xml` para toda la aplicación, se crea un archivo en cada directorio `modules/[nombre_modulo]/i18n/`. Así se consigue que los módulos sean más independientes de la aplicación, lo que es necesario para reutilizarlos, como por ejemplo en los plugins (ver Capítulo 17).

Como actualizar los diccionarios a mano es un proceso muy propenso a cometer errores, Symfony incluye a partir de su versión 1.1 una tarea que permite automatizar todo este proceso. La tarea `i18n:extract` procesa una aplicación Symfony completa y extrae todas las cadenas de texto que se tienen que traducir. Los argumentos que se pasan a esta tarea son el nombre de la aplicación y una cultura:

```
| > php symfony i18n:extract frontend en
```

Por defecto esta tarea no modifica los diccionarios, sino que simplemente muestra el número de cadenas anteriores y actuales de `i18n`. Para añadir las nuevas cadenas de texto al diccionario, se debe utilizar la opción `--auto-save`:

```
| > php symfony i18n:extract --auto-save frontend en
```

También es posible borrar las cadenas de texto anteriores utilizando la opción `--auto-delete`:

```
| > php symfony i18n:extract --auto-save --auto-delete frontend en
```

Nota La tarea `i18n:extract` presenta algunas limitaciones actualmente. En primer lugar sólo funciona con el diccionario por defecto `messages` y sólo es capaz de manejar traducciones basadas en archivos (XLIFF y `gettext`). Además, esta tarea sólo guarda y borra cadenas de texto en el archivo principal `apps/frontend/i18n/messages.XX.xml`

13.4.5. Trabajando con otros elementos que requieren traducción

Otros elementos también pueden requerir ser traducidos:

- Las imágenes, documentos y cualquier otro tipo de contenido estático pueden variar en función de la cultura del usuario. Un ejemplo típico es el de las imágenes que se utilizan para mostrar un contenido de texto con una tipografía muy especial. En este caso, se pueden crear subdirectorios para cada una de las culturas disponibles (utilizando el valor `culture` para el nombre de cada subdirectorio):

```
| <?php echo image_tag($sf_user->getCulture().'/miTexto.gif') ?>
```

- Los mensajes de error de los archivos de validación se muestran automáticamente mediante `__()`, por lo que para traducirlos, solo es necesario añadirlos a los archivos de diccionario.
- Las páginas por defecto de Symfony (página no encontrada, error interno de servidor, acceso restringido, etc.) están escritas en inglés y tienen que reescribirse para las aplicaciones `i18n`. Probablemente, la solución consiste en crear un módulo `default` propio en la aplicación y utilizar `__()` en las plantillas. El Capítulo 19 explica cómo personalizar estas páginas.

13.4.6. Cómo realizar traducciones complejas

La traducción mediante `__()` requiere que se se le pase como argumento una frase completa. Sin embargo, es muy común tener variables mezcladas con el texto en una frase. Aunque puede ser tentador intentar cortar las frases en varios trozos, el resultado es que las llamadas al *helper* pierden su significado. Afortunadamente, el *helper* `__()` dispone de una opción para reemplazar el valor de las variables y que permite crear diccionarios que conservan su significado y

simplifican la traducción. Las etiquetas HTML también se pueden incluir en la llamada al *helper*. El listado 13-16 muestra un ejemplo.

Listado 13-16 - Traduciendo frases con etiquetas HTML y código PHP

```
// Frases originales
Welcome to all the <strong>new</strong> users.<br />
There are <?php echo count_logged() ?> persons logged.

// Ejemplo malo de como traducir las frases anteriores
<?php echo __('Welcome to all the') ?>
<strong><?php echo __('new') ?></strong>
<?php echo __('users') ?>.<br />
<?php echo __('There are') ?>
<?php echo count_logged() ?>
<?php echo __('persons logged') ?>

// Ejemplo correcto para traducir las frases anteriores
<?php echo __('Welcome to all the <strong>new</strong> users') ?> <br />
<?php echo __('There are %1% persons logged', array('%1%' => count_logged())) ?>
```

En este ejemplo, el nombre que se utiliza para la sustitución es %1%, pero puede utilizarse cualquier nombre, ya que el reemplazo se realiza en el *helper* mediante la función `strtr()` de PHP.

Otro de los problemas habituales de las traducciones es el uso del plural. En función del número de resultados, el texto cambia, pero no lo hace de la misma forma en todos los idiomas. La última frase del listado 13-16 por ejemplo no es correcta si `count_logged()` devuelve 0 o 1. Aunque es posible comprobar el valor devuelto por la función y seleccionar la frase adecuada mediante código PHP, esta forma de trabajar es bastante tediosa. Además, cada idioma tiene sus propias reglas gramaticales, por lo que intentar inferir el plural de las palabras puede ser muy complicado. Como se trata de un problema muy habitual, Symfony incluye un *helper* llamado `format_number_choice()`. El listado 13-17 muestra cómo utilizar este *helper*.

Listado 13-17 - Traduciendo las frases en función del valor de los parámetros

```
<?php echo format_number_choice(
    '[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf]There are%1% persons
logged', array('%1%' => count_logged()), count_logged()) ?>
```

El primer argumento está formado por las diferentes posibilidades de frases. El segundo parámetro es el patrón utilizado para reemplazar variables (como con el *helper* `__()`) y es opcional. El tercer argumento es el número utilizado para determinar la frase que se utiliza.

Las frases de las diferentes posibilidades se separan mediante el carácter `|` seguido de un array de valores, utilizando la siguiente sintaxis:

- `[1,2]`: acepta valores entre 1 y 2, ambos incluidos.
- `(1,2)`: acepta valores entre 1 y 2, ambos excluidos.
- `{1,2,3,4}`: sólo se aceptan los valores definidos en este conjunto.

- `[-Inf,0)`: acepta valores mayores o iguales que `-infinito` y que son estrictamente menores que `0`.
- `{n: n % 10 > 1 && n % 10 < 5}`: la condición se cumple para números como 2, 3, 4, 22, 23, 24 (muy útil en idiomas como el polaco y el ruso).

Se puede utilizar cualquier combinación no vacía de paréntesis y corchetes.

Para que la traducción funcione correctamente, el archivo XLIFF debe contener el mensaje tal y como aparece en la llamada al *helper* `format_number_choice()`. El listado 13-18 muestra un ejemplo.

Listado 13-18 - diccionario XLIFF para un argumento de `format_number_choice()`

```
...
<trans-unit id="3">
  <source>[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf]There are%1% persons
logged</source>
  <target>[0]Personne n'est connecté|[1]Une personne est connectée|(1,+Inf]Ily a %1%
personnes en ligne</target>
</trans-unit>
...
```

Si se trabaja con contenidos internacionalizados en las plantillas, es habitual encontrarse con problemas de charsets. Si se emplea un charset propio de un idioma, se debe modificar cada vez que el usuario cambia su cultura. Además, las plantillas escritas en un determinado charset no muestran correctamente los caracteres pertenecientes a otro charset.

Por este motivo, si se utiliza más de una cultura, es muy recomendable crear todas las plantillas con el charset UTF-8 y que el layout declare que su contenido es UTF-8. Si se utiliza siempre UTF-8, es poco probable que se produzcan sorpresas desagradables.

Las aplicaciones construidas con Symfony definen el charset utilizado de forma centralizada en el archivo `settings.yml`. Si se modifica su valor, se modifican todas las cabeceras `content-type` de todas las páginas de respuesta.

```
all:
  .settings:
    charset: utf-8
```

13.4.7. Utilizando el helper de traducción fuera de una plantilla

No todo el texto que se muestra en las páginas viene de las plantillas. Por este motivo, es habitual tener que utilizar el *helper* `__()` en otras partes de la aplicación: acciones, filtros, clases del modelo, etc. El listado 13-19 muestra cómo utilizar el *helper* en una acción mediante la instancia del objeto `I18N` obtenida a través del *singleton* de contexto.

Listado 13-19 - Utilizando `__()` en una acción

```
| $this->getContext()->getI18N()->__($texto, $argumentos, 'mensajes');
```


13.5. Resumen

La internacionalización y localización de las aplicaciones web es una tarea sencilla si se trabaja con el concepto de la cultura del usuario. Los *helpers* utilizan la cultura para mostrar la información en el formato correcto y el contenido localizado que se guardan en la base de datos se ve como si fuera parte de una única tabla. Para la traducción de las interfaces, el *helper* `__()` y los diccionarios XLIFF permiten obtener la máxima flexibilidad con el mínimo trabajo.

Capítulo 14. Generador de la parte de administración

Cuidado Este capítulo describe el generador de la parte de administración de las aplicaciones que todavía utiliza el sistema de formularios de Symfony 1.0. La parte del generador CRUD de este capítulo se ha eliminado y se ha incluido en el libro de formularios de Symfony 1.1 que publicaremos próximamente.

Muchas aplicaciones web se reducen a una mera interfaz de acceso a la información almacenada en una base de datos. Symfony automatiza la tarea repetitiva de crear módulos para manipular datos mediante el uso de objetos Propel. Si el modelo de objetos está bien definido, es posible incluso generar de forma automática la parte de administración completa de un sitio web. En este capítulo se explican los 2 tipos de generadores automáticos incluidos en Symfony: el *scaffolding* (literalmente se puede traducir como "andamiaje") y el generador de la parte de administración. Este último generador se basa en un archivo de configuración especial con su propia sintaxis, por lo que la mayor parte de este capítulo hace referencia a las posibilidades que ofrece el generador de la administración.

14.1. Generación de código en función del modelo

En las aplicaciones web, las operaciones de acceso a los datos se pueden clasificar en una de las siguientes categorías:

- Insertar un registro (*creation*, en inglés)
- Obtener registros (*retrieval*, en inglés)
- Modificar un registro o alguna de sus columnas (*update*, en inglés)
- Borrar un registro (*deletion*, en inglés)

Como estas operaciones son tan comunes, se ha creado un acrónimo para referirse a todas ellas: CRUD (por las iniciales de sus nombres en inglés). Muchas páginas se reducen a alguna de esas operaciones. En un foro por ejemplo, el listado de los últimos mensajes es una operación de obtener registros y responder a un mensaje se corresponde con la opción de insertar un registro.

En muchas aplicaciones web se crean continuamente acciones y plantillas que realizan las operaciones CRUD para una determinada tabla de datos. En Symfony, el modelo contiene la información necesaria para poder generar de forma automática el código de las operaciones CRUD, de forma que se simplifica el desarrollo inicial de la parte de administración de las aplicaciones.

14.1.1. Modelo de datos de ejemplo

A lo largo de este capítulo, los listados de código muestran las posibilidades del generador de administraciones de Symfony mediante un ejemplo sencillo, similar al utilizado en el Capítulo 8.

Se trata de la típica aplicación para crear un blog, que contiene las clases `Article` y `Comment`. El listado 14-1 muestra el esquema de datos y la figura 14-1 lo ilustra.

Listado 14-1 - Archivo `schema.yml` de la aplicación de ejemplo

```
propel:
  blog_article:
    _attributes: { phpName: Article }
    id:
      title:      varchar(255)
      content:    longvarchar
      created_at:
  blog_comment:
    _attributes: { phpName: Comment }
    id:
      article_id:
      author:     varchar(255)
      content:    longvarchar
      created_at:
```

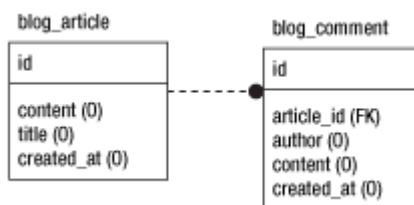


Figura 14.1. Modelo de datos de ejemplo

La generación de código no impone ninguna regla o restricción a la creación del esquema. Symfony utiliza el esquema tal y como se ha definido, interpreta sus atributos y genera la parte de administración de la aplicación.

Sugerencia Para aprovechar al máximo este capítulo, deberías hacer todos los ejemplos que se incluyen. Si se realizan todos los pasos descritos en los listados de código, se obtiene un mejor conocimiento de lo que genera Symfony y de lo que se puede llegar a hacer con el código generado. La recomendación es que crees una estructura de datos como la descrita anteriormente para crear una base de datos con las tablas `blog_article` y `blog_comment`, rellenándolas con datos de prueba.

14.2. Creando la parte de administración de las aplicaciones

Symfony es capaz de generar módulos más avanzados para la parte de gestión o administración de las aplicaciones, también basados en las definiciones de las clases del modelo del archivo `schema.yml`. Se puede crear toda la parte de administración de la aplicación mediante módulos generados automáticamente. Los ejemplos de esta sección describen los módulos de administración creados para una aplicación llamada `backend`. El esqueleto de la aplicación se puede crear mediante la tarea `generate:app` de Symfony:

```
| > php symfony generate:app backend
```

Los módulos de administración interpretan el modelo con la ayuda de un archivo de configuración especial llamado `generator.yml`, que se puede modificar para extender los

componentes generados automáticamente y para controlar el aspecto visual de los módulos. Este tipo de módulos también disponen de los mecanismos habituales descritos en los capítulos anteriores (layout, validación, enrutamiento, configuración propia, carga automática de clases, etc.). Incluso es posible redefinir las acciones y plantillas generadas para incluir características propias, aunque el archivo `generator.yml` es suficiente para realizar la mayoría de modificaciones, por lo que el código PHP solamente es necesario para las tareas muy específicas.

Nota Como el generador de la parte de administración utiliza el sistema de validación de Symfony 1.0, el plugin `sfCompat10` se habilita de forma automática.

14.2.1. Iniciando un módulo de administración

Symfony permite crear la parte de administración de una aplicación módulo a módulo. Los módulos se generan en base a objetos Propel mediante la tarea `propel:init-admin`, que utiliza una sintaxis similar a la que se utiliza para iniciar un *scaffolding*:

```
| > php symfony propel:init-admin backend article Article
```

Este comando es suficiente para crear un módulo llamado `article` en la aplicación `backend` y basado en la definición de la clase `Article`, que además es accesible desde la dirección:

```
| http://localhost/backend.php/article
```

El aspecto visual de los módulos generados automáticamente, que se muestra en las figuras 14-5 y 14-6, es suficiente para incluirlo tal cual en una aplicación comercial.

article list



Id Title		Content	Created at
1	Welcome to the symfony weblog!	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.	December 1, 2006 1:17 PM
2	Life is beautiful	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?	December 1, 2006 1:17 PM
2 results			
			 create

Figura 14.2. Vista "list" del módulo "article" en la aplicación "backend"

edit article

Title:	<input type="text" value="Welcome to the symfony \"/>
Content:	<div>This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.</div>
Created at:	<input type="text" value="12/1/06"/> 

 list
  save
  save and add

 delete

Figura 14.3. Vista "edit" del módulo "article" en la aplicación "backend"

14.2.2. Un vistazo al código generado

El código del módulo de administración `module`, que se encuentra en el directorio `apps/backend/modules/article/`, está completamente vacío porque solo ha sido iniciado. La mejor forma de comprobar el código generado para este módulo es acceder con el navegador a sus páginas y después comprobar los contenidos de la carpeta `cache/`. El listado 14-4 muestra todas las acciones y plantillas generadas que se encuentran en la `cache`.

Listado 14-4 - Elementos de administración generados automáticamente, en `cache/backend/ENV/modules/article/`

```
// En actions/actions.class.php
create      // Redirige a "edit"
delete      // Borra una fila
edit        // Muestra un formulario para modificar la columnas de una fila
            // y procesa el envío del formulario
index       // Redirige a "list"
list        // Muestra un listado de todas las filas de la tabla
save        // Redirige a "edit"

// En templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
```

```

_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
editSuccess.php
listSuccess.php

```

Los módulos de administración generados automáticamente se componen básicamente de las vistas edit y list. Si se observa el código PHP, se encontrará un código muy modular, fácil de leer y extensible.

14.2.3. Conceptos básicos del archivo de configuración generator.yml

Los módulos de administración generados se basan en las opciones del archivo de configuración generator.yml. Las opciones de configuración por defecto para un módulo de administración recién creado llamado article se pueden ver en el archivo backend/modules/article/config/generator.yml, reproducido en el listado 14-5.

Listado 14-5 - Configuración por defecto para la generación de la administración, en backend/modules/article/config/generator.yml

```

generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

```

Esta configuración es suficiente para generar una administración básica. Todas las opciones propias se añaden bajo la clave param, después de la línea theme (lo que significa que todas las líneas que se añadan al final del archivo generator.yml tienen que empezar al menos por 4 espacios en blanco, para que estén correctamente indentadas). El listado 14-6 muestra un archivo generator.yml típico.

Listado 14-6 - Configuración completa típica para el generador

```

generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

    fields:
      author_id:   { name: Article author }

  list:
    title:         List of all articles
    display:       [title, author_id, category_id]
    fields:
      published_on: { params: date_format='dd/MM/yy' }
    layout:        stacked
    params:
      |
      %%is_published%%<strong>%%=title%%</strong><br /><em>by %%author%%
      in %%category%% (%%published_on%%)</em><p>%%content_summary%%</p>

```

```

    filters:      [title, category_id, author_id, is_published]
    max_per_page: 2

  edit:
    title:        Editing article "%%title%%"
    display:
      "Post":     [title, category_id, content]
      "Workflow": [author_id, is_published, created_on]
    fields:
      category_id: { params: disabled=true }
      is_published: { type: plain }
      created_on:   { type: plain, params: date_format='dd/MM/yy' }
      author_id:    { params: size=5 include_custom=>> Choose an author << }
      published_on: { credentials:  }
      content:      { params: rich=true tinymce_options=height:150 }

```

Las siguientes secciones explican en detalle todas las opciones que se pueden utilizar en este archivo de configuración.

14.3. Configuración del generador

El archivo de configuración del generador es muy poderoso, ya que permite modificar la administración generada automáticamente de muchas formas. Lo único malo de que tenga tantas posibilidades es que la descripción completa de su sintaxis es muy larga de leer y cuesta aprenderla, por lo que este capítulo es uno de los más largos del libro. El sitio web de Symfony dispone de un recurso adicional para aprender más fácilmente su sintaxis: la chuleta del generador de la administración, que se puede ver en la figura 14-7 y que se puede descargar desde <http://www.symfony-project.org/uploads/assets/sfAdminGeneratorRefCard.pdf>. Puede ser de utilidad tener la chuleta a mano cuando se leen los ejemplos de este capítulo.

Los ejemplos de esta sección modifican el módulo de administración `article` y también el módulo `comment` basado en la definición de la clase `Comment`. Antes de modificar el módulo `comment`, es necesario crearlo mediante la tarea `propel:init-admin`:

```
| > php symfony propel:init-admin backend comment Comment
```

[illegible]

Figura 14.4. Chuleta del generador de administraciones

14.3.1. Campos

Por defecto, las columnas de la vista `list` y los campos de la vista `edit` son las columnas definidas en el archivo `schema.yml`. El archivo `generator.yml` permite seleccionar los campos que se muestran, los que se ocultan e incluso añadir campos propios (aunque no tengan una correspondencia directa con el modelo de objetos).

14.3.2. Opciones de los campos

El generador de la administración crea un `field` para cada columna del archivo `schema.yml`. Bajo la clave `fields` se puede definir la forma en la que se muestra cada campo, su formato, etc. El ejemplo que se muestra en el listado 14-7 define un valor propio para el atributo `class` y un tipo de campo propio para `title`, además de un título y un mensaje de ayuda para el campo `content`. Las siguientes secciones describen en detalle cómo funciona cada opción.

Listado 14-7 - Establecer un título propio a cada columna

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

  fields:
    title:        { name: Título del artículo, type: textarea tag, params:
```



```
class=foo }
    content:      { name: Cuerpo, help: Rellene el cuerpo del artículo }
```

Además de las opciones globales para todas las vistas, se pueden redefinir las opciones de la clave `fields` para cada una de las vistas (`list` y `edit` en este ejemplo) tal y como muestra el listado 14-8.

Listado 14-8 - Redefiniendo las opciones globales en cada vista

```
generator:
  class:      sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

  fields:
    title:      { name: Título del artículo }
    content:    { name: Cuerpo }

  list:
    fields:
      title:    { name: Título }

  edit:
    fields:
      content:  { name: Cuerpo del artículo }
```

Este ejemplo se puede tomar como una regla general: cualquier opción establecida para todo el módulo mediante la clave `fields`, se puede redefinir en la configuración de cualquier vista (`list` y `edit`).

14.3.2.1. Mostrando nuevos campos

La sección `fields` permite definir para cada vista los campos que se muestran, los que se ocultan, la forma en la que se agrupan y las opciones para ordenarlos. Para ello se emplea la clave `display`. El código del listado 14-9 reordena los campos del módulo `comment`:

Listado 14-9 - Seleccionando los campos que se muestran, en `modules/comment/config/generator.yml`

```
generator:
  class:      sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

  fields:
    article_id:  { name: Artículo }
    created_at:  { name: Publicado en }
    content:     { name: Cuerpo }

  list:
    display:     [id, article_id, content]
```

```

edit:
  display:
    NONE:      [article_id]
    Editable:  [author, content, created_at]

```

Con esta configuración, la vista `list` muestra 3 columnas, como se ve en la figura 14-8 y el formulario de la vista `edit` muestra 4 campos, agrupados en 2 secciones, como se ve en la figura 14-9.

comment list

Id Article Body		
1	1	Well, if this comment displays, it means that your weblog does work...
2	1	Thank you for your feedback. It really helps to see people understanding you.
3	2	Mine is not bad either. I'd like to see more deers, though.
4	2	How can you be so positive? There are so many subjects to worry about out there!
5	2	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results		

+ create

Figura 14.5. Columnas seleccionadas para la vista "list" del módulo "comment"

edit comment

Article:

1 ▼

Editable

Author:

Anonymous

Body:

Well, if this comment displays, it means that your weblog does work...

Published on:

12/1/06

list

✓ save

✓ save and add

delete

Figura 14.6. Agrupando campos en la vista "edit" del módulo "comment"

De esta forma, la opción `display` tiene 2 propósitos:

- Seleccionar las columnas que se muestran y el orden en el que lo hacen. Se utiliza un array simple con el nombre de los campos, como en la vista `list` anterior.
- Agrupar los campos, para lo que se utiliza un array asociativo cuya clave es el nombre del grupo o `NONE` si se quiere definir un grupo sin nombre. Los campos se indican mediante un array simple con los nombres de los campos.

Sugerencia Por defecto, las columnas que son clave primaria no aparecen en ninguna de las vistas.

14.3.2.2. Campos propios

Los campos que se configuran en el archivo `generator.yml` ni siquiera tienen que corresponderse con alguna de las columnas definidas en el esquema. Si la clase relacionada incluye un método *getter* para el campo propio, este se puede utilizar como un campo más de la vista `list`; si además del *getter* existe un método *setter*, el campo también se puede utilizar en la vista `edit`. En el listado 14-10 se muestra un ejemplo que extiende el modelo de `Article` para añadir el método `getNbComments()` que obtiene el número de comentarios de un artículo.

Listado 14-10 - Añadiendo un *getter* propio en el modelo, en `lib/model/Article.php`

```
public function getNbComments()
{
    return $this->countComments();
}
```

Una vez definido este *getter*, el campo `nb_comments` está disponible como campo del módulo generado (el *getter* utiliza como nombre la habitual transformación `camelCase` del nombre del campo) como se muestra en el listado 14-11.

Listado 14-11 - Los *getters* propios permiten añadir más columnas a los módulos de administración, en `backend/modules/article/config/generator.yml`

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

  list:
    display:      [id, title, nb_comments, created_at]
```

La vista `list` resultante se muestra en la figura 14-10.

article list


Id	Title	nb comments	Created at
1	Welcome to the symfony weblog!	2	December 1, 2006 1:17 PM
2	Life is beautiful	3	December 1, 2006 1:17 PM
2 results			
 create			

Figura 14.7. Campo propio en la vista "list" del módulo "article"

Los campos propios también pueden devolver código HTML para mostrarlo directamente. El listado 14-12 por ejemplo extiende la clase `Comment` con un método llamado `getArticleLink()` y que devuelve el enlace al artículo.

Listado 14-12 - Añadiendo un *getter* propio que devuelve código HTML, en `lib/model/Comment.class.php`

```
public function getArticleLink()
{
    return link_to($this->getArticle()->getTitle(), 'article/
edit?id='.$this->getArticleId());
}
```

Este nuevo *getter* se puede utilizar como un campo propio en la vista `comment/list` utilizando la misma sintaxis que en el listado 14-11. El resultado se muestra en el listado 14-13 y se ilustra en la figura 14-11, en la que se puede ver el código HTML generado por el *getter* (un enlace al artículo) en la segunda columna sustituyendo a la clave primaria del artículo.

Listado 14-13 - Los *getter* propios que devuelven código HTML también se pueden utilizar como columnas, en `modules/comment/config/generator.yml`

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

  list:
    display:      [id, article_link, content]
```

comment list

Id	Article link	Body
1	Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
2	Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
3	Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
4	Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
5	Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results		

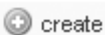


Figura 14.8. Campo propio en la vista "list" del módulo "comment"

14.3.2.3. Campos parciales

El código del modelo debe ser independiente de su presentación. El método `getArticleLink()` de ejemplo anterior no respeta el principio de la separación en capas, porque la capa del modelo incluye cierto código correspondiente a la vista. Para conseguir el mismo efecto pero manteniendo la separación de capas, es mejor incluir el código que genera el HTML del campo propio en un elemento parcial. Afortunadamente, el generador de la administración permite utilizar elementos parciales si la declaración del nombre del campo contiene un guión bajo como primer carácter. De esta forma, el archivo `generator.yml` del listado 14-13 debería modificarse para ser como el del listado 14-14.

Listado 14-14 - Se pueden utilizar elementos parciales como campos, mediante el uso del prefijo _

```
list:
  display:      [id, _article_link, created_at]
```

Para que funcione la configuración anterior, es necesario crear un elemento parcial llamado `_article_link.php` en el directorio `modules/comment/templates/`, tal y como muestra el listado 14-15.

Listado 14-15 - Elemento parcial para la vista list del ejemplo, en `modules/comment/templates/_article_link.php`

```
<?php echo link_to($comment->getArticle()->getTitle(), 'article/
edit?id='.$comment->getArticleId()) ?>
```

La plantilla de un elemento parcial tiene acceso al objeto actual mediante una variable que se llama igual que la clase (`$comment` en este ejemplo). Si se trabajara con un módulo construido para la clase llamada `GrupoUsuario`, el elemento parcial tiene acceso al objeto actual mediante la variable `$grupo_usuario`.

El resultado de este ejemplo es idéntico al mostrado en la figura 14-11, salvo que en este caso se respeta la separación en capas. Si se acostumbra a separar el código en capas, el resultado será que las aplicaciones creadas son más fáciles de mantener.

Si se quieren definir los parámetros para un elemento parcial, se utiliza la misma sintaxis que para un campo normal. Bajo la clave `field` se indican los parámetros y en el nombre del campo no se debe incluir el guión bajo (`_`) inicial. El listado 14-16 muestra un ejemplo.

Listado 14-16 - Las propiedades de un elemento parcial se pueden definir bajo la clave `fields`

```
fields:
  article_link: { name: Artículo }
```

Si el código del elemento parcial crece demasiado, es recomendable sustituirlo por un componente. Para definir un campo basado en un componente, solamente es necesario reemplazar el prefijo `_` por el prefijo `~`, como muestra el listado 14-17.

Listado 14-17 - Se pueden utilizar componentes en los campos, mediante el prefijo `~`

```
...
list:
  display: [id, ~article_link, created_at]
```

En la plantilla que se genera, la configuración anterior resulta en una llamada al componente `articleLink` del módulo actual.

Nota Los campos propios y los campos creados con elementos parciales se pueden utilizar en las vistas `list`, `edit` y en los filtros. Si se utiliza el mismo elemento parcial en varias vistas, la variable `$type` almacena el contexto (`list`, `edit` o `filter`).

14.3.3. Modificando la vista

Si se quiere modificar el aspecto visual de las vistas `edit` y `list`, no se deben modificar las plantillas. Como se generan automáticamente, no es una buena idea modificarlas. En su lugar, se debe utilizar el archivo de configuración `generator.yml`, porque puede hacer prácticamente cualquier cosa que se necesite sin tener que sacrificar la modularidad de la aplicación.

14.3.3.1. Modificando el título de la vista

Además de una serie de campos propios, las páginas `list` y `edit` pueden mostrar un título específico. El listado 14-18 muestra cómo modificar el título de las vistas del módulo `article`. La vista `edit` resultante se ilustra en la figura 14-12.

Listado 14-18 - Estableciendo el título de cada vista, en `backend/modules/article/config/generator.yml`

```
list:
  title: List of Articles
  ...

edit:
```

title:	Body of article %title%
display:	[content]

Body of article Welcome to the symfony weblog!

Content:

This is the first post of this weblog.
 Honestly, it is just a test to check if it works fine. Please comment it as much as you like.

list |
 save |
 save and add |
 delete

Figura 14.9. Título propio en la vista "edit" del módulo "article"

Como los títulos por defecto utilizan el nombre de cada clase, normalmente no es necesario modificarlos (siempre que el modelo utilice nombres de clase explícitos).

Sugerencia En los valores de las opciones del archivo `generator.yml`, se puede acceder al valor de un campo mediante su nombre encerrado entre los caracteres `%`.

14.3.3.2. Añadiendo mensajes de ayuda

En las vistas `list` y `edit`, se pueden añadir *"tooltips"* o mensajes de ayuda para describir los campos que se muestran en los formularios. El listado 14-19 muestra como añadir un mensaje de ayuda para el campo `article_id` en la vista `edit` del módulo `comment`. Para ello, se utiliza la propiedad `help` bajo la clave `fields`. El resultado se muestra en la figura 14-13.

Listado 14-19 - Añadiendo un mensaje de ayuda en la vista `edit`, en `modules/comment/config/generator.yml`

```
edit:
  fields:
    ...
    article_id: { help: The current comment relates to this article }
```

edit comment

Article:

1

The current comment relates to this article

Figura 14.10. Mensaje de ayuda en la vista "edit" del módulo "comment"

En la vista `list`, los mensajes de ayuda se muestran en la cabecera de la columna; en la vista `edit` los mensajes se muestran debajo de cada cuadro de texto.

14.3.3.3. Modificando el formato de la fecha

Las fechas se pueden mostrar siguiendo un formato propio si se utiliza la opción `date_format`, tal y como se muestra en el listado 14-20.

Listado 14-20 - Dando formato a la fecha en la vista list

```
list:
  fields:
    created_at: { name: Published, params: date_format='dd/MM' }
```

La sintaxis que se utiliza es la misma que la del *helper* `format_date()` descrito en el capítulo anterior.

Todo el texto incluido en las plantillas generadas automáticamente está *internacionalizado*, es decir, todos los textos se muestran mediante llamadas al *helper* `__()`. De esta forma, es muy fácil traducir una aplicación de administración generada automáticamente incluyendo la traducción de todas las frases en un archivo XLIFF, en el directorio `apps/frontend/i18n/`, tal y como se explica en el capítulo anterior.

14.3.4. Opciones específicas para la vista "list"

La vista `list` puede mostrar la información de cada fila en varias columnas o de forma conjunta en una sola línea. También puede mostrar opciones para filtrar los resultados, paginación de resultados y opciones para ordenar los datos. Todas estas opciones se pueden modificar mediante los archivos de configuración, como se muestra en las siguientes secciones.

14.3.4.1. Modificando el layout

Por defecto, la unión entre la vista `list` y la vista `edit` se realiza mediante la columna que muestra la clave primaria. Si se observa de nuevo la figura 14-11, se ve que la columna `id` de la lista de comentarios no solo muestra la clave primaria de cada comentario, sino que incluye un enlace que permite a los usuarios acceder de forma directa a la vista `edit`.

Si se quiere mostrar en otra columna el enlace a los datos detallados, se añade el prefijo `=` al nombre de la columna que se utiliza en la clave `display`. El listado 14-21 elimina la columna `id` de la vista `list` de los comentarios y establece el enlace en el campo `content`. La figura 14-14 muestra el resultado de este cambio.

Listado 14-21 - Cambiando el enlace a la vista edit en la vista list, en `modules/comment/config/generator.yml`

```
list:
  display: [article_link, =content]
```


comment list

Article	Body
Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results	

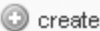


Figura 14.11. Estableciendo el enlace a la vista "edit" en otra columna, en la vista "list" del módulo "comment"

La vista `list` muestra por defecto todos sus datos en varias columnas. También es posible mostrar de forma seguida todos los datos en una sola cadena de texto que ocupe toda la anchura de la tabla. El aspecto con el que se muestran los datos se denomina *"layout"* y la forma en la que se muestran todos seguidos se denomina *stacked*. Si se utiliza el *layout stacked*, la clave `params` debe contener el patrón que define el orden en el que se muestran los datos. El listado 14-22 muestra por ejemplo el *layout* deseado para la vista `list` del módulo `comment`. La figura 14-15 ilustra el resultado final.

Listado 14-22 - Uso del *layout stacked* en la vista `list`, en `modules/comment/config/generator.yml`

```
list:
  layout:  stacked
  params: |
    %=content%% <br />
    (sent by %%author%% on %%created_at%% about %%article_link%%)
  display: [created_at, author, content]
```

comment list

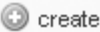
Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
5 results		
		

Figura 14.12. Layout "stacked" en la vista "list" del módulo "comment"

El layout normal en varias columnas requiere un array con el nombre de los campos en la clave `display`; sin embargo, el layout `stacked` requiere que la clave `params` incluya el código HTML que se utilizará para mostrar cada fila de datos. No obstante, el array de la clave `display` también se utiliza en el layout `stacked` para determinar las cabeceras de columnas disponibles para reordenar los datos mostrados.

14.3.4.2. Filtrando los resultados

En la vista de tipo `list`, se pueden añadir fácilmente una serie de filtros. Con estos filtros, los usuarios pueden mostrar menos resultados y pueden obtener más rápidamente los que están buscando. Los filtros se configuran mediante un array con los nombres de los campos en la clave `filters`. El listado 14-23 muestra como incluir un filtro según los campos `article_id`, `author` y `created_at` en la vista `list` del módulo `comment`, y la figura 14-16 ilustra el resultado. Para que el ejemplo funcione correctamente, es necesario añadir un método `__toString()` a la clase `Article` (este método puede devolver, por ejemplo, el valor `title` del artículo).

Listado 14-23 - Incluyendo filtros en la vista `list`, en `modules/comment/config/generator.yml`

```
list:
  filters: [article_id, author, created_at]
  layout: stacked
  params: |
    %=content%<br />
    (sent by %%author%% on %%created_at%% about %%article_link%%)
  display: [created_at, author, content]
```

comment list

Published on	Author	Body
		Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)
		Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)
		Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)
		How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)
		Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)

5 results

[+ create](#)

filters

Article:

Author:

Published on:

[reset](#) [filter](#)

Figura 14.13. Filtros en la vista "list" del módulo "comment"

Los filtros que muestra Symfony dependen del tipo de cada columna:

- Para las columnas de texto (como el campo author en el módulo comment), el filtro es un cuadro de texto que permite realizar búsquedas textuales incluso con comodines (*).
- Para las claves externas (como el campo article_id en el módulo comment), el filtro es una lista desplegable con los datos de la columna correspondiente en la tabla asociada. Como sucede con el `helper object_select_tag()`, las opciones de la lista desplegable son las que devuelve el método `__toString()` de la clase relacionada.
- Para las fechas (como el campo created_at en el módulo comment), el filtro está formado por un par de elementos para seleccionar fechas (que muestran un calendario) de forma que se pueda indicar un intervalo temporal.
- Para las columnas booleanas, el filtro muestra una lista desplegable con los valores true, false y true or false (la última opción es para reinicializar el filtro).

De la misma forma que se pueden utilizar elementos parciales en las listas, también es posible utilizar filtros parciales que permitan definir filtrados que no realiza Symfony. En el siguiente ejemplo se utiliza un campo llamado state que solo puede contener dos valores (open y closed), pero estos valores se almacenan directamente en cada fila de la tabla (no se utiliza una relación con otra tabla). Un filtro de Symfony en este campo mostrará un cuadro de texto, pero lo más lógico sería mostrar una lista desplegable con los dos únicos valores permitidos. Mediante un filtro parcial es fácil mostrar esta lista desplegable. El listado 14-24 muestra un ejemplo de cómo realizar este filtro.

Listado 14-24 - Utilizando un filtro parcial

```
// El elemento parcial se define en templates/_state.php
<?php echo select_tag('filters[state]', options_for_select(array(
    '' => '',
    'open' => 'open',
    'closed' => 'closed',
), isset($filters['state']) ? $filters['state'] : '')) ?>
```

```
// Se añade el filtro parcial en la lista de filtros de config/generator.yml
list:
    filters:      [date, _state]
```

El elemento parcial tiene acceso a la variable `$filters`, que es muy útil para obtener el valor actual del filtro.

Existe una última opción que es muy útil para buscar valores vacíos. Si se quiere filtrar por ejemplo la lista de comentarios para mostrar solamente los que no tienen autor, no se puede dejar vacío el filtro del autor, ya que en este caso se ignorará este filtro. La solución es establecer la opción `filter_is_empty` del campo a `true`, como en el listado 14-25, y el filtro mostrará un checkbox que permite buscar los valores vacíos, tal y como ilustra la figura 14-17.

Listado 14-25 - Filtrando los valores vacíos para el campo author en la vista list

```
list:
    fields:
        author:  { filter_is_empty: true }
    filters:    [article_id, author, created_at]
```

comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		

5 results

+ create

filters

Article:

Author:

Published on:

reset | filter

Figura 14.14. Permitiendo filtrar valores vacíos en el campo "author"

14.3.4.3. Ordenando el listado

Como muestra la figura 14-18, en la vista `list` las columnas que forman la cabecera de la tabla son enlaces que se pueden utilizar para reordenar los datos del listado. Las cabeceras se muestran tanto en el layout normal como en el layout `stacked`. Al pinchar en cualquiera de estos enlaces, se recarga la página añadiendo un parámetro `sort` que permite reordenar los datos de forma adecuada.

List of Articles

Id	Title	nb comments	Created at
1	Welcome to the symfony weblog!	2	December 1, 2006 1:17 PM
2	Life is beautiful	3	December 1, 2006 1:17 PM
2 results			



Figura 14.15. Las cabeceras de la tabla de la vista "list" permiten reordenar los datos

Se puede utilizar la misma sintaxis que emplea Symfony para incluir un enlace que apunte directamente a los datos ordenados de una forma determinada:

```
<?php echo link_to('Listado de comentarios ordenados por fecha', 'comment/
list?sort=created_at&type=desc' ) ?>
```

También es posible indicar en el archivo `generator.yml` el orden por defecto para la vista `list` mediante el parámetro `sort`. El listado 14-26 muestra un ejemplo de la sintaxis que debe utilizarse.

Listado 14-26 - Estableciendo un orden por defecto en la vista `list`


```
list:
  sort:  created_at
  # Sintaxis alternativa para indicar la forma de ordenar
  sort:  [created_at, desc]
```

Nota Solamente se pueden reordenar los datos mediante los campos que se corresponden con columnas reales, no mediante los campos propios y los campos parciales.

14.3.4.4. Modificando la paginación

La administración generada automáticamente tiene en cuenta la posibilidad de que las tablas contengan muchos datos, por lo que la vista `list` incluye por defecto una paginación de datos. Si el número total de filas de la tabla es mayor que el número máximo de filas por página, entonces aparece la paginación al final del listado. La figura 14-19 muestra el ejemplo de un listado con 6 comentarios de prueba para el que el número máximo de comentarios por página es de 5. La paginación de datos asegura un buen rendimiento a la aplicación, porque solamente se obtienen los datos de las filas que se muestran, y permite una buena usabilidad, ya que hasta las filas que contienen millones de filas se pueden manejar con el módulo de administración.

comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
6 results		

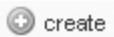


Figura 14.16. La paginación se muestra cuando el listado es muy largo

El número máximo de filas que se muestran en cada página se controla mediante la opción `max_per_page`:

```
list:
    max_per_page: 5
```

14.3.4.5. Mejorando el rendimiento mediante una Join

El generador de la administración utiliza por defecto el método `doSelect()` para obtener las filas de datos. Sin embargo, si se utilizan objetos relacionados en el listado, el número de consultas a la base de datos puede aumentar demasiado. Si se quiere mostrar por ejemplo el nombre del artículo en el listado de comentarios, se debe hacer una consulta adicional por cada comentario para obtener el objeto `Article` relacionado. Afortunadamente, se puede indicar al paginador que utilice un método específico tipo `doSelectJoinXXX()` para optimizar el número de consultas necesario. La opción `peer_method` es la encargada de indicar el método a utilizar.

```
list:
    peer_method: doSelectJoinArticle
```

En el Capítulo 18 se explica más detalladamente el concepto de Join.

14.3.5. Opciones específicas para la vista "edit"

La vista `edit` permite al usuario modificar el valor de cualquier columna de una fila de datos específica. En función del tipo de dato, Symfony determina automáticamente el tipo de campo de formulario que se muestra. Después, genera un *helper* de tipo `object_*_tag()` y le pasa el objeto y la propiedad a editar. Si por ejemplo la configuración de la vista `edit` del artículo estipula que el usuario puede editar el campo `title`:

```
edit:
    display: [title, ...]
```

Entonces, la página edit muestra un cuadro de texto normal para editar el campo title, ya que esta columna se define como de tipo varchar en el esquema.

```
| <?php echo object_input_tag($article, 'getTitle') ?>
```

14.3.5.1. Modificando el tipo de campo de formulario

Las reglas que se utilizan por defecto para determinar el tipo de campo de formulario son las siguientes:

- Las columnas de tipo integer, float, char, varchar(size) se muestran en la vista edit mediante `object_input_tag()`.
- Las columnas de tipo longvarchar aparecen como `object_textarea_tag()`.
- Una columna que es una clave externa, se muestra mediante `object_select_tag()`.
- Las columnas de tipo boolean aparecen como `object_checkbox_tag()`.
- Las columnas de tipo timestamp o date se muestran mediante `object_input_date_tag()`.

En ocasiones, puede ser necesario saltarse estas reglas por defecto para indicar directamente el tipo de campo de formulario utilizado para una columna. Para ello, se utiliza la opción `type` bajo la clave `fields` con el nombre del *helper* que se quiere utilizar. Las opciones del *helper* `object_*_tag()` generado se pueden modificar con la opción `params`. El listado 14-27 muestra un ejemplo.

Listado 14-27 - Indicando un tipo especial de campo de formulario y sus opciones en la vista edit

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

  edit:
    fields:
      id:          ## No se muestra un cuadro de texto, solamente el texto
                  { type: plain }
      author:      ## El contenido del cuadro de texto no se puede modificar
                  { params: disabled=true }
      content:     ## El campo es un textarea (object_textarea_tag)
                  { type: textarea_tag, params: rich=true css=user.css
tinymce_options=width:330 }
      article_id:  ## El campo es una lista desplegable (object_select_tag)
                  { params: include_custom=Choose an article }
      ...
```

La opciones indicadas en `params` se pasan directamente al *helper* `object_*_tag()` generado. La opción `params` del campo `article_id` en el ejemplo anterior produce el siguiente código en la plantilla:

```
<?php echo object_select_tag($comment, 'getArticleId', 'related_class=Article',
'include_custom=Choose an article') ?>
```

De esta forma, todas las opciones disponibles para los *helpers* de formulario se pueden utilizar para modificar la vista edit.

14.3.5.2. Manejando los campos parciales

Las vistas de tipo edit puede utilizar los mismos elementos parciales que se emplean en las vistas de tipo list. La única diferencia es que, en la acción, se debe realizar manualmente la actualización de la columna en función del valor enviado por el elemento parcial. Symfony puede procesar automáticamente los campos normales (los que se corresponden con columnas reales) pero no puede adivinar la forma de tratar los datos que utilizan campos parciales.

Si por ejemplo se define un modulo de administración para una clase User, los campos disponibles pueden ser id, nickname y password. El administrador del sitio web debe ser capaz de modificar la contraseña de un usuario si así se le solicita, pero la vista edit no debería mostrar el valor de la contraseña por motivos de seguridad. En su lugar, el formulario debería mostrar un cuadro de texto vacío para la contraseña y así el usuario puede introducir una nueva contraseña si desea cambiar su valor. Las opciones del archivo generator.yml para una vista edit de este tipo se muestran en el listado 14-28.

Listado 14-28 - Incluyendo un campo parcial en la vista edit

```
edit:
  display:      [id, nickname, _newpassword]
  fields:
    newpassword: { name: Password, help: Introduce una contraseña para
modificar su valor, dejalo vacío para mantener la contraseña actual }
```

El elemento parcial templates/_newpassword.php debe ser similar a:

```
<?php echo input_password_tag('newpassword', '') ?>
```

Este elemento parcial utiliza un *helper* de formulario sencillo y no un *helper* para objetos, ya que no es deseable obtener el valor de la contraseña a partir del objeto User actual, porque podría mostrar la contraseña del usuario.

A continuación, para utilizar el valor de este campo para actualizar el objeto en la acción, se debe extender el método updateUserFromRequest() de la acción. Para ello, se crea un método con el mismo nombre en la clase de la acción y se crea el código necesario para manejar el elemento parcial, como muestra el listado 14-29.

Listado 14-29 - Procesando un campo parcial en la acción, en modules/user/actions/actions.class.php

```
class userActions extends sfActions
{
  protected function updateUserFromRequest($petition)
  {
    // Procesar Los datos del campo parcial
    $password = $petition->getParameter('newpassword');
```



```
        if ($password)
        {
            $this->user->setPassword($password);
        }

        // Dejar que Symfony procese los otros datos
        parent::updateUserFromRequest();
    }
}
```

Nota En una aplicación real, la vista `user/edit` normalmente tendría 2 campos para la contraseña y el valor del segundo campo debe coincidir con el valor del primero para evitar los errores al escribir la contraseña. En la práctica, como se vio en el Capítulo 10, este comportamiento se consigue mediante un validador. Los módulos generados automáticamente pueden hacer uso de este mecanismo de la misma forma que el resto de módulos.

14.3.6. Trabajando con claves externas

Si el esquema de la aplicación define relaciones entre tablas, los módulos generados para la administración pueden aprovechar esas relaciones para automatizar aun más los campos, simplificando enormemente la gestión de las relaciones entre tablas.

14.3.6.1. Relaciones uno-a-muchos

El generador de la administración se ocupa automáticamente de las relaciones de tablas de tipo 1-n. Como se muestra en la figura 14-1, la tabla `blog_comment` se relaciona con la tabla `blog_article` mediante el campo `article_id`. Si se utiliza el generador de administraciones para iniciar el módulo de la clase `Comment`, la acción `comment/edit` muestra automáticamente el campo `article_id` como una lista desplegable con los valores de los ID de todas las filas de datos de la tabla `blog_article` (la figura 14-9 también muestra una figura de esta relación).

Además, si se define un método `__toString()` en la clase `Article`, la lista desplegable puede mostrar otro texto para cada opción en vez del valor de la clave primaria de la fila.

Si se quiere mostrar la lista de comentarios relacionados con un artículo en el módulo `article` (relación 1-n) se debe modificar el módulo y utilizar un campo parcial.

14.3.6.2. Relaciones muchos-a-muchos

Symfony también se encarga de las relaciones n-n, pero como estas relaciones no se pueden definir en el esquema, es necesario añadir un par de opciones al archivo `generator.yml`.

Las relaciones muchos-a-muchos requieren una tabla intermedia. Si por ejemplo existe una relación n-n entre la tabla `blog_article` y la tabla `blog_author` (un artículo puede estar escrito por más de un autor y un mismo autor puede escribir varios artículos), la base de datos debe contener una tabla llamada `blog_article_author` o algo parecido, como se muestra en la figura Figure 14-20.

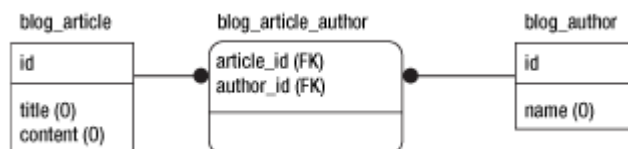


Figura 14.17. Uso de una tabla intermedia en las relaciones muchos-a-muchos

El modelo en este caso dispone de una clase llamada `ArticleAuthor`, que es el único dato que necesita el generador de la administración y que se indica en la opción `through_class` del campo adecuado.

En un módulo generado automáticamente a partir de la clase `Article`, se puede añadir un nuevo campo para crear una asociación n-n con la clase `Author` mediante las opciones del archivo `generator.yml` mostrado en el listado 14-30.

Listado 14-30 - Definiendo las relaciones muchos-a-muchos mediante la opción `through_class`

```

edit:
  fields:
    article_author: { type: admin_double_list, params:
through_class=ArticleAuthor }
  
```

Este nuevo campo gestiona las relaciones entre los objetos existentes, por lo que no es suficiente con mostrar una lista desplegable. Este tipo de relaciones exige un tipo especial de campo para introducir los datos. Symfony incluye 3 tipos de campos especiales para relacionar los elementos de las 2 listas (que se muestran en la figura 14-21).

- El tipo `admin_double_list` es un conjunto de 2 listas desplegables expandidas, además de los botones que permiten pasar elementos de la primera lista (elementos disponibles) a la segunda lista (elementos seleccionados).
- El tipo `admin_select_list` es una lista desplegable expandida que permite seleccionar más de 1 elemento cada vez.
- El tipo `admin_check_list` es una lista de elementos checkbox seleccionables.



Figura 14.18. Tipos de campos especiales disponibles para la gestión de las relaciones muchos-a-muchos

14.3.7. Añadiendo nuevas interacciones

Los módulos de administración permiten a los usuarios realizar las operaciones CRUD habituales, aunque también es posible añadir otras interacciones diferentes o restringir las interacciones disponibles en una vista. La configuración que se muestra en el listado 14-31 habilita todas las operaciones CRUD habituales para el módulo `article`.

Listado 14-31 - Definiendo las interacciones de cada vista, en `backend/modules/article/config/generator.yml`

```
list:
  title:          List of Articles
  object_actions:
    _edit:        ~
    _delete:      ~
  batch_actions:
    _delete:      ~
  actions:
    _create:      ~

edit:
  title:          Body of article %%title%%
  actions:
    _list:        ~
    _save:        ~
    _save_and_add: ~
    _delete:      ~
```

En la vista de tipo `list`, existen tres opciones relacionadas con las acciones: la lista de las acciones disponibles para todos los objetos (`object_actions`), la lista de opciones disponibles para una selección de objetos (`batch_actions`) y la lista de acciones disponibles para la página entera (`actions`). La lista de interacciones definidas en el listado 14-31 producen el resultado de la figura 14-22. Cada fila de datos muestra un botón para modificar la información y un botón para eliminar ese registro, además de un *checkbox* en cada fila para poder borrar varios elementos seleccionados. Al final de la lista se muestra un botón para crear nuevos elementos.

List of Articles




Id	Title	nb comments	Created at	Actions
1	Welcome to the symfony weblog!	3	December 1, 2006 1:17 PM	 
2	Life is beautiful	3	December 1, 2006 1:17 PM	 
2 results				
				 create

Figura 14.19. Interacciones de la vista "list"

En la vista `edit`, como sólo se modifica un registro de datos cada vez, solamente se define un conjunto de acciones (en `actions`). Las interacciones definidas en el listado 14-31 se muestran con el aspecto de la figura 14-23. Tanto la acción `save` (guardar) como la acción `save_and_add`

(guardar_y_añadir) guardan los cambios realizados en los datos. La única diferencia es que la acción `save` vuelve a mostrar la vista `edit` con los nuevos datos y la acción `save_and_add` muestra la vista `edit` con un formulario vacío para añadir otro elemento. Por tanto, la acción `save_and_add` es un atajo muy útil cuando se están añadiendo varios elementos de forma consecutiva. El botón de la acción `delete` (borrar) se encuentra lo suficientemente alejado de los otros 2 botones como para que no sea pulsado por accidente.

Los nombres de las interacciones que empiezan con un guión bajo (`_`) son reconocidos por Symfony y por tanto, utilizan el mismo icono y realizan la misma acción que las interacciones por defecto. El generador de la administración es capaz de reconocer las acciones `_edit`, `_delete`, `_create`, `_list`, `_save`, `_save_and_add` y `_create`.

Body of article Life is beautiful

Content:

The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?

list

save

save and add

delete

Figura 14.20. Interacciones de la vista "edit"







También es posible definir interacciones propias, para lo que se debe especificar un nombre que no empiece por guión bajo y una acción del módulo actual, tal y como se muestra en el listado 14-32.

Listado 14-32 - Definiendo una interacción propia

```
list:
  title:          List of Articles
  object_actions:
    _edit:        -
    _delete:      -
    addcomment:   { name: Add a comment, action: addComment, icon: backend/
addcomment.png }
```

Ahora, cada artículo que aparece en el listado muestra un botón con la imagen `addcomment.png`, tal y como se muestra en la figura 14-24. Al pinchar sobre ese botón, se ejecuta la acción `addComment` del módulo actual. La clave primaria del objeto relacionado se pasa automáticamente a los parámetros de la petición que se produce.

List of Articles

Id	Title	nb comments	Created at	Actions
1	Welcome to the symfony weblog!	3	December 1, 2006 1:17 PM	  
2	Life is beautiful	3	December 1, 2006 1:17 PM	  
2 results				

Add a comment

+ create

Figura 14.21. Interacciones propias en la vista "list"

La acción `addComment` puede ser tan sencilla como la que muestra el listado 14-33.

Listado 14-33 - Acción para una interacción propia, en `actions/actions.class.php`

```
public function executeAddComment($peticion)
{
    $comment = new Comment();
    $comment->setArticleId($peticion->getParameter('id'));
    $comment->save();

    $this->redirect('comment/edit?id='.$comment->getId());
}
```

Las acciones de tipo batch reciben un array con todas las claves primarias de las filas seleccionadas mediante el parámetro `sf_admin_batch_selection` de la petición.

Por último, si se quieren eliminar todas las acciones para una determinada vista, se utiliza una lista vacía como la del listado 14-34.

Listado 14-34 - Eliminando todas las acciones en la vista `list`

```
list:
  title:      List of Articles
  actions:    {}
```

14.3.8. Validación de formularios

Si se observa el código de la plantilla `_edit_form.php` generada, que se encuentra en el directorio `cache/` del proyecto, se puede ver que los campos del formulario utilizan un nombrado especial. En la vista `edit` generada, los nombres de los campos del formulario se definen como la concatenación del nombre del módulo (utilizando guiones bajos) y el nombre del campo encerrado entre corchetes.

Si la vista `edit` de la clase `Article` tiene un campo llamado `title`, la plantilla será similar a la del listado 14-35 y el campo se identifica como `article[title]`.

Listado 14-35 - Sintaxis de los nombres generados para los campos

```
// generator.yml
generator:
  class:      sfPropelAdminGenerator
```

```

    param:
      model_class:      Article
      theme:            default
      edit:
        display: [title]
    // Plantilla _edit_form.php generada
    <?php echo object_input_tag($article, 'getTitle', array('control_name' =>
'<article[title]>')) ?>
    // Código HTML generado
    <input type="text" name="article[title]" id="article_title" value="My Title" />

```

El uso de estos nombres de campos facilita el procesamiento de los formularios. Sin embargo, como se explica en el Capítulo 10, complica un poco la configuración del validador, por lo que se deben cambiar los corchetes [] por llaves { } en la clave `fields` del archivo de validación. Además, cuando se utiliza el nombre de un campo como parámetro del validador, se debe utilizar el nombre tal y como aparece en el código HTML (es decir, con los corchetes, pero entre comillas). El listado 14-36 muestra un ejemplo de la sintaxis especial que se debe utilizar para el validador de los formularios generados automáticamente.

Listado 14-36 - Sintaxis del archivo de validación para los formularios generados automáticamente

```

## Se reemplazan Los corchetes por comillas en la lista de campos
fields:
  article{title}:
    required:
      msg: You must provide a title
    ## Para Los parámetros del validador se utiliza el nombre original del campo
entre comillas
    sfCompareValidator:
      check:      "user[newpassword]"
      compare_error: The password confirmation does not match the password.

```

14.3.9. Restringiendo las acciones del usuario mediante credenciales

Los campos y las interacciones disponibles en un módulo de administración pueden variar en función de las credenciales del usuario conectado (el Capítulo 6 describe las opciones de seguridad de Symfony).

Los campos definidos en el generador pueden incluir una opción `credentials` para restringir su acceso solamente a los usuarios con la credencial adecuada. Esta característica se puede utilizar tanto en la vista `list` como en la vista `edit`. Además, el generador puede ocultar algunas interacciones en función de la credenciales del usuario. El listado 14-37 muestra estas opciones.

Listado 14-37 - Utilizando credenciales en `generator.yml`

```

## La columna id solamente se muestra para Los usuarios con la credencial "admin"
list:
  title:      List of Articles
  layout:     tabular
  display:    [id, =title, content, nb_comments]
  fields:
    id:       { credentials: [admin] }

```

```

## La interacción "addcomment" se restringe a Los usuarios con la credencial "admin"
list:
  title:          List of Articles
  object_actions:
    _edit:        -
    _delete:      -
    addcomment:   { credentials: [admin], name: Add a comment, action:
addComment, icon: backend/addcomment.png }

```

14.4. Modificando el aspecto de los módulos generados

La presentación de los módulos generados se puede modificar completamente para integrarlo con cualquier otro estilo gráfico. Los cambios no solo se pueden realizar mediante una hoja de estilos, sino que es posible redefinir las plantillas por defecto.

14.4.1. Utilizando una hoja de estilos propia

Como el código HTML generado tiene un contenido bien estructurado, es posible modificar fácilmente su aspecto.

Mediante la opción `css` de la configuración del generador es posible definir la hoja de estilos alternativa que se utiliza en el módulo de administración, como se muestra en el listado 14-38.

Listado 14-38 - Utilizando una hoja de estilos propia en vez de la de por defecto

```

generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default
    css:          mystylesheet

```

Además, también es posible utilizar las opciones habituales del archivo `view.yml` del módulo para redefinir los estilos utilizados en cada vista.

14.4.2. Creando una cabecera y un pie propios

Las vistas `list` y `edit` incluyen por defecto elementos parciales para la cabecera y el pie de página. Como no existen por defecto elementos parciales en el directorio `templates/` del módulo de administración, solamente es necesario crearlos con los siguientes nombres para que se incluyan de forma automática:

```

_list_header.php
_list_footer.php
_edit_header.php
_edit_footer.php

```

Si se quiere añadir por ejemplo una cabecera propia en la vista `article/edit`, se crea un archivo llamado `_edit_header.php` como el que muestra el listado 14-39. No es necesario realizar más configuraciones para que se incluya automáticamente.

Listado 14-39 - Ejemplo de elemento parcial para la cabecera de la vista edit, en `modules/articles/templates/_edit_header.php`

```
<?php if ($article->getNbComments() > 0): ?>
    <h2>This article has <?php echo $article->getNbComments() ?> comments.</h2>
<?php endif; ?>
```

Debe tenerse en cuenta que un elemento parcial de la vista edit siempre tiene acceso al objeto al que hace referencia mediante una variable con el mismo nombre que la clase y que un elemento parcial de la vista list tiene acceso al paginador actual mediante la variable `$pager`.

Las acciones del módulo de administración pueden recibir parámetros propios mediante la opción `query_string` del *helper* `link_to()`. Por ejemplo, para extender el elemento parcial `_edit_header` anterior con un enlace a los comentarios del artículo, se utiliza el siguiente código:

```
<?php if ($article->getNbComments() > 0): ?>
    <h2>This article has <?php echo link_to($article->getNbComments(). ' comments',
'comment/list', array('query_string' =>
'filter=filter&filters%5Barticle_id%5D= '.$article->getId())) ?></h2>
<?php endif; ?>
```

El valor que se pasa a la opción `query_string` es una versión codificada del siguiente valor más fácil de leer:

```
'filter=filter&filters[article_id]='.$article->getId()
```

Se filtran los comentarios que se muestran a solamente los que estén relacionados con `$article`. Si se utiliza la opción `query_string`, es posible especificar el orden en el que se ordenan los datos y los filtros utilizados para mostrar una vista de tipo list. Esta opción también es útil para las interacciones propias.

14.4.3. Modificando el tema

Existen otros elementos parciales en el directorio `templates/` del módulo que heredan del framework y que se pueden redefinir para adaptarse a las necesidades de cada proyecto.

Las plantillas del generador están divididas en pequeñas partes que se pueden redefinir de forma independiente, al igual que se pueden modificar las acciones una a una.

No obstante, si se quieren redefinir todos los elementos parciales para varios módulos, lo mejor es crear un tema que se pueda reutilizar. Un tema es un conjunto completo de plantillas y acciones que se pueden utilizar en un módulo de administración si así se indica en el archivo `generator.yml`. En el tema por defecto, Symfony utiliza los archivos definidos en `$sf_symfony_lib_dir/plugins/sfPropelPlugin/data/generator/sfPropelAdmin/default/`.

Los archivos de los temas tienen que guardarse en el directorio `data/generator/sfPropelAdmin/[nombre_tema]/template/` del proyecto, y la mejor forma de crear un nuevo tema es copiando los archivos del tema por defecto (que se encuentran en el directorio `$sf_symfony_lib_dir/plugins/sfPropelPlugin/data/generator/sfPropelAdmin/default/template/`). De esta forma, es fácil asegurarse de que el tema propio contiene todos los archivos requeridos:


```
// Elementos parciales, en [nombre_tema]/template/templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php

// Acciones, en [nombre_tema]/template/actions/actions.class.php
processFilters() // Procesa los filtros de la petición
addFiltersCriteria() // Añade un filtro al objeto Criteria
processSort()
addSortCriteria()
```

Se debe tener en cuenta que los archivos de las plantillas son en realidad "plantillas de plantillas", es decir, archivos PHP que se procesan mediante una herramienta especial para generar las plantillas en función de las opciones del generador (este proceso se conoce como la fase de compilación). Como las plantillas generadas deben contener código PHP que se ejecuta cuando se accede a estas plantillas, los archivos que son "plantillas de plantillas" tienen que utilizar una sintaxis alternativa para que el código PHP final no se ejecute durante el proceso de compilación de las plantillas. El listado 14-40 muestra un trozo de una de las "plantillas de plantillas" de Symfony.

Listado 14-40 - Sintaxis de las plantillas de plantillas

```
<?php foreach ($this->getPrimaryKey() as $pk): ?>
[?php echo object_input_hidden_tag($<?php echo $this->getSingularName()
?>,'get<?php echo $pk->getPhpName() ?>') ?]
<?php endforeach; ?>
```

En el listado anterior, el código PHP indicado mediante `<?` se ejecuta durante la compilación, mientras que el código indicado mediante `[?` se ejecuta solamente durante la ejecución final de la plantilla generada. El generador de plantillas reemplaza las etiquetas `[?` en etiquetas `<?`, por lo que la plantilla resultante es la siguiente:

```
<?php echo object_input_hidden_tag($article, 'getId') ?>
```

Trabajar con las "plantillas de plantillas" es bastante complicado, por lo que el mejor consejo para crear un tema propio es comenzarlo a partir del tema default, modificarlo poco a poco y probar los cambios continuamente.

Sugerencia También es posible encapsular un tema completo para el generador en un plugin, con lo que el tema es más fácil de reutilizar y más fácil de instalar en diferentes aplicaciones. El Capítulo 17 incluye más información.

El generador de la parte de administración utiliza una serie de componentes internos de Symfony que automatizan la creación de acciones y plantillas en la cache, el uso de temas y el procesamiento de las "plantillas de plantillas".

De esta forma, Symfony proporciona todas las herramientas para crear tu propio generador, que puede ser similar a los existentes o ser completamente diferente. La generación automática de un módulo se gestiona mediante el método `generate()` de la clase `sfGeneratorManager`. Por ejemplo, para generar una administración, Symfony realiza internamente la siguiente llamada a este método:

```
$generator_manager = new sfGeneratorManager();  
$data = $generator_manager->generate('sfPropelAdminGenerator', $parameters);
```

Si se quiere construir un generador propio, es conveniente mirar la documentación de la API de las clases `sfGeneratorManager` y `sfGenerator`, y utilizar las clases `sfAdminGenerator` y `sfCRUDGenerator` como ejemplo.

14.5. Resumen

Para generar automáticamente los módulos de una aplicación de gestión, lo principal es disponer de un esquema y un modelo de objetos bien definidos. La modificación de los módulos de una administración generada automáticamente se realiza mediante los archivos de configuración.

El archivo `generator.yml` es la clave de los módulos generados automáticamente. Permite modificar completamente el contenido, las opciones y el aspecto gráfico de las vistas `list` y `edit`. Sin utilizar ni una sola línea de código PHP y solamente mediante opciones en un archivo de configuración YAML es posible añadir títulos a los campos de formulario, mensajes de ayuda, filtros, configurar la ordenación de los datos, definir el tamaño de los listados, el tipo de campos empleados en los formularios, las relaciones con claves externas, las interacciones propias y el uso de credenciales.

Si el generador de las administraciones no permite incluir las características requeridas por el proyecto, se pueden utilizar elementos parciales y se pueden redefinir las acciones para conseguir la máxima flexibilidad. Además, se pueden reutilizar todas las adaptaciones realizadas al generador de administraciones mediante el uso de los temas.

Capítulo 15. Pruebas unitarias y funcionales

La automatización de pruebas (*automated tests*) es uno de los mayores avances en la programación desde la invención de la orientación a objetos. Concretamente en el desarrollo de las aplicaciones web, las pruebas aseguran la calidad de la aplicación incluso cuando el desarrollo de nuevas versiones es muy activo. En este capítulo se introducen todas las herramientas y utilidades que proporciona Symfony para automatizar las pruebas.

15.1. Automatización de pruebas

Cualquier programador con experiencia en el desarrollo de aplicaciones web conoce de sobra el esfuerzo que supone probar correctamente la aplicación. Crear casos de prueba, ejecutarlos y analizar sus resultados es una tarea tediosa. Además, es habitual que los requisitos de la aplicación varíen constantemente, con el consiguiente aumento del número de versiones de la aplicación y la refactorización continua del código. En este contexto, es muy probable que aparezcan nuevos errores.

Este es el motivo por el que la automatización de pruebas es una recomendación, aunque no una obligación, útil para crear un entorno de desarrollo satisfactorio. Los conjuntos de casos de prueba garantizan que la aplicación hace lo que se supone que debe hacer. Incluso cuando el código interno de la aplicación cambia constantemente, las pruebas automatizadas permiten garantizar que los cambios no introducen incompatibilidades en el funcionamiento de la aplicación. Además, este tipo de pruebas obligan a los programadores a crear pruebas en un formato estandarizado y muy rígido que pueda ser procesado por un framework de pruebas.

En ocasiones, las pruebas automatizadas pueden reemplazar la documentación técnica de la aplicación, ya que ilustran de forma clara el funcionamiento de la aplicación. Un buen conjunto de pruebas muestra la salida que produce la aplicación para una serie de entradas de prueba, por lo que es suficiente para entender el propósito de cada método.

Symfony aplica este principio a su propio código. El código interno del framework se valida mediante pruebas automáticas. Estas pruebas unitarias y funcionales no se incluyen en la distribución estándar de Symfony, pero se pueden descargar directamente desde el repositorio de Subversion y se pueden acceder online en <http://trac.symfony-project.com/browser/branches/1.1/test>

15.1.1. Pruebas unitarias y funcionales

Las pruebas unitarias aseguran que un único componente de la aplicación produce una salida correcta para una determinada entrada. Este tipo de pruebas validan la forma en la que las funciones y métodos trabajan en cada caso particular. Las pruebas unitarias se encargan de un único caso cada vez, lo que significa que un único método puede necesitar varias pruebas unitarias si su funcionamiento varía en función del contexto.

Las pruebas funcionales no solo validan la transformación de una entrada en una salida, sino que validan una característica completa. Un sistema de cache por ejemplo solamente puede ser validado por una prueba funcional, ya que comprende más de 1 solo paso: la primera vez que se solicita una página, se produce su código; la segunda vez, se obtiene directamente de la cache. De modo que las pruebas funcionales validan procesos y requieren de un escenario. En Symfony, se deberían crear pruebas funcionales para todas las acciones.

Para las interacciones más complejas, estos 2 tipos de pruebas no son suficientes. Las interacciones de Ajax, por ejemplo, requieren de un navegador web que ejecute código JavaScript, por lo que es necesaria una herramienta externa para la automatización de las pruebas. Además, los efectos visuales solamente pueden ser validados por una persona.

Si las pruebas automatizadas van a validar una aplicación compleja, probablemente sea necesario el uso combinado de estos 3 tipos de pruebas. Como recomendación final, es aconsejable crear pruebas sencillas y fáciles de entender.

Nota Las pruebas automatizadas comparan un resultado con la salida esperada para ese método. En otras palabras, evalúan "asertos" (del inglés, "*assertions*", que son expresiones del tipo `$a == 2`. El valor de salida de un aserto es `true` o `false`, lo que determina si la prueba tiene éxito o falla. La palabra "aserto" es de uso común en las técnicas de automatización de pruebas.

15.1.2. Desarrollo basado en pruebas

La metodología conocida como TDD o "desarrollo basado en pruebas" ("*test-driven development*") establece que las pruebas se escriben antes que el código de la aplicación. Crear las pruebas antes que el código, ayuda a pensar y centrarse en el funcionamiento de un método antes de programarlo. Se trata de una buena práctica que también recomiendan otras metodologías como XP ("*Extreme Programming*"). Además, es un hecho innegable que si no se escriben las pruebas antes, se acaba sin escribirlas nunca.

En el siguiente ejemplo se supone que se quiere desarrollar una función elimine los caracteres problemáticos de una cadena de texto. La función elimina todos los espacios en blanco del principio y del final de la cadena; además, reemplaza todos los caracteres que no son alfanuméricos por guiones bajos y transforma todas las mayúsculas en minúsculas. En el desarrollo basado en pruebas, en primer lugar se piensa en todos los posibles casos de funcionamiento de este método y se elabora una serie de entradas de prueba junto con el resultado esperado para cada una, como se muestra en la tabla 15-1.

Tabla 15-1 - Lista de casos de prueba para la función que elimina caracteres problemáticos

Dato de entrada	Resultado esperado
" valor "	"valor"
"valor otrovalor"	"valor_otrovalor"
"-)valor:..=otrovalor?"	"__valor__otrovalor_"
"OtroValor"	"otrovalor"

"Un valor y otro valor!"	"_un_valor_y_otro_valor_"
--------------------------	---------------------------

A continuación, se crearían las pruebas unitarias, se ejecutarían y todas fallarían. Después, se escribe el código necesario para realizar correctamente el primer caso y se vuelven a pasar todas las pruebas. En esta ocasión, la primera prueba no fallaría. Así se seguiría desarrollando el código del método completo hasta que todas las pruebas se pasen correctamente.

Una aplicación desarrollada con la metodología basada en pruebas, acaba teniendo tanto código para pruebas como código para aplicación. Por tanto, las pruebas deberían ser sencillas para no perder el tiempo arreglando los problemas con el código de las pruebas.

Nota Refactorizar el código de un método puede crear errores que antes no existían. Esta es otra razón por la que es una buena idea pasar todas las pruebas creadas antes de instalar una nueva versión de la aplicación en el servidor de producción. Esta técnica se conoce como *"regression testing"*.

15.1.3. El framework de pruebas Lime

En el ámbito de PHP existen muchos frameworks para crear pruebas unitarias, siendo los más conocidos PHPUnit y SimpleTest. Symfony incluye su propio framework llamado Lime. Se basa en la librería `Test::More` de Perl y es compatible con TAP, lo que significa que los resultados de las pruebas se muestran con el formato definido en el *"Test Anything Protocol"*, creado para facilitar la lectura de los resultados de las pruebas.

Lime proporciona el soporte para las pruebas unitarias, es más eficiente que otros frameworks de pruebas de PHP y tiene las siguientes ventajas:

- Ejecuta los archivos de prueba en un entorno independiente para evitar interferencias entre las diferentes pruebas. No todos los frameworks de pruebas garantizan un entorno de ejecución "limpio" para cada prueba.
- Las pruebas de Lime son fáciles de leer y sus resultados también lo son. En los sistemas operativos que lo soportan, los resultados de Lime utilizan diferentes colores para mostrar de forma clara la información más importante.
- Symfony utiliza Lime para sus propias pruebas y su *"regression testing"*, por lo que el código fuente de Symfony incluye muchos ejemplos reales de pruebas unitarias y funcionales.
- El núcleo de Lime se valida mediante pruebas unitarias.
- Está escrito con PHP, es muy rápido y está bien diseñado internamente. Consta únicamente de un archivo, llamado `lime.php`, y no tiene ninguna dependencia.

Las pruebas que se muestran en las secciones siguientes utilizan la sintaxis de Lime, por lo que funcionan directamente en cualquier instalación de Symfony.

Nota Las pruebas unitarias y funcionales no están pensadas para lanzarlas en un servidor de producción. Se trata de herramientas para el programador, por lo que solamente deberían ejecutarse en la máquina de desarrollo del programador y no en un servidor de producción.

15.2. Pruebas unitarias

Las pruebas unitarias de Symfony son archivos PHP normales cuyo nombre termina en `Test.php` y que se encuentran en el directorio `test/unit/` de la aplicación. Su sintaxis es sencilla y fácil de leer.

15.2.1. ¿Qué aspecto tienen las pruebas unitarias?

El listado 15-1 muestra un conjunto típico de pruebas unitarias para la función `strtolower()`. En primer lugar, se instancia el objeto `lime_test` (todavía no hace falta que te preocupes de sus parámetros). Cada prueba unitaria consiste en una llamada a un método de la instancia de `lime_test`. El último parámetro de estos métodos siempre es una cadena de texto opcional que se utiliza como resultado del método.

Listado 15-1 - Archivo de ejemplo de prueba unitaria, en `test/unit/strtolowerTest.php`

```
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');
require_once(dirname(__FILE__).'../lib/strtolower.php');

$t = new lime_test(7, new lime_output_color());

// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string',
    'strtolower() returns a string');
$t->is(strtolower('FOO'), 'foo',
    'strtolower() transforms the input to lowercase');
$t->is(strtolower('foo'), 'foo',
    'strtolower() leaves lowercase characters unchanged');
$t->is(strtolower('12#@~'), '12#@~',
    'strtolower() leaves non alphabetical characters unchanged');
$t->is(strtolower('FOO BAR'), 'foo bar',
    'strtolower() leaves blanks alone');
$t->is(strtolower('FoO bAr'), 'foo bar',
    'strtolower() deals with mixed case input');
$t->is(strtolower(''), 'foo',
    'strtolower() transforms empty strings into foo');
```

Para ejecutar el conjunto de pruebas, se utiliza la tarea `test:unit` desde la línea de comandos. El resultado de esta tarea en la línea de comandos es muy explícito, lo que permite localizar fácilmente las pruebas que han fallado y las que se han ejecutado correctamente. El listado 15-2 muestra el resultado del ejemplo anterior.

Listado 15-2 - Ejecutando una prueba unitaria desde la línea de comandos

```
> php symfony test:unit strtolower

1..7
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
```

```

ok 3 - strtolower() leaves lowercase characters unchanged
ok 4 - strtolower() leaves non alphabetical characters unchanged
ok 5 - strtolower() leaves blanks alone
ok 6 - strtolower() deals with mixed case input
not ok 7 - strtolower() transforms empty strings into foo
#     Failed test (.\\batch\\test.php at line 21)
#         got: ''
#     expected: 'foo'
# Looks like you failed 1 tests of 7.

```

Sugerencia La instrucción incluye al principio del listado 15-1 es opcional, pero hace que el archivo de la prueba sea un script de PHP independiente, es decir, que se puede ejecutar sin utilizar la línea de comandos de Symfony, mediante `php test/unit/strtolowerTest.php`.

15.2.2. Métodos para las pruebas unitarias

El objeto `lime_test` dispone de un gran número de métodos para las pruebas, como se muestra en la figura 15-2.

Tabla 15-2 - Métodos del objeto `lime_test` para las pruebas unitarias

Método	Descripción
<code>diag(\$mensaje)</code>	Muestra un comentario, pero no ejecuta ninguna prueba
<code>ok(\$prueba, \$mensaje)</code>	Si la condición que se indica es <code>true</code> , la prueba tiene éxito
<code>is(\$valor1, \$valor2, \$mensaje)</code>	Compara 2 valores y la prueba pasa si los 2 son iguales (==)
<code>isnt(\$valor1, \$valor2, \$mensaje)</code>	Compara 2 valores y la prueba pasa si no son iguales
<code>like(\$cadena, \$expresionRegular, \$mensaje)</code>	Prueba que una cadena cumpla con el patrón de una expresión regular
<code>unlike(\$cadena, \$expresionRegular, \$mensaje)</code>	Prueba que una cadena no cumpla con el patrón de una expresión regular
<code>cmp_ok(\$valor1, \$operador, \$valor2, \$mensaje)</code>	Compara 2 valores mediante el operador que se indica
<code>isa_ok(\$variable, \$tipo, \$mensaje)</code>	Comprueba si la variable que se le pasa es del tipo que se indica
<code>isa_ok(\$objeto, \$clase, \$mensaje)</code>	Comprueba si el objeto que se le pasa es de la clase que se indica
<code>can_ok(\$objeto, \$metodo, \$mensaje)</code>	Comprueba si el objeto que se le pasa dispone del método que se indica
<code>is_deeply(\$array1, \$array2, \$mensaje)</code>	Comprueba que 2 arrays tengan los mismos valores
<code>include_ok(\$archivo, \$mensaje)</code>	Valida que un archivo existe y que ha sido incluido correctamente
<code>fail()</code>	Provoca que la prueba siempre falle (es útil para las excepciones)
<code>pass()</code>	Provoca que la prueba siempre se pase (es útil para las excepciones)

<code>skip(\$mensaje, \$numeroPruebas)</code>	Cuenta como si fueran <code>\$numeroPruebas</code> pruebas (es útil para las pruebas condicionales)
<code>todo()</code>	Cuenta como si fuera 1 prueba (es útil para las pruebas que todavía no se han escrito)

La sintaxis es tan clara que prácticamente se explica por sí sola. Casi todos los métodos permiten indicar un mensaje como último parámetro. Este mensaje es el que se muestra como resultado de la prueba cuando esta tiene éxito. La mejor manera de aprender a utilizar estos métodos es utilizarlos, así que es importante el código del listado 15-3, que utiliza todos los métodos.

Listado 15-3 - Probando los métodos del objeto `lime_test`, en `test/unit/ejemploTest.php`

```
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');

// Funciones y objetos vacíos para las pruebas
class miObjeto
{
    public function miMetodo()
    {
    }
}

function lanza_una_excepcion()
{
    throw new Exception('excepción lanzada');
}

// Inicializar el objeto de pruebas
$t = new lime_test(16, new lime_output_color());

$t->diag('hola mundo');
$t->ok(1 == '1', 'el operador de igualdad ignora el tipo de la variable');
$t->is(1, '1', 'las cadenas se convierten en números para realizar la comparación');
$t->isnt(0, 1, '0 y 1 no son lo mismo');
$t->like('prueba01', '/prueba\d+/', 'prueba01 sigue el patrón para numerar las pruebas');
$t->unlike('pruebas01', '/prueba\d+/', 'pruebas01 no sigue el patrón');
$t->cmp_ok(1, '<', 2, '1 es inferior a 2');
$t->cmp_ok(1, '!=', true, '1 y true no son exactamente lo mismo');
$t->isa_ok('valor', 'string', '\'\valor\' es una cadena de texto');
$t->isa_ok(new miObjeto(), 'miObjeto', 'new crea un objeto de la clase correcta');
$t->can_ok(new miObjeto(), 'miMetodo', 'los objetos de la clase miObjeto tienen un método llamado miMetood');
$array1 = array(1, 2, array(1 => 'valor', 'a' => '4'));
$t->is_deeply($array1, array(1, 2, array(1 => 'valor', 'a' => '4')),
    'el primer array y el segundo array son iguales');
$t->include_ok('./nombreArchivo.php', 'el archivo nombreArchivo.php ha sido incluido correctamente');

try
{
    lanza_una_excepcion();
}
```



```

    $t->fail('no debería ejecutarse ningún código después de lanzarse la excepción');
}
catch (Exception $e)
{
    $t->pass('la excepción ha sido capturada correctamente');
}

if (!isset($variable))
{
    $t->skip('saltamos una prueba para mantener el contador de pruebas correcto para la
condición', 1);
}
else
{
    $t->ok($variable, 'valor');
}

$t->todo('la última prueba que falta');

```

Las pruebas unitarias de Symfony incluyen muchos más ejemplos de uso de todos estos métodos.

Sugerencia Puede que sea confuso el uso de `is()` en vez de `ok()` en el ejemplo anterior. La razón es que el mensaje de error que muestra `is()` es mucho más explícito, ya que muestra los 2 argumentos de la prueba, mientras que `ok()` simplemente dice que la prueba ha fallado.

15.2.3. Parámetros para las pruebas

En la inicialización del objeto `lime_test` se indica como primer parámetro el número de pruebas que se van a ejecutar. Si el número de pruebas realmente realizadas no coincide con este valor, la salida producida por Lime muestra un aviso. El conjunto de pruebas del listado 15-3 producen la salida del listado 15-4. Como en la inicialización se indica que se deben ejecutar 16 pruebas y realmente solo se han realizado 15, en la salida se muestra un mensaje de aviso.

Listado 15-4 - El contador de pruebas realizadas permite planificar las pruebas

```

> php symfony test:unit ejemplo

1..16
# hola mundo
ok 1 - el operador de igualdad ignora el tipo de la variable
ok 2 - las cadenas se convierten en números para realizar la comparación
ok 3 - 0 y 1 no son lo mismo
ok 4 - prueba01 sigue el patrón para numerar las pruebas
ok 5 - pruebas01 no sigue el patrón
ok 6 - 1 es inferior a 2
ok 7 - 1 y true no son exactamente lo mismo
ok 8 - 'valor' es una cadena de texto
ok 9 - new crea un objeto de la clase correcta
ok 10 - los objetos de la clase miObjeto tienen un método llamado miMetood
ok 11 - el primer array y el segundo array son iguales
not ok 12 - el archivo nombreArchivo.php ha sido incluido correctamente
#     Failed test (.\test\unit\ejemploTest.php at line 35)
#     Tried to include './nombreArchivo.php'

```

```

ok 13 - la excepción ha sido capturada correctamente
ok 14 # SKIP saltamos una prueba para mantener el contador de pruebas correcto para la
condición
ok 15 # TODO la última prueba que falta
# Looks like you planned 16 tests but only ran 15.
# Looks like you failed 1 tests of 16.

```

El método `diag()` no cuenta como una prueba. Se utiliza para mostrar mensajes, de forma que la salida por pantalla esté más organizada y sea más fácil de leer. Por otra parte, los métodos `todo()` y `skip()` cuentan como si fueran pruebas reales. La combinación `pass()/fail()` dentro de un bloque `try/catch` cuenta como una sola prueba.

Una estrategia de pruebas bien planificada requiere que se indique el número esperado de pruebas. Indicar este número es una buena forma de validar los propios archivos de pruebas, sobre todo en los casos más complicados en los que algunas pruebas se ejecutan dentro de condiciones y/o excepciones. Además, si la prueba falla en cualquier punto, es muy fácil de verlo porque el número de pruebas realizadas no coincide con el número de pruebas esperadas.

El segundo parámetro del constructor del objeto `lime_test` indica el objeto que se utiliza para mostrar los resultados. Se trata de un objeto que extiende la clase `lime_output`. La mayoría de las veces, como las pruebas se realizan en una interfaz de comandos, la salida se construye mediante el objeto `lime_output_color`, que muestra la salida coloreada en los sistemas que lo permiten.

15.2.4. La tarea `test:unit`

La tarea `test:unit`, que se utiliza para ejecutar las pruebas unitarias desde la línea de comandos, admite como argumento una serie de nombres de pruebas o un patrón de nombre de archivos. El listado 15-5 muestra los detalles.

Listado 15-5 - Ejecutando las pruebas unitarias

```

// Estructura del directorio de pruebas
test/
  unit/
    miFuncionalTest.php
    miSegundoFuncionalTest.php
    otro/
      nombreTest.php
> php symfony test:unit miFuncional          ## Ejecutar miFuncionalTest.php
> php symfony test:unit miFuncional miSegundoFuncional ## Ejecuta las 2 pruebas
> php symfony test:unit 'otro/*'             ## Ejecuta nombreTest.php
> php symfony test:unit '*'                   ## Ejecuta todas las pruebas (de
forma recursiva)

```

15.2.5. Stubs, Fixtures y carga automática de clases

La carga automática de clases no funciona por defecto en las pruebas unitarias. Por tanto, todas las clases que se utilizan en una prueba se deben definir en el propio archivo de la prueba o se deben incluir como una dependencia externa. Este es el motivo por el que muchos archivos de pruebas empiezan con un grupo de instrucciones `include`, como se muestra en el listado 15-6.

Listado 15-6 - Incluyendo las clases de forma explícita en las pruebas unitarias

```

<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');

$t = new lime_test(7, new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
$t->is(sfToolkit::isPathAbsolute('/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('C:\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('d:/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('..\\test'), false,
    'isPathAbsolute() returns false if path is relative');

```

En las pruebas unitarias, no solo se debe instanciar el objeto que se está probando, sino también el objeto del que depende. Como las pruebas unitarias deben ser autosuficientes, depender de otras clases puede provocar que más de una prueba falle si alguna clase no funciona correctamente. Además, crear objetos reales es una tarea costosa, tanto en número de líneas de código necesarias como en tiempo de ejecución. Debe tenerse en cuenta que la velocidad de ejecución es esencial para las pruebas unitarias, ya que los programadores en seguida se cansan de los procesos que son muy lentos.

Si se incluyen muchos scripts en una prueba unitaria, lo más útil es utilizar un sistema sencillo de carga automática de clases. Para ello, la clase `sfSimpleAutoload` (que se debe incluir manualmente) dispone del método `initSimpleAutoload()`, que utiliza como parámetro una ruta absoluta. Todas las clases que se encuentren bajo esa ruta, se cargarán automáticamente. Si por ejemplo se quieren cargar automáticamente todas las clases del directorio `$sf_symfony_lib_dir/util/`, se utilizan las siguientes instrucciones al principio del script de la prueba unitaria:

```

require_once($sf_symfony_lib_dir.'/autoload/sfSimpleAutoload.class.php');
$autoload = new sfSimpleAutoload();
$autoload->addDirectory($sf_symfony_lib_dir.'/util');
$autoload->register();

```

Sugerencia Los objetos Propel generados automáticamente dependen de muchísimas clases, por lo que en cuanto se quiere probar un objeto Propel es necesario utilizar la carga automática de clases. Además, para que funcione Propel es necesario incluir los archivos del directorio `plugins/sfPropelPlugin/lib/vendor/propel/` y también es preciso añadir Propel en el *include path* mediante la instrucción `set_include_path($configuration->getSymfonyLibDir().'/`

```
plugins/sfPropelPlugin/lib/
vendor'.PATH_SEPARATOR.$configuration->getRootDir().PATH_SEPARATOR.get_include_path()).
```

Otra técnica muy utilizada para evitar los problemas de la carga automática de clases es el uso de *stubs* o clases falsas. Un *stub* es una implementación alternativa de una clase en la que los métodos reales se sustituyen por datos simples especialmente preparados. De esta forma, se emula el comportamiento de la clase real y se reduce su tiempo de ejecución. Los casos típicos para utilizar *stubs* son las conexiones con bases de datos y las interfaces de los servicios web. En el listado 15-7, las pruebas unitarias para una API de un servicio de mapas utilizan la clase `WebService`. En vez de ejecutar el método `fetch()` real de la clase del servicio web, la prueba utiliza un *stub* que devuelve datos de prueba.

Listado 15-7 - Utilizando stubs en las pruebas unitarias

```
require_once(dirname(__FILE__).'../../lib/WebService.class.php');
require_once(dirname(__FILE__).'../../lib/MapAPI.class.php');

class testWebService extends WebService
{
    public static function fetch()
    {
        return file_get_contents(dirname(__FILE__).'../fixtures/data/servicio_web_falso.xml');
    }
}

$miMapa = new MapAPI();

$t = new lime_test(1, new lime_output_color());

$t->is($miMapa->getMapSize(testWebService::fetch(), 100));
```

Los datos de prueba pueden ser más complejos que una cadena de texto o la llamada a un método. Los datos de prueba complejos se suelen denominar "*fixtures*". Para mejorar el código de las pruebas unitarias, es recomendable mantener los *fixtures* en archivos independientes, sobre todo si se utilizan en más de una prueba. Además, Symfony es capaz de transformar un archivo YAML en un array mediante el método `sfYAML::load()`. De esta forma, en vez de escribir arrays PHP muy grandes, los datos para las pruebas se pueden guardar en archivos YAML, como en el listado 15-8.

Listado 15-8 - Usando archivos para los "fixtures" de las pruebas unitarias

```
// En fixtures.yml:
-
  input:  '/test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  '\\test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'C:\\test'
  output: true
```

```

    comment: isPathAbsolute() returns true if path is absolute
-
    input:  'd:/test'
    output: true
    comment: isPathAbsolute() returns true if path is absolute
-
    input:  'test'
    output: false
    comment: isPathAbsolute() returns false if path is relative
-
    input:  '../test'
    output: false
    comment: isPathAbsolute() returns false if path is relative
-
    input:  '..\\test'
    output: false
    comment: isPathAbsolute() returns false if path is relative
// En testTest.php
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');
require_once($sf_symfony_lib_dir.'/yaml/sfYaml.class.php');

$testCases = sfYaml::load(dirname(__FILE__).'../fixtures.yml');

$t = new lime_test(count($testCases), new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
foreach ($testCases as $case)
{
    $t->is(sfToolkit::isPathAbsolute($case['input']), $case['output'],$case['comment']);
}

```

15.3. Pruebas funcionales

Las pruebas funcionales validan partes de las aplicaciones. Estas pruebas simulan la navegación del usuario, realizan peticiones y comprueban los elementos de la respuesta, tal y como lo haría manualmente un usuario para validar que una determinada acción hace lo que se supone que tiene que hacer. En las pruebas funcionales, se ejecuta un escenario correspondiente a lo que se denomina un "caso de uso".

15.3.1. ¿Cómo son las pruebas funcionales?

Las pruebas funcionales se podrían realizar mediante un navegador en forma de texto y un montón de asertos definidos con expresiones regulares complejas, pero sería una pérdida de tiempo muy grande. Symfony dispone de un objeto especial, llamado `sfBrowser`, que actúa como un navegador que está accediendo a una aplicación Symfony, pero sin necesidad de utilizar un servidor web real (y sin la penalización de las conexiones HTTP). Este objeto permite el acceso directo a los objetos que forman cada petición (el objeto petición, el objeto sesión, el objeto contexto y el objeto respuesta). Symfony también dispone de una extensión de esta clase

llamada `sfTestBrowser`, que está especialmente diseñada para las pruebas funcionales y que tiene todas las características de `sfBrowser`, además de algunos métodos muy útiles para los asertos.

Una prueba funcional suele comenzar con la inicialización del objeto del navegador para pruebas. Este objeto permite realizar una petición a una acción de la aplicación y permite verificar que algunos elementos están presentes en la respuesta.

Por ejemplo, cada vez que se genera el esqueleto de un módulo mediante las tareas `generate:module` o `propel:generate-crud`, Symfony crea una prueba funcional de prueba para este módulo. La prueba realiza una petición a la acción por defecto del módulo y comprueba el código de estado de la respuesta, el módulo y la acción calculados por el sistema de enrutamiento y la presencia de una frase específica en el contenido de la respuesta. Si el módulo se llama `foobar`, el archivo `foobarActionsTest.php` generado es similar al del listado 15-9.

Listado 15-9 - Prueba funcional por defecto para un módulo nuevo, en `tests/functional/frontend/foobarActionsTest.php`

```
<?php

include(dirname(__FILE__).'/../bootstrap/functional.php');

// Create a new test browser
$browser = new sfTestBrowser();

$browser->
    get('/foobar/index')->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'index')->
    checkResponseElement('body', '!/This is a temporary page/')
    ;
```

Sugerencia Todos los métodos del navegador de Symfony devuelven un objeto `sfTestBrowser`, por lo que se pueden encadenar las llamadas a los métodos para que los archivos de prueba sean más fáciles de leer. Esta estrategia se llama "interfaz fluida con el objeto", ya que nada puede parar el flujo de llamadas a los métodos del objeto.

Las pruebas funcionales pueden contener varias peticiones y asertos más complejos, como se mostrará en las próximas secciones.

Para ejecutar una prueba funcional, se utiliza la tarea `test:functional` de la línea de comandos de Symfony, como se muestra en el listado 15-10. Los argumentos que se indican a la tarea son el nombre de la aplicación y el nombre de la prueba (omitiendo el sufijo `Test.php`).

Listado 15-10 - Ejecutando una prueba funcional mediante la línea de comandos

```
> php symfony test:functional frontend foobarActions

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
```

```
not ok 4 - response selector body does not match regex /This is a temporary page/
# Looks like you failed 1 tests of 4.
1..4
```

Por defecto, las pruebas funcionales generadas automáticamente para un módulo nuevo no pasan correctamente todas las pruebas. El motivo es que en los módulos nuevos, la acción `index` redirige a una página de bienvenida (que pertenece al módulo `default` de Symfony) que contiene la frase *"This is a temporary page"*. Mientras no se modifique la acción `index` del módulo, las pruebas funcionales de este módulo no se pasarán correctamente, lo que garantiza que no se ejecuten correctamente todas las pruebas para un módulo que está sin terminar.

Nota En las pruebas funcionales, la carga automática de clases está activada, por lo que no se deben incluir los archivos manualmente.

15.3.2. Navegando con el objeto `sfTestBrowser`

El navegador para pruebas permite realizar peticiones GET y POST. En ambos casos se utiliza una URI real como parámetro. El listado 15-11 muestra cómo crear peticiones con el objeto `sfTestBrowser` para simular peticiones reales.

Listado 15-11 - Simulando peticiones con el objeto `sfTestBrowser`

```
include(dirname(__FILE__).'../../bootstrap/functional.php');

// Se crea un nuevo navegador de pruebas
$b = new sfTestBrowser();

$b->get('/foobar/show/id/1');           // Petición GET
$b->post('/foobar/show', array('id' => 1)); // Petición POST

// Los métodos get() y post() son atajos del método call()
$b->call('/foobar/show/id/1', 'get');
$b->call('/foobar/show', 'post', array('id' => 1));

// El método call() puede simular peticiones de cualquier método
$b->call('/foobar/show/id/1', 'head');
$b->call('/foobar/add/id/1', 'put');
$b->call('/foobar/delete/id/1', 'delete');
```

Una navegación típica no sólo está formada por peticiones a determinadas acciones, sino que también incluye clicks sobre enlaces y botones. Como se muestra en el listado 15-12, el objeto `sfTestBrowser` también es capaz de simular la acción de pinchar sobre estos elementos.

Listado 15-12 - Simulando una navegación real con el objeto `sfTestBrowser`

```
$b->get('/');           // Petición a la página principal
$b->get('/foobar/show/id/1');
$b->back();             // Volver a la página anterior del historial
$b->forward();          // Ir a la página siguiente del historial
$b->reload();           // Recargar la página actual
$b->click('go');        // Buscar un enlace o botón llamado 'go' y pincharlo
```

El navegador para pruebas incluye un mecanismo para guardar todas las peticiones realizadas, por lo que los métodos `back()` y `forward()` funcionan de la misma manera que en un navegador real.

Sugerencia El navegador de pruebas incluye sus propios mecanismos para gestionar las sesiones (`sfTestStorage`) y las cookies.

Entre las interacciones que más se deben probar, las de los formularios son probablemente las más necesarias. Symfony dispone de 3 formas de probar la introducción de datos en los formularios y su envío. Se puede crear una petición POST con los parámetros que se quieren enviar, se puede llamar al método `click()` con los parámetros del formulario en un array o se pueden rellenar los campos del formulario de uno en uno y después pulsar sobre el botón de envío. En cualquiera de los 3 casos, la petición POST resultante es la misma. El listado 15-13 muestra un ejemplo.

Listado 15-13 - Simulando el envío de un formulario con datos mediante el objeto `sfTestBrowser`

```
// Plantilla de ejemplo en modules/foobar/templates/editSuccess.php
<?php echo form_tag('foobar/update') ?>
    <?php echo input_hidden_tag('id', $sf_params->get('id')) ?>
    <?php echo input_tag('name', 'foo') ?>
    <?php echo submit_tag('go') ?>
    <?php echo textarea('text1', 'foo') ?>
    <?php echo textarea('text2', 'bar') ?>
</form>

// Prueba funcional de ejemplo para este formulario
$b = new sfTestBrowser();
$b->get('/foobar/edit/id/1');

// Opción 1: petición POST
$b->post('/foobar/update', array('id' => 1, 'name' => 'dummy', 'commit' => 'go'));

// Opción 2: Pulsar sobre el botón de envío con parámetros
$b->click('go', array('name' => 'dummy'));

// Opción 3: Introducir los valores del formulario campo a campo y
// presionar el botón de envío
$b->setField('name', 'dummy')->
    click('go');
```

Nota En las opciones 2 y 3, los valores por defecto del formulario se incluyen automáticamente en su envío y no es necesario especificar el destino del formulario.

Si una acción finaliza con una redirección (`redirect()`), el navegador para pruebas no sigue automáticamente la redirección, sino que se debe seguir manualmente mediante `followRedirect()`, como se muestra en el listado 15-14.

Listado 15-14 - El navegador para pruebas no sigue automáticamente las redirecciones

```
// Acción de ejemplo en modules/foobar/actions/actions.class.php
public function executeUpdate($peticion)
{
```



```
// ...
$this->redirect('foobar/show?id='.$petition->getParameter('id'));
}

// Prueba funcional de ejemplo para esta acción
$b = new sfTestBrowser();
$b->get('/foobar/edit?id=1')->
    click('go', array('name' => 'dummy'))->
    isRedirected()->    // Check that request is redirected
    followRedirect();    // Manually follow the redirection
```

Existe un último método muy útil para la navegación: `restart()`, que inicializa el historial de navegación, la sesión y las cookies, es decir, como si se reiniciara el navegador.

Una vez realizada la primera petición, el objeto `sfTestBrowser` dispone de acceso directo a los objetos de la petición, del contexto y de la respuesta. De esta forma, se pueden probar muchas cosas diferentes, desde el contenido textual de las páginas a las cabeceras de la respuesta, pasando por los parámetros de la petición y la configuración:

```
$petition = $b->getRequest();
$contexto = $b->getContext();
$respuesta = $b->getResponse();
```

Todos los métodos para realizar la navegación descritos en los listados 15-10 a 15-13, no solamente están disponibles para las pruebas, sino que se pueden acceder desde cualquier parte de la aplicación mediante el objeto `sfBrowser`. La llamada que se debe realizar es la siguiente:

```
// Crear un nuevo navegador
$b = new sfBrowser();
$b->get('/foobar/show/id/1')->
    setField('name', 'dummy')->
    click('go');
$content = $b->getResponse()->getContent();
// ...
```

El objeto `sfBrowser` es muy útil para ejecutar scripts programados, como por ejemplo para navegar por una serie de páginas para generar la cache de cada página (el Capítulo 18 muestra un ejemplo detallado).

15.3.3. Utilizando asertos

Como el objeto `sfTestBrowser` dispone de acceso directo a la respuesta y a otros componentes de la petición, es posible realizar pruebas sobre estos componentes. Se podría crear un nuevo objeto `lime_test` para estas pruebas, pero por suerte, `sfTestBrowser` dispone de un método llamado `test()` que devuelve un objeto `lime_test` sobre el que se pueden invocar los métodos para asertos descritos anteriormente. El listado 15-15 muestra cómo realizar asertos mediante `sfTestBrowser`.

Listado 15-15 - El navegador para pruebas dispone del método `test()` para realizar pruebas

```
$b = new sfTestBrowser();
$b->get('/foobar/edit/id/1');
$request = $b->getRequest();
```

```

    $context = $b->getContext();
    $response = $b->getResponse();

    // Acceder a los métodos de Lime_test mediante el método test()
    $b->test()->is($request->getParameter('id'), 1);
    $b->test()->is($response->getStatusCode(), 200);
    $b->test()->is($response->getHttpHeader('content-type'), 'text/html;charset=utf-8');
    $b->test()->like($response->getContent(), '/edit/');

```

Nota Los métodos `getResponse()`, `getContext()`, `getRequest()` y `test()` no devuelven un objeto `sfTestBrowser`, por lo que no se pueden encadenar después de ellos otras llamadas a los métodos de `sfTestBrowser`.

Las cookies enviadas y recibidas se pueden probar fácilmente mediante los objetos de la petición y de la respuesta, como se muestra en el listado 15-16.

Listado 15-16 - Probando las cookies con `sfTestBrowser`

```

    $b->test()->is($request->getCookie('foo'), 'bar');      // Cookie enviada
    $cookies = $response->getCookies();
    $b->test()->is($cookies['foo'], 'foo=bar');            // Cookie recibida

```

Si se utiliza el método `test()` para probar los elementos de la petición, se acaban escribiendo unas líneas de código demasiado largas. Afortunadamente, `sfTestbrowser` contiene una serie de métodos especiales que permiten mantener las pruebas funcionales cortas y fáciles de leer, además de que devuelven objetos `sfTestBrowser`. El listado 15-15 se podría reescribir por ejemplo de forma más sencilla como se muestra en el listado 15-17.

Listado 15-17 - Realizando pruebas directamente con `sfTestBrowser`

```

    $b = new sfTestBrowser();
    $b->get('/foobar/edit/id/1')->
        isRequestParameter('id', 1)->
        isStatusCode()->
        isResponseHeader('content-type', 'text/html; charset=utf-8')->
        responseContains('edit');

```

El código de estado `200` es el valor por defecto que espera el método `isStatusCode()`, por lo que, para comprobar si la respuesta es correcta, se puede realizar la llamada sin argumentos.

Otra ventaja del uso de estos métodos especiales es que no es necesario especificar el texto que se muestra en la salida, como sí que era necesario en los métodos del objeto `lime_test`. Los mensajes se generan automáticamente en los métodos especiales, y la salida producida es clara y muy sencilla de entender.

```

    # get /foobar/edit/id/1
    ok 1 - request parameter "id" is "1"
    ok 2 - status code is "200"
    ok 3 - response header "content-type" is "text/html"
    ok 4 - response contains "edit"
    1..4

```

En la práctica, los métodos especiales del listado 15-17 cubren la mayor parte de las pruebas habituales, por lo que raramente es necesario utilizar el método `test()` del objeto `sfTestBrowser`.

El listado 15-14 demuestra que `sfTestBrowser` no sigue directamente las redirecciones. La ventaja de este comportamiento es que se pueden probar las propias redirecciones. El listado 15-18 muestra cómo probar la respuesta del listado 15-14.

Listado 15-18 - Probando las redirecciones con `sfTestBrowser`

```
$b = new sfTestBrowser();
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'update')->

    isRedirected()->      // Comprobar que la respuesta es una redirección
    followRedirect()->    // Obligar manualmente a seguir la redirección

    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'show');
```

15.3.4. Utilizando los selectores CSS

Muchas pruebas funcionales validan que una página sea correcta comprobando que un determinado texto se encuentre en el contenido de la respuesta. Utilizando el método `responseContains()` y las expresiones regulares, es posible comprobar que existe un determinado texto, los atributos de una etiqueta o sus valores. Pero si lo que se quiere probar se encuentra en lo más profundo del árbol DOM del contenido de la respuesta, la solución de las expresiones regulares es demasiado compleja.

Este es el motivo por el que el objeto `sfTestBrowser` dispone de un método llamado `getResponseDom()`. El método devuelve un objeto DOM de libXML2, que es mucho más fácil de procesar que el texto simple. El listado 15-19 muestra un ejemplo de uso de este método.

Listado 15-19 - El navegador para pruebas devuelve el contenido de la respuesta como un objeto DOM

```
$b = new sfTestBrowser();
$b->get('/foobar/edit/id/1');
$dom = $b->getResponseDom();
$b->test()->is($dom->getElementsByTagName('input')->item(1)->getAttribute('type'),'text');
```

Sin embargo, procesar un documento HTML con los métodos DOM de PHP no es lo suficientemente rápido y sencillo. Por su parte, los selectores utilizados en las hojas de estilos CSS son una forma aun más potente de obtener los elementos de un documento HTML. Symfony incluye una herramienta llamada `sfDomCssSelector`, cuyo constructor espera un documento DOM como argumento. Esta utilidad dispone de un método llamado `getTexts()` que devuelve un

array de las cadenas de texto seleccionadas mediante un selector CSS, y otro método llamado `getElements()` que devuelve un array de elementos DOM. El listado 15-20 muestra un ejemplo.

Listado 15-20 - El navegador para pruebas permite acceder al contenido de la respuesta mediante el objeto `sfDomCssSelector`

```
$b = new sfTestBrowser();
$b->get('/foobar/edit/id/1');
$c = new sfDomCssSelector($b->getResponseDom());
$b->test()->is($c->getTexts('form input[type="hidden"][value="1"]'), array(''));
$b->test()->is($c->getTexts('form textarea[name="text1"]'), array('foo'));
$b->test()->is($c->getTexts('form input[type="submit"]'), array(''));
```

Como es habitual, Symfony busca siempre la máxima brevedad y claridad en el código, por lo que se dispone de un método alternativo llamado `checkResponseElement()`. Utilizando este método, el listado 15-20 se puede transformar en el listado 15-21.

Listado 15-21 - El navegador para pruebas permite acceder a los elementos de la respuesta utilizando selectores de CSS

```
$b = new sfTestBrowser();
$b->get('/foobar/edit/id/1')->
    checkResponseElement('form input[type="hidden"][value="1"]', true)->
    checkResponseElement('form textarea[name="text1"]', 'foo')->
    checkResponseElement('form input[type="submit"]', 1);
```

El comportamiento del método `checkResponseElement()` depende del tipo de dato del segundo argumento que se le pasa:

- Si es un valor booleano, comprueba si existe un elemento que cumpla con el selector CSS indicado.
- Si es un número entero, comprueba que el selector CSS indicado devuelva el número de elementos de este parámetro.
- Si es una expresión regular, comprueba que el primer elemento seleccionado mediante el selector CSS cumpla el patrón de la expresión regular.
- Si es una expresión regular precedida de `!`, comprueba que el primer elemento seleccionado mediante el selector CSS no cumpla con el patrón de la expresión regular.
- En el resto de casos, compara el primer elemento seleccionado mediante el selector CSS y el valor del segundo argumento que se pasa en forma de cadena de texto.

El método acepta además un tercer parámetro opcional en forma de array asociativo. De esta forma es posible no solo realizar la prueba sobre el primer elemento devuelto por el selector CSS (si es que devuelve varios elementos) sino sobre otro elemento que se encuentra en una posición determinada, tal y como muestra el listado 15-22.

Listado 15-22 - Utilizando la opción de posición para comprobar un elemento que se encuentra en una posición determinada

```
$b = new sfTestBrowser();
$b->get('/foobar/edit?id=1')->
```

```
checkResponseElement('form textarea', 'foo')->
checkResponseElement('form textarea', 'bar', array('position' => 1));
```

El array de opciones también se puede utilizar para realizar 2 pruebas a la vez. Se puede comprobar que existe un elemento que cumple un selector y al mismo tiempo comprobar cuantos elementos lo cumplen, como se muestra en el listado 15-23.

Listado 15-23 - Utilizando la opción para contar el número de elementos que cumplen el selector CSS

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form input', true, array('count' => 3));
```

La herramienta del selector es bastante potente, ya que acepta la mayor parte de los selectores de CSS 3. De esta forma, se pueden hacer selecciones tan complejas como las que se muestran en el listado 15-24.

Listado 15-24 - Ejemplo de selectores CSS complejos que acepta checkResponseElement()

```
$b->checkResponseElement('ul#list li a[href]', 'click me');
$b->checkResponseElement('ul > li', 'click me');
$b->checkResponseElement('ul + li', 'click me');
$b->checkResponseElement('h1, h2', 'click me');
$b->checkResponseElement('a[class$="foo"][href*="bar.html"]', 'my link');
$b->checkResponseElement('p:last ul:nth-child(2) li:contains("Some text")');
```

15.3.5. Probando los errores

En ocasiones, las acciones o la parte del modelo lanzan excepciones a propósito (por ejemplo para mostrar una página de error 404). Aunque se pueden utilizar selectores CSS para comprobar la presencia de un determinado mensaje de error en el código HTML generado, es mejor utilizar el método `throwsException` para comprobar si se ha lanzado una excepción, tal y como muestra el listado 15-25.

Listado 15-25 - Probando las excepciones

```
$b = new sfTestBrowser();
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'update')->

    throwsException()->                                // Comprueba que la última petición lanza una
    excepción
    throwsException('RuntimeException')-> // Comprueba el tipo de excepción
    throwsException(null, '/error/');    // Comprueba si el mensaje de la excepción
    cumple con una expresión regular
```

15.3.6. Trabajando en el entorno de pruebas

El objeto `sfTestBrowser` utiliza un controlador frontal especial, que trabaja en el entorno `test`. El listado 15-26 muestra la configuración por defecto de este entorno.

Listado 15-26 - Configuración por defecto del entorno test, en frontend/config/settings.yml

```
test:
  .settings:
    error_reporting:    <?php echo (E_ALL | E_STRICT & ~E_NOTICE)."\n" ?>
    cache:              off
    web_debug:          off
    no_script_name:     off
    etag:               off
```

En este entorno, la cache y la barra de depuración web están desactivadas. No obstante, la ejecución del código genera logs en un archivo distinto a los logs de los entornos `dev` y `prod`, por lo que se pueden observar de forma independiente (en `miproyecto/log/frontend_test.log`). Además, en este entorno las excepciones no detienen la ejecución de los scripts, de forma que se pueda ejecutar un conjunto completo de pruebas incluso cuando falla alguna prueba. También es posible definir una conexión específica con la base de datos, por ejemplo para utilizar una base de datos que tenga datos de prueba.

Antes de utilizar el objeto `sfTestBrowser`, es necesario inicializarlo. Si se necesita, es posible especificar el nombre del servidor para la aplicación y una dirección IP para el cliente, por si la aplicación controla estos dos parámetros. El listado 15-27 muestra cómo configurar estos parámetros.

Listado 15-27 - Indicar el hostname y la IP en el navegador para pruebas

```
| $b = new sfTestBrowser('miaplicacion.ejemplo.com', '123.456.789.123');
```

15.3.7. La tarea test:functional

La tarea `test:functional` puede ejecutar una o varias pruebas funcionales, dependiendo del número de argumentos indicados. La sintaxis que se utiliza es muy similar a la de la tarea `test:unit`, salvo que la tarea para pruebas funcionales requiere como primer argumento el nombre de una aplicación, tal y como muestra el listado 15-28.

Listado 15-28 - Sintaxis de la tarea para pruebas funcionales

```
// Estructura del directorio de pruebas
test/
  functional/
    frontend/
      miModuloActionsTest.php
      miEscenarioTest.php
    backend/
      miOtroEscenarioTest.php
## Ejecutar todas las pruebas funcionales de una aplicacion recursivamente
> php symfony test:functional frontend
```

```
## Ejecutar la prueba funcional cuyo nombre se indica como parámetro
> php symfony test:functional frontend myScenario

## Ejecutar todas las pruebas funcionales cuyos nombres cumplan con el patrón indicado
> php symfony test:functional frontend my*
```

15.4. Recomendaciones sobre el nombre de las pruebas

En esta sección se presentan algunas de las buenas prácticas para mantener bien organizadas las pruebas y facilitar su mantenimiento. Los consejos abarcan la organización de los archivos, de las pruebas unitarias y de las pruebas funcionales.

En lo que respecta a la estructura de archivos, los archivos de las pruebas unitarias deberían nombrarse según la clase que se supone que están probando y las pruebas funcionales deberían nombrarse en función del módulo o escenario que se supone que están probando. El listado 15-29 muestra un ejemplo de estas recomendaciones. Como el número de archivos en el directorio `test/` puede crecer muy rápidamente, si no se siguen estas recomendaciones, es posible que sea muy difícil encontrar el archivo de una prueba determinada.

Listado 15-29 - Ejemplo de recomendaciones sobre el nombre de los archivos

```
test/
  unit/
    miFuncionTest.php
    miSegundaFuncionTest.php
  foo/
    barTest.php
  functional/
    frontend/
      miModuloActionsTest.php
      miEscenarioTest.php
    backend/
      miOtroEscenarioTest.php
```

En las pruebas unitarias, una buena práctica consiste en agrupar las pruebas según la función o método y comenzar cada grupo de pruebas con una llamada al método `diag()`. Los mensajes de cada prueba unitaria deberían mostrar el nombre de la función o método que se prueba, seguido de un verbo y una propiedad, de forma que el resultado que se muestra parezca una frase que describe una propiedad de un objeto. El listado 15-30 muestra un ejemplo.

Listado 15-30 - Ejemplo de recomendaciones para las pruebas unitarias

```
// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string', 'strtolower() devuelve una cadena de texto');
$t->is(strtolower('FOO'), 'foo', 'strtolower() transforma la entrada en minúsculas');
# strtolower()
ok 1 - strtolower() devuelve una cadena de texto
ok 2 - strtolower() transforma la entrada en minúsculas
```

Las pruebas funcionales deberían agruparse por página y deberían comenzar con una petición. El listado 15-31 muestra un ejemplo de esta práctica.

Listado 15-31 - Ejemplo de recomendaciones para las pruebas funcionales

```

$browser->
    get('/foobar/index')->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'index')->
    checkResponseElement('body', '/foobar/')
;
# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
ok 4 - response selector body matches regex /foobar/

```

Si se sigue esta recomendación, el resultado de la prueba es lo suficientemente claro como para utilizarlo como documentación técnica del proyecto, ya que puede hacer innecesaria la propia documentación de la aplicación.

15.5. Otras utilidades para pruebas

Las herramientas que incluye Symfony para realizar pruebas unitarias y funcionales son suficientes para la mayoría de casos. No obstante, se muestran a continuación algunas técnicas adicionales para resolver problemas comunes con las pruebas automatizadas: ejecutar pruebas en un entorno independiente, acceder a la base de datos desde las pruebas, probar la cache y realizar pruebas de las interacciones en el lado del cliente.

15.5.1. Ejecutando las pruebas en grupos

Las tareas `test:unit` y `test:functional` ejecutan una sola prueba o un conjunto de pruebas. Sin embargo, si se ejecutan las tareas sin indicar ningún parámetro, se lanzan todas las pruebas unitarias y funcionales del directorio `test/`. Para evitar el riesgo de *interferencias* de unas pruebas a otras, cada prueba se ejecuta en un entorno de ejecución independiente. Además, cuando se ejecutan todas las pruebas, el resultado que se muestra no es el mismo que el que genera cada prueba de forma independiente, ya que en este caso la salida estaría formada por miles de líneas. Lo que se hace es generar una salida resumida especialmente preparada. Por este motivo, la ejecución de un gran número de pruebas utiliza un *test harness*, que es un framework de pruebas con algunas características especiales. El *test harness* depende de un componente del framework Lime llamado `lime_harness`. Como se muestra en el listado 15-32, la salida producida indica el estado de las pruebas archivo por archivo y al final se muestra un resumen de todas las pruebas que se han pasado y el número total de pruebas.

Listado 15-32 - Ejecutando todas las pruebas mediante el *test harness*

```

> php symfony test:all

unit/miFuncionTest.php.....ok
unit/miSegundaFuncionTest.php.....ok
unit/foo/barTest.php.....not ok

Failed Test                               Stat  Total  Fail  List of Failed

```



```
-----
unit/foo/barTest.php          0      2      2  62 63
Failed 1/3 test scripts, 66.66% okay. 2/53 subtests failed, 96.22% okay.
```

Las pruebas se ejecutan de la misma forma que si se lanzaran una a una, solamente es la salida la que se resume para hacerla más útil. De hecho, la estadística final se centra en las pruebas que no han tenido éxito y ayuda a localizarlas.

Incluso es posible lanzar todas las pruebas de cualquier tipo mediante la tarea `test:all`, que también hace uso del *test harness*, como se muestra en el listado 15-33. Una buena práctica consiste en ejecutar esta tarea antes de realizar el paso a producción del nuevo código, ya que asegura que no se ha introducido ningún nuevo error desde la versión anterior.

Listado 15-33 - Ejecutando todas las pruebas de un proyecto

```
> php symfony test:all
```

15.5.2. Acceso a la base de datos

Normalmente, las pruebas unitarias necesitan acceder a la base de datos. Cuando se llama al método `sfTestBrowser::get()` por primera vez, se inicializa una conexión con la base de datos. No obstante, si se necesita acceder a la base de datos antes de utilizar `sfTestBrowser`, se debe inicializar el objeto `sfDatabaseManager` a mano, como muestra el listado 15-34.

Listado 15-34 - Inicializando la base de datos en una prueba

```
$databaseManager = new sfDatabaseManager();
$databaseManager->loadConfiguration();
// Opcionalmente, se puede obtener la conexión con la base de datos
$con = Propel::getConnection();
```

Antes de comenzar las pruebas, se suele cargar la base de datos con datos de prueba, también llamados *fixtures*. El objeto `sfPropelData` permite realizar esta carga. No solamente es posible utilizar este objeto para cargar datos a partir de un archivo (como con la tarea `propel:data-load`) sino que también es posible hacerlo desde un array, como muestra el listado 15-35.

Listado 15-35 - Cargando datos en la base de datos desde una prueba

```
$data = new sfPropelData();

// Cargar datos desde un archivo
$data->loadData(sfConfig::get('sf_data_dir').'/fixtures/test_data.yml');

// Cargar datos desde un array
$fixtures = array(
    'Article' => array(
        'article_1' => array(
            'title'      => 'foo title',
            'body'       => 'bar body',
            'created_at' => time(),
        ),
        'article_2' => array(
            'title'      => 'foo foo title',
```

```

        'body'          => 'bar bar body',
        'created_at' => time(),
    ),
),
);
$data->loadDataFromArray($fixtures);

```

Una vez cargados los datos, se pueden utilizar los objetos Propel necesarios como en cualquier aplicación normal. Las pruebas unitarias deben incluir los archivos correspondientes a esos objetos (se puede utilizar la clase `sfSimpleAutoload` para automatizar la carga, como se explicó en la sección anterior "Stubs, Fixtures y carga automática de clases"). Los objetos de Propel se cargan automáticamente en las pruebas funcionales.

15.5.3. Probando la cache

Cuando se habilita la cache para una aplicación, las pruebas funcionales se encargan de verificar que las acciones guardadas en la cache se comportan como deberían.

En primer lugar, se habilita la cache para el entorno de pruebas (en el archivo `settings.yml`). Una vez habilitada, se puede utilizar el método `isCached()` del objeto `sfTestBrowser` para comprobar si una página se ha obtenido directamente de la cache o ha sido generada en ese momento. El listado 15-36 muestra cómo utilizar este método.

Listado 15-36 - Probando la cache con el método `isCached()`

```

<?php

include(dirname(__FILE__).'../../bootstrap/functional.php');

// Create a new test browser
$b = new sfTestBrowser();

$b->get('/mymodule');
$b->isCached(true);           // Comprueba si La respuesta viene de La cache
$b->isCached(true, true);    // Comprueba si La respuesta de La cache incluye el
                             layoutlayout
$b->isCached(false);         // Comprueba que La respuesta no venga de La cache

```

Nota No es necesario borrar la cache antes de realizar la prueba funcional, ya que el proceso de arranque utilizado por la prueba se encarga de hacerlo automáticamente.

15.5.4. Probando las interacciones en el lado del cliente

El principal inconveniente de las técnicas descritas anteriormente es que no pueden simular el comportamiento de JavaScript. Si se definen interacciones muy complejas, como por ejemplo interacciones con Ajax, es necesario reproducir de forma exacta los movimientos del ratón y las pulsaciones de teclado que realiza el usuario y ejecutar los scripts de JavaScript. Normalmente, estas pruebas se hacen a mano, pero cuestan mucho tiempo y son propensas a cometer errores.

La solución a estos problemas se llama Selenium (<http://www.openqa.org/selenium/>), que consiste en un framework de pruebas escrito completamente en JavaScript. Selenium permite realizar una serie de acciones en la página de la misma forma que las haría un usuario normal. La

ventaja de Selenium sobre el objeto `sfBrowser` es que Selenium es capaz de ejecutar todo el código JavaScript de la página, incluidas las interacciones creadas con Ajax.

Symfony no incluye Selenium por defecto. Para instalarlo, se crea un directorio llamado `selenium/` en el directorio `web/` del proyecto y se descomprime el contenido del archivo descargado desde <http://www.openqa.org/selenium-core/download.action>. Como Selenium se basa en JavaScript y la mayoría de navegadores tienen unas restricciones de seguridad muy estrictas, es importante ejecutar Selenium desde el mismo servidor y el mismo puerto que el que utiliza la propia aplicación.

Cuidado Debe ponerse especial cuidado en no subir el directorio `selenium/` al servidor de producción, ya que estaría disponible para cualquier usuario que acceda a la raíz del servidor desde un navegador web.

Las pruebas de Selenium se escriben en HTML y se guardan en el directorio `web/selenium/tests/`. El listado 15-37 muestra un ejemplo de prueba funcional en la que se carga la página principal, se pulsa el enlace "pinchame" y se busca el texto "Hola Mundo" en el contenido de la respuesta. Para acceder a la aplicación en el entorno test, se debe utilizar el controlador frontal llamado `frontend_test.php`.

Listado 15-37 - Un ejemplo de prueba de Selenium, en `web/selenium/test/testIndex.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
  <title>Index tests</title>
</head>
<body>
<table cellpadding="0">
<tbody>
  <tr><td colspan="3">First step</td></tr>
  <tr><td>open</td>          <td>/frontend_test.php/</td> <td>&nbsp;</td></tr>
  <tr><td>clickAndWait</td>  <td>link=pinchame</td>    <td>&nbsp;</td></tr>
  <tr><td>assertTextPresent</td> <td>Hola Mundo</td>    <td>&nbsp;</td></tr>
</tbody>
</table>
</body>
</html>
```

Cada caso de prueba consiste en una página HTML con una tabla de 3 columnas: comando, destino y valor. No obstante, no todos los comandos indican un valor. En caso de que no se utilice un valor, es recomendable incluir el valor ` ` en esa columna (para que la tabla se vea mejor). El sitio web de Selenium dispone de la lista completa de comandos que se pueden utilizar.

También es necesario añadir esta prueba al conjunto completo de pruebas, insertando una nueva línea en la tabla del archivo `TestSuite.html` del mismo directorio. El listado 15-38 muestra cómo hacerlo.

Listado 15-38 - Añadiendo un archivo de pruebas al conjunto de pruebas, en web/selenium/test/TestSuite.html

```
...  
<tr><td><a href='./testIndex.html'>Mi primera prueba</a></td></tr>  
...
```

Para ejecutar la prueba, solamente es necesario acceder a la página:

```
| http://miaplicacion.ejemplo.com/selenium/index.html
```

Si se selecciona la *"Main Test Suite"* y se pulsa sobre el botón de ejecutar todas las pruebas, el navegador reproducirá automáticamente todos los pasos que se han indicado.

Nota Como las pruebas de Selenium se ejecutan en el propio navegador, permiten descubrir las inconsistencias entre navegadores. Si se construye la prueba para un solo navegador, se puede lanzar esa prueba sobre todos los navegadores y comprobar su funcionamiento.

Como las pruebas de Selenium se crean con HTML, acaba siendo muy aburrido escribir todo ese código HTML. Afortunadamente, existe una extensión de Selenium para Firefox (<http://seleniumrecorder.mozdev.org/>) que permite *grabar* todos los movimientos y acciones realizadas sobre una página y guardarlos como una prueba. Mientras se graba una sesión de navegación, se pueden añadir pruebas de tipo asertos pulsando el botón derecho sobre la ventana del navegador y seleccionando la opción apropiada del menú *"Append Selenium Command"*.

Una vez realizados todos los movimientos y añadidos todos los comandos, se pueden guardar en un archivo HTML para añadirlo al conjunto de pruebas. La extensión de Firefox incluso permite ejecutar las pruebas de Selenium que se han creado con la extensión.

Nota No debe olvidarse reinicializar los datos de prueba antes de lanzar cada prueba de Selenium.

15.6. Resumen

La automatización de pruebas abarca tanto las pruebas unitarias (que validan métodos o funciones) como las pruebas funcionales (que validan características completas de la aplicación). Symfony utiliza el framework de pruebas Lime para las pruebas unitarias y proporciona la clase `sfTestBrowser` para las pruebas funcionales. En ambos casos, se dispone de métodos para realizar todo tipo de asertos, desde los más sencillos hasta los más complejos, como por ejemplo los que se realizan mediante los selectores de CSS. La línea de comandos de Symfony permite ejecutar las pruebas de una en una (mediante las tareas `test:unit` y `test:functional`) o en grupo (mediante la tarea `test:all`). En lo que respecta a los datos, las pruebas automatizadas utilizan *stubs* (clases falsas) y *fixtures* (datos de prueba complejos) y Symfony simplifica su uso desde las pruebas unitarias.

Si se definen las suficientes pruebas unitarias como para probar la mayor parte de una aplicación (quizás aplicando la metodología TDD), es mucho más seguro refactorizar el código de la aplicación y añadir nuevas características. Incluso, en ocasiones, las pruebas pueden reducir el tiempo requerido para la documentación técnica del proyecto.

Capítulo 16. Herramientas para la administración de aplicaciones

Durante el desarrollo y la instalación de las aplicaciones, los programadores necesitan toda la información posible para determinar si la aplicación está funcionando como debería. Normalmente, esta información se obtiene mediante los archivos de log y las herramientas de depuración o *debug*. Los frameworks como Symfony son el núcleo de las aplicaciones, por lo que es esencial que el propio framework disponga de las herramientas necesarias para asegurar un desarrollo eficiente de las aplicaciones.

Durante la ejecución de una aplicación en el servidor de producción, el administrador de sistemas repite una serie de tareas, como la rotación de los logs, la actualización de las aplicaciones, etc. Por este motivo, un framework también debe proporcionar las herramientas necesarias para automatizar lo más posible estas tareas.

En este capítulo se detallan las herramientas de gestión de aplicaciones que dispone Symfony para realizar todas las tareas anteriores.

16.1. Logs

La única forma de comprender lo sucedido cuando falla la ejecución de una petición, consiste en echar un vistazo a la traza generada por el proceso que se ejecuta. Afortunadamente, y como se va a ver en esta sección, tanto PHP como Symfony guardan mucha información de este tipo en archivos de log.

16.1.1. Logs de PHP

PHP dispone de una directiva llamada `error_reporting`, que se define en el archivo de configuración `php.ini`, y que especifica los eventos de PHP que se guardan en el archivo de log. Symfony permite redefinir el valor de esta opción, tanto a nivel de aplicación como de entorno, en el archivo `settings.yml`, tal y como se muestra en el listado 16-1.

Listado 16-1 - Indicando el valor de la directiva `error_reporting`, en `frontend/config/settings.yml`

```
prod:
  .settings:
    error_reporting: <?php echo (E_PARSE | E_COMPILE_ERROR | E_ERROR | E_CORE_ERROR |
E_USER_ERROR)."\n" ?>

dev:
  .settings:
    error_reporting: <?php echo (E_ALL | E_STRICT)."\n" ?>
```

Para no penalizar el rendimiento de la aplicación en el entorno de producción, el servidor solamente guarda en el archivo de log los errores críticos de PHP. No obstante, en el entorno de

desarrollo, se guardan en el log todos los tipos de eventos, de forma que el programador puede disponer de la máxima información para seguir la pista a los errores.

El lugar en el que se guardan los archivos de log de PHP depende de la configuración del archivo `php.ini`. Si no se ha modificado su valor, PHP utiliza las herramientas de log del servidor web (como por ejemplo los logs de error del servidor Apache). En este caso, los archivos de log de PHP se encuentran en el directorio de logs del servidor web.

16.1.2. Logs de Symfony

Además de los archivos de log creados por PHP, Symfony también guarda mucha información de sus propios eventos en otros archivos de log. Los archivos de log creados por Symfony se encuentran en el directorio `miproyecto/log/`. Symfony crea un archivo por cada aplicación y cada entorno. El archivo del entorno de desarrollo de una aplicación llamada `frontend` sería `frontend_dev.log` y el archivo de log del entorno de producción de la misma aplicación se llamaría `frontend_prod.log`.

Si se dispone de una aplicación Symfony ejecutándose, se puede observar que la sintaxis de los archivos de log generados es muy sencilla. Cada evento resulta en una nueva línea en el archivo de log de la aplicación. Cada línea incluye la fecha y hora a la que se ha producido, el tipo de evento, el objeto que ha sido procesado y otros detalles relevantes que dependen de cada tipo de evento y/o objeto procesado. El listado 16-2 muestra un ejemplo del contenido de un archivo de log de Symfony.

Listado 16-2 - Contenido de un archivo de log de Symfony, en `log/frontend_dev.php`

```
Nov 15 16:30:25 symfony [info ] {sfAction} call "barActions->executemessages()"
Nov 15 16:30:25 symfony [debug] SELECT bd_message.ID, bd_message.SENDER_ID, bd...
Nov 15 16:30:25 symfony [info ] {sfCreole} executeQuery(): SELECT bd_message.ID...
Nov 15 16:30:25 symfony [info ] {sfView} set slot "leftbar" (bar/index)
Nov 15 16:30:25 symfony [info ] {sfView} set slot "messageblock" (bar/mes...
Nov 15 16:30:25 symfony [info ] {sfView} execute view for template "messa...
Nov 15 16:30:25 symfony [info ] {sfView} render "/home/production/miproyecto/...
Nov 15 16:30:25 symfony [info ] {sfView} render to client
```

Estos archivos de log contienen mucha información, como por ejemplo las consultas SQL enviadas a la base de datos, las plantillas que se han procesado, las llamadas realizadas entre objetos, etc.

A partir de Symfony 1.1 es posible personalizar el formato de los mensajes de log redefiniendo las opciones `format` y/o `time_format` en el archivo de configuración `factories.yml`, tal y como muestra el listado 16-3.

Listado 16-3 - Modificando el formato de los mensajes de log

```
all:
  logger:
    param:
      format:      %time% %type% [%priority%] %message%%EOL%
      time_format: %b %d %H:%M:%S
```

16.1.2.1. Configuración del nivel de log de Symfony

Symfony define ocho niveles diferentes para los mensajes de log: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` y `debug`, que son los mismos niveles que define el paquete `PEAR::Log` (<http://pear.php.net/package/Log/>). El archivo de configuración `factories.yml` de cada aplicación permite definir el nivel de los mensajes que se guardan en el archivo de log, como se muestra en el listado 16-4.

Listado 16-4 - Configuración por defecto de los archivos de log en Symfony, en `frontend/config/factories.yml`

```
prod:
  logger:
    param:
      level: err
```

Por defecto, en todos los entornos salvo en el de producción, se guardan en los archivos de log todos los mensajes (hasta el nivel menos importante, el nivel `debug`). En el entorno de producción, los archivos de log están deshabilitados. Además, en este mismo entorno, si se activan los logs asignando el valor `on` a la opción `logging_enabled` en el archivo de configuración `settings.yml`, solamente se guardan los mensajes más importantes (de `crit` a `emerg`).

En el archivo `factories.yml` se puede modificar el nivel de los mensajes guardados para cada entorno de ejecución, de forma que se limite el tipo de mensajes que se guardan en el archivo de log.

Sugerencia Para determinar si están habilitados los archivos de log, se puede utilizar la instrucción `sfConfig::get('sf_logging_enabled')`.

16.1.2.2. Añadiendo un mensaje de log

Además de los mensajes generados por Symfony, también es posible añadir mensajes propios en el archivo de log desde el código de la aplicación, utilizando alguna de las técnicas mostradas en el listado 16-5.

Listado 16-5 - Añadiendo un mensaje de log propio

```
// Desde la acción
$this->logMessage($mensaje, $nivel);

// Desde una plantilla
<?php use_helper('Debug') ?>
<?php log_message($mensaje, $nivel) ?>
```

El valor de la opción `$nivel` puede ser uno de los valores definidos para los mensajes de log de Symfony.

Además, para escribir un mensaje en el log desde cualquier punto de la aplicación, se pueden utilizar directamente los métodos de `sfLogger`, como se muestra en el listado 16-6. Los métodos disponibles comparten el mismo nombre que los niveles de log definidos.

Listado 16-6 - Añadiendo un mensaje de log propio desde cualquier punto de la aplicación

```
if (sfConfig::get('sf_logging_enabled'))
{
    sfContext::getInstance()->getLogger()->info($mensaje);
}
```

El mecanismo de log de Symfony es muy sencillo y es muy fácil de personalizar.

El único requisito es que las clases del nuevo mecanismo de log deben heredar de la clase `sfLogger`, que define un método llamado `doLog()`. Symfony invoca el método `doLog()` con dos parámetros: `$mensaje` (el mensaje que se quiere guardar en el log) y `$prioridad` (el nivel del mensaje).

La siguiente clase `miPropioLog` define un mecanismo de log muy sencillo que utiliza la función `error_log` de PHP:

```
class miPropioLog extends sfLogger
{
    protected function doLog($mensaje, $prioridad)
    {
        error_log(sprintf('%s (%s)', $mensaje, sfLogger::getPriorityName($prioridad)));
    }
}
```

Para crear un mecanismo de log a partir de una clase existente, puedes implementar la interfaz `sfLoggerInterface`, que define un método llamado `log()` que toma los mismos parámetros que el método `doLog()` anterior:

```
require_once('Log.php');
require_once('Log/error_log.php');

// Clase sencilla que implementa la interfaz del mecanismo de Log
// que se quiere utilizar con Symfony
class Log_my_error_log extends Log_error_log implements sfLoggerInterface
{
}
```

16.1.2.3. Borrando y rotando archivos de log

Periódicamente es necesario borrar los archivos del directorio `log/` de las aplicaciones, ya que estos archivos suelen crecer en tamaño varios MB cada pocos días, aunque todo depende del tráfico de la aplicación. Symfony proporciona la tarea `log:clear` para este propósito, y se puede ejecutar de forma periódica manualmente o mediante una tarea programada. El siguiente comando por ejemplo borra todos los archivos de log:

```
> php symfony log:clear
```

Para mejorar el rendimiento y la seguridad de la aplicación, suele ser habitual almacenar los archivos de log de Symfony en varios archivos pequeños en vez de en un solo archivo muy grande. La estrategia de almacenamiento ideal para los archivos de log es la de vaciar y hacer una copia de seguridad cada poco tiempo del archivo de log principal y mantener un número limitado de copias de seguridad.

Esta estrategia se denomina *rotación de archivos de log* y el listado 16-7 muestra cómo activar una rotación con un período (period) de 7 días y un historial o número de copias de seguridad (history) de 10. De esta forma, se trabaja con un archivo de log activo y se dispone de 10 copias de seguridad, cada una con los mensajes de log de 7 días diferentes. Cuando transcurren otros 7 días, el archivo de log activo se transforma en una copia de seguridad y se borra el archivo de la copia de seguridad más antigua.

Listado 16-7 - Ejecutando la rotación de logs

```
| > php symfony log:rotate frontend prod --period=7 --history=10
```

Las copias de seguridad de los archivos de log se almacenan en el directorio `logs/history/` y a su nombre se les añade un sufijo con la fecha completa en la que fueron guardados.

16.2. Depuración de aplicaciones

No importa lo buenos que sean los programadores o lo bueno que sea Symfony, siempre se acaban cometiendo errores. Una de las claves para el desarrollo rápido de aplicaciones es la detección y comprensión de los errores producidos. Afortunadamente, Symfony proporciona varias herramientas para depurar las aplicaciones.

16.2.1. Modo debug de Symfony

Symfony dispone de un modo llamado "debug" que facilita el desarrollo y la depuración de las aplicaciones. Cuando se activa este modo, ocurre lo siguiente:

- La configuración de la aplicación se comprueba en cada petición, por lo que cualquier cambio en la configuración se aplica inmediatamente, sin necesidad de borrar la cache de configuración.
- Los mensajes de error muestran la traza completa de ejecución de forma clara y útil, para que sea más fácil de encontrar el elemento que está fallando.
- Se muestran más herramientas de depuración (como por ejemplo, todas las consultas a la base de datos).
- También se activa el modo debug de Propel, por lo que cualquier error en la llamada a un objeto de Propel, muestra una lista completa de los errores producidos en toda la arquitectura Propel.

Por otra parte, cuando se desactiva el modo debug, las peticiones se procesan de la siguiente forma:

- Los archivos de configuración YAML se procesan una sola vez y se transforman en archivos PHP que se almacenan en la carpeta `cache/config/`. Todas las peticiones que se realizan después de la primera petición, no tienen en cuenta los archivos YAML de configuración y utilizan en su lugar la configuración guardada en la cache. Por tanto, el procesamiento de cada petición es mucho más rápido.
- Para forzar a que se vuelva a procesar la configuración de la aplicación, es necesario borrar a mano la cache de configuración.

- Cualquier error que se produzca durante el procesamiento de la petición, devuelve una respuesta con el código de estado 500 (Error Interno del Servidor) y no se muestran los detalles internos del error.

El modo debug se activa para cada aplicación en su controlador frontal. Este modo se controla mediante el valor del tercer argumento que se pasa al método `getApplicationConfiguration()`, como se muestra en el listado 16-8.

Listado 16-8 - Controlador frontal de ejemplo con el modo debug activado, en `web/frontend_dev.php`

```
<?php

require_once(dirname(__FILE__).'../config/ProjectConfiguration.class.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend', 'dev',
true);
sfContext::createInstance($configuration)->dispatch();
```

Cuidado En el servidor de producción, no se debería activar el modo debug y no se debería guardar ningún controlador frontal con este modo activado. El modo debug no solo penaliza el rendimiento de la aplicación, sino que revela información interna de la aplicación. Aunque las herramientas de depuración nunca desvelan la información necesaria para conectarse con la base de datos, la traza generada en las excepciones está llena de información demasiado sensible y que puede ser aprovechada por un usuario malintencionado.

16.2.2. Excepciones Symfony

Cuando se produce una excepción y está activado el modo debug, Symfony muestra un mensaje de error muy útil que contiene toda la información necesaria para descubrir la causa del problema.

Los mensajes que produce la excepción están escritos de forma clara y hacen referencia a la causa más probable del problema. Normalmente ofrecen posibles soluciones para arreglar el error y para la mayoría de problemas comunes, incluso se muestra un enlace a la página del sitio web de Symfony que contiene más información sobre la excepción. La página con el mensaje de la excepción muestra en qué parte del código PHP se ha producido el error y la lista completa de los métodos que se han invocado, como se muestra en la figura 16-1. De esta forma, es posible seguir la traza de ejecución hasta la primera llamada que causó el problema. También se muestran los argumentos que se pasan a cada método.

Nota Symfony se basa en las excepciones de PHP para notificar los errores, que es un método mucho mejor que el funcionamiento de las aplicaciones desarrolladas con PHP 4. Para notificar un error de tipo 404, se utiliza el método `sfError404Exception`.

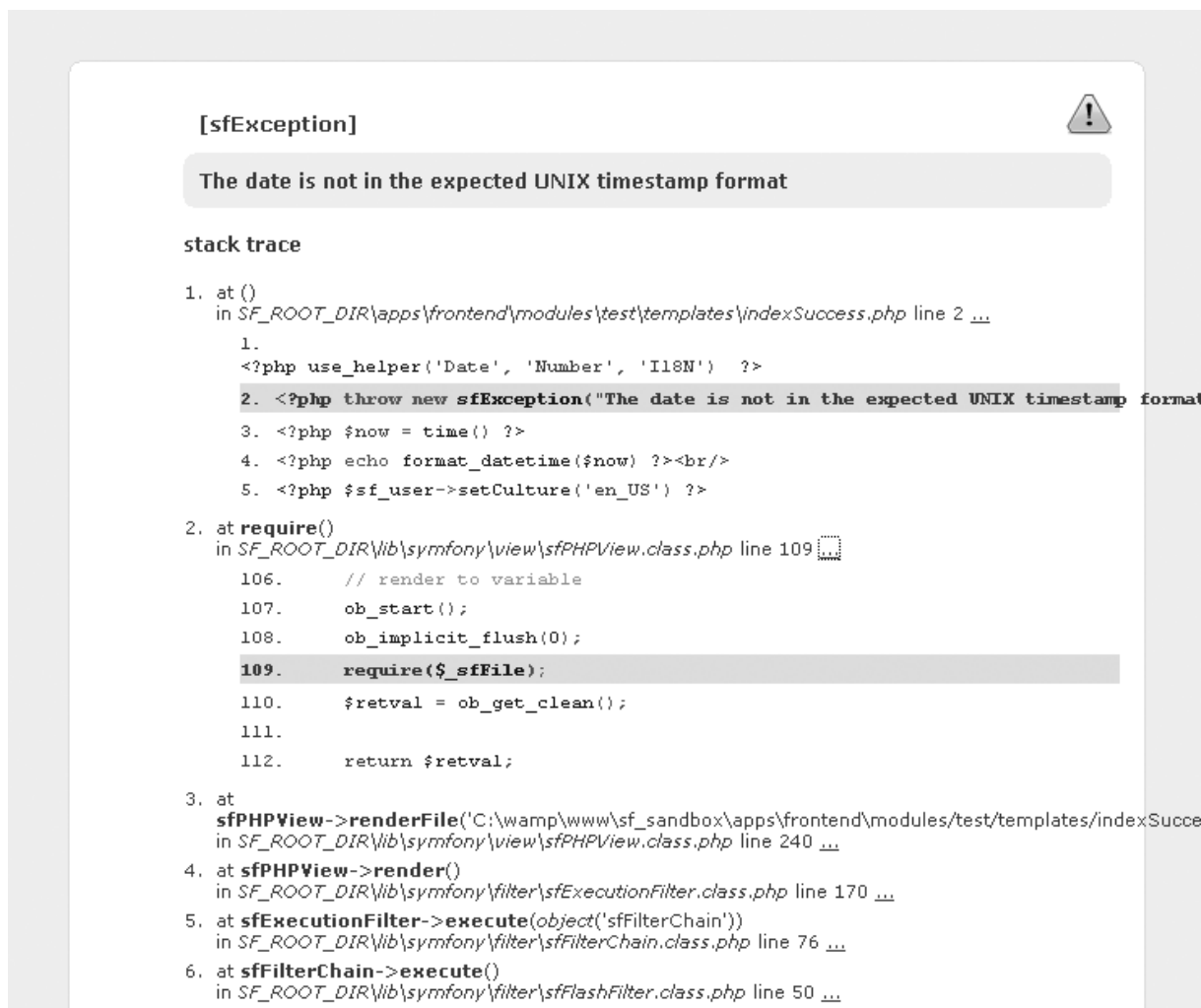


Figura 16.1. Mensaje mostrado por una excepción de una aplicación Symfony

Mientras se desarrolla la aplicación, las excepciones Symfony son de gran utilidad para depurar el funcionamiento de las aplicaciones.

16.2.3. Extensión Xdebug

La extensión Xdebug de PHP (<http://xdebug.org/>) permite ampliar la cantidad de información que el servidor web almacena en los archivos de log. Symfony es capaz de integrar los mensajes de Xdebug en sus propios mensajes de error, por lo que es una buena idea activar esta extensión cuando se están depurando las aplicaciones. La instalación de la extensión depende de la plataforma en la que se realiza, por lo que se debe consultar la información disponible en el sitio web de Xdebug. Una vez instalada, se activa manualmente en el archivo de configuración `php.ini`. En los sistemas *nix, se activa añadiendo la siguiente línea:

```
| zend_extension="/usr/local/lib/php/extensions/no-debug-non-zts-20041030/xdebug.so"
```

En los sistemas Windows, la activación de Xdebug se realiza mediante:

```
| extension=php_xdebug.dll
```

El listado 16-9 muestra un ejemplo de la configuración de Xdebug, que también se debe añadir al archivo `php.ini`.

Listado 16-9 - Configuración de ejemplo para Xdebug

```
;xdebug.profiler_enable=1
;xdebug.profiler_output_dir="/tmp/xdebug"
xdebug.auto_trace=1           ; enable tracing
xdebug.trace_format=0
;xdebug.show_mem_delta=0      ; memory difference
;xdebug.show_local_vars=1
;xdebug.max_nesting_level=100
```

Por último, para activar la extensión Xdebug, se debe reiniciar el servidor web.

Cuidado No debe olvidarse desactivar la extensión Xdebug en el servidor de producción. Si no se desactiva, el rendimiento de la aplicación disminuye notablemente.

16.2.4. Barra de depuración web

Los archivos de log guardan información muy útil, pero no siempre son fáciles de leer. La tarea más básica, que consiste en localizar las líneas del archivo de log correspondientes a una determinada petición, suele complicarse cuando existen varios usuarios simultáneos en la aplicación y cuando el archivo de log es muy grande. En ese momento es cuando se hace necesaria la barra de depuración web.

Esta barra de depuración se muestra como una caja semitransparente superpuesta sobre el contenido de la ventana del navegador y que aparece en la esquina superior derecha, como se ve en la figura 16-2. Esta barra permite acceder directamente a los eventos guardados en el log, a la configuración actual, las propiedades de los objetos de la petición y de la respuesta, los detalles de las consultas realizadas a la base de datos y una tabla con los tiempos empleados en cada elemento de la petición.

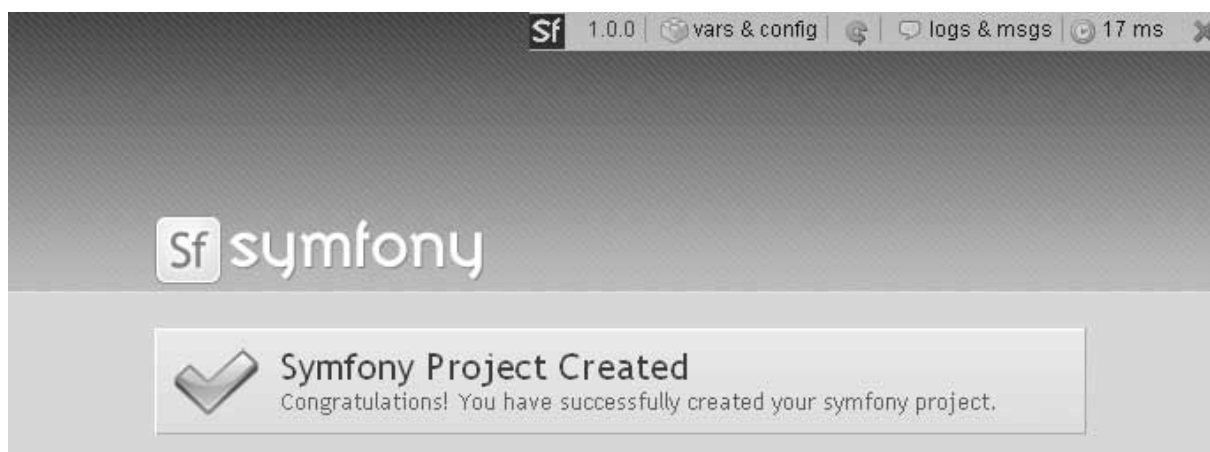


Figura 16.2. La barra de depuración web se muestra en la esquina superior derecha de la ventana del navegador

El color de fondo de la barra de depuración web depende del máximo nivel de los mensajes de log producidos durante la petición. Si ningún mensaje pasa del nivel debug, la barra se muestra con color de fondo gris. Si al menos un mensaje alcanza el nivel err, la barra muestra un color de fondo rojo.

Nota No debe confundirse el modo debug y la barra de depuración web. La barra se puede mostrar incluso cuando el modo debug está desactivado, aunque en este caso, muestra mucha menos información.

Para activar la barra de depuración web en una aplicación, se utiliza la opción `web_debug` del archivo de configuración `settings.yml`. En los entornos de ejecución `prod` y `test`, el valor por defecto de la opción `web_debug` es `off`, por lo que se debe activar manualmente si se necesita. En el entorno de ejecución `dev`, esta opción tiene un valor por defecto de `on`, tal y como muestra el listado 16-10.

Listado 16-10 - Activando la barra de depuración web, en `frontend/config/settings.yml`

```
dev:
  .settings:
    web_debug:          on
```

Cuando se muestra la barra de depuración web, ofrece mucha información:

- Si se pincha sobre el logotipo de Symfony, la barra se oculta. Cuando está minimizada, la barra no oculta los elementos de la página que se encuentran en la esquina superior derecha.
- Como muestra la figura 16-3, cuando se pincha sobre la opción `vars & config`, se muestran los detalles de la petición, de la respuesta, de las opciones de configuración, de las opciones globales y de las propiedades PHP. La línea superior resume el estado de las opciones de configuración más importantes, como el modo debug, la cache y la presencia/ausencia de un acelerador de PHP (su nombre aparece en rojo si está desactivado y en color verde si se encuentra activado).



Figura 16.3. La sección "vars & config" muestra todas las variables y constantes de la petición

- Cuando la cache se encuentra activada, se muestra una flecha verde en la barra de depuración web. Si se pulsa sobre esta flecha, la página se vuelve a procesar entera, independientemente de si se encuentra almacenada en la cache (no obstante, la cache no se vacía al pulsar sobre esta flecha).

- Como muestra la figura 16-4, al pulsar sobre la sección `logs & msgs`, se muestran los mensajes de log para la petición actual. En función de la importancia de los eventos, las líneas se muestran en gris, amarillo o rojo. Mediante los enlaces que se muestran en forma de lista en la parte superior, es posible filtrar los mensajes de log en función de su categoría.

#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/module/action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list')
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****'), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.66 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article/list"
16	View	render "sf_app_dir/modules/article/templates/listSuccess.php"
17	View	decorate content with "sf_app_dir/templates/layout.php"
18	View	render "sf_app_dir/templates/layout.php"

Figura 16.4. La sección "logs & msgs" muestra los mensajes de log de la petición actual

Nota Cuando la acción es el resultado de una redirección, solamente se muestran los mensajes de log de la última petición, por lo que es imprescindible consultar los archivos de log completos para depurar las aplicaciones.

- Si durante el procesamiento de la petición se han ejecutado consultas SQL, se muestra un icono de una base de datos en la barra de depuración web. Si se pulsa sobre este icono, se muestra el detalle de las consultas realizadas, como se muestra en la figura 16-5.
- A la derecha del icono del reloj se muestra el tiempo total de procesamiento requerido por la petición. Como el modo debug y la propia barra de depuración consumen muchos recursos, el tiempo que se muestra es mucho más lento que la ejecución real de la petición. Por tanto, es más importante fijarse en las diferencias de tiempos producidas por los cambios introducidos que en el propio tiempo mostrado. Si se pulsa sobre el icono del reloj, se muestran los detalles del tiempo de procesamiento de cada categoría, tal y como se muestra en la figura 16-6. Symfony muestra el tiempo consumido en las diferentes partes que componen el procesamiento de la petición. Como solamente tiene sentido optimizar el tiempo de procesamiento propio de la petición, no se muestra el tiempo

consumido por el núcleo de Symfony. Esta es la razón por la que la suma de todos los tiempos individuales no es igual al tiempo total mostrado.

- Si se pulsa sobre la X roja a la derecha de la barra, se oculta la barra de depuración web.

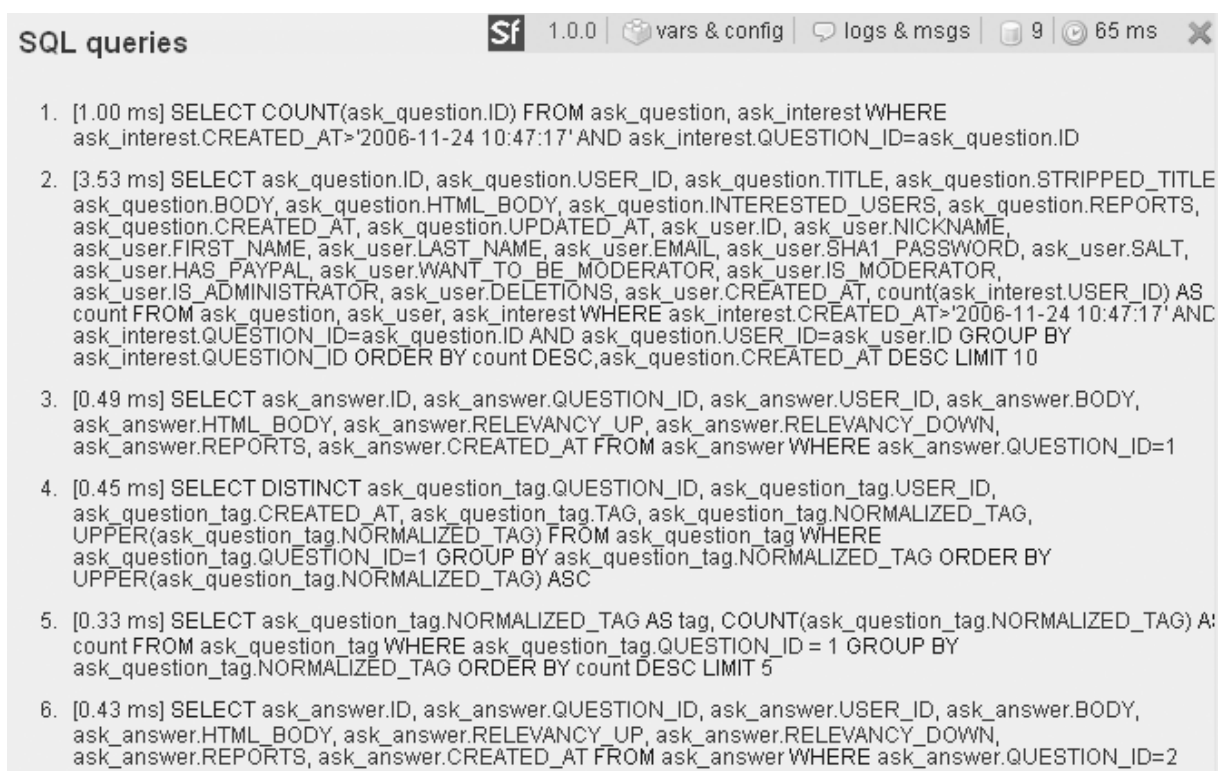


Figura 16.5. La sección de consultas a la base de datos muestra las consultas ejecutadas durante la petición actual

Timers Sf 1.0.0 vars & config logs & msgs 9 609 ms

type	calls	time (ms)
Configuration	14	60.42
Action "question/frontpage"	1	132.43
Database	9	7.56
View "Success" for "question/frontpage"	1	243.24
Partial "question/_question_list"	1	151.49
Partial "question/_question_block"	2	119.74
Partial "question/_interested_user"	2	12.99
Partial "moderator/_question_options"	2	2.80
Component "sidebar/default"	1	0.02
Partial "sidebar/_default"	1	25.62
Partial "tag/_tag_cloud"	1	2.09
Partial "question/_search"	1	0.97
Partial "sidebar/_rss_links"	1	3.08
Partial "sidebar/_moderation"	1	1.32
Partial "sidebar/_administration"	1	1.85

Figura 16.6. El icono del reloj muestra el tiempo de ejecución dividido por categorías

Symfony utiliza la clase `sfTimer` para calcular el tiempo empleado en la configuración, el modelo, la acción y la vista. Utilizando el mismo objeto, se puede calcular el tiempo empleado por un proceso propio y mostrar el resultado junto con el resto de tiempos de la barra de depuración

web. Se trata de algo muy útil cuando se está trabajando en la optimización del rendimiento de la aplicación.

Para inicializar el control del tiempo para un fragmento de código, se utiliza el método `getTimer()`. Este método devuelve un objeto de tipo `sfTimer` y comienza a contar el tiempo. Para detener el avance del contador de tiempo, se invoca el método `addTime()`. En ese instante, se puede obtener el tiempo transcurrido mediante el método `getElapsedTime()` y se muestra automáticamente junto con el resto de tiempos en la barra de depuración web.

```
// Inicializar el contador y empezar a contar el tiempo
$contador = sfTimerManager::getTimer('miContador');

// Otras instrucciones y código
...

// Detener el contador y sumar el tiempo transcurrido
$contador->addTime();

// Obtener el resultado (y detener el contador si no estaba detenido)
$tiempoTranscurrido = $contador->getElapsedTime();
```

La ventaja de asignar un nombre a cada contador, es que se puede utilizar varias veces para acumular diferentes tiempos. Si por ejemplo el contador `miContador` se utiliza en un método que se llama 2 veces en cada petición, la segunda llamada al método `getTimer('miContador')` comienza a contar el tiempo desde donde se quedó la última vez que se llamó a `addTime()`, por lo que el tiempo transcurrido se sumará al tiempo anterior. El método `getCalls()` del contador devuelve el número de veces que ha sido utilizado el contador desde que se inició la petición, y este dato también se muestra en la barra de depuración web.

```
// Obtener el número de veces que ha sido utilizado el contador
$llamadas = $contador->getCalls();
```

Si se utiliza Xdebug, los mensajes de log son mucho más completos. Se guarda en el log todos los archivos PHP y todas las funciones que han sido llamadas, y Symfony integra esta información con su propio log interno. Cada fila de la tabla de mensajes de log dispone de una flecha bidireccional que se puede pulsar para obtener más detalles sobre la petición relacionada. Si algo no va bien, el modo Xdebug es el que más información proporciona para averiguar la causa.

Nota La barra de depuración web no se incluye por defecto en las respuestas de tipo Ajax y en los documentos cuyo Content-Type no es de tipo HTML. Para el resto de las páginas, se puede deshabilitar la barra de depuración web manualmente desde la acción mediante la llamada a `sfConfig::set('sf_web_debug', false)`.

16.2.5. Depuración manual

Aunque muchas veces es suficiente con acceder a los mensajes de log generados por el framework, en ocasiones es mejor poder generar mensajes de log propios. Symfony dispone de utilidades, que se pueden acceder desde las acciones y desde las plantillas, para crear trazas sobre los eventos y/o valores presentes durante la ejecución de la petición.

Los mensajes de log propios aparecen en el archivo de log de Symfony y en la barra de depuración web, como cualquier otro mensaje de Symfony. (El listado 16-5 anterior muestra un ejemplo de la sintaxis de un mensaje de log propio). Los mensajes de log propios se pueden utilizar por ejemplo para comprobar el valor de una variable en una plantilla. El listado 16-11 muestra cómo utilizar la barra de depuración web desde una plantilla para obtener información para el programador (también se puede utilizar el método `$this->logMessage()` desde una acción).

Listado 16-11 - Creando un mensaje de log propio para depurar la aplicación

```
<?php use_helper('Debug') ?>
...
<?php if ($problem): ?>
    <?php log_message('{sfAction} ha pasado por aquí', 'err') ?>
    ...
<?php endif ?>
```

Si se utiliza el nivel `err`, se garantiza que el evento sea claramente visible en la lista de mensajes, como se muestra en la figura 16-7.

#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/:module/:action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list')
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****',), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.35 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article/list"
16	View	render "sf_app_dir\modules\article\templates\listSuccess.php"
17	Action	been there
18	View	decorate content with "sf_app_dir\templates\layout.php"
19	View	render "sf_app_dir\templates\layout.php"

Figura 16.7. Mensaje de log propio en la sección "logs & msgs" de la barra de depuración web

Si no se quiere añadir una línea al log, sino que solo se necesita mostrar un mensaje corto o un valor, se debería utilizar `debug_message` en vez de `log_message`. Este método de la acción (para el que también existe un *helper* con el mismo nombre) muestra un mensaje en la barra de

depuración web, en la parte superior de la sección logs & msgs. El listado 16-12 muestra un ejemplo de uso de esta utilidad.

Listado 16-12 - Mostrando un mensaje en la barra de depuración web

```
// En una acción
$this->debugMessage($mensaje);

// En una plantilla
<?php use_helper('Debug') ?>
<?php debug_message($mensaje) ?>
```

16.3. Utilizando Symfony fuera de la web

En ocasiones, los scripts se ejecutan desde la línea de comandos o mediante una tarea programada. Muchas veces estos scripts deben tener acceso a todas las clases y características de Symfony, por ejemplo para enviar correos electrónicos de forma automática o para actualizar el modelo de datos mediante cálculos y operaciones complejas.

La forma más sencilla de crear un script de este tipo consiste en añadir al script las primeras líneas de los controladores frontales de Symfony para inicializar el framework. Además, estos scripts también pueden utilizar la línea de comandos de Symfony para aprovecharse del procesamiento de los argumentos y de la inicialización automática de la base de datos.

16.3.1. Archivos por lotes

La inicialización completa de Symfony se puede realizar con sólo unas pocas líneas de código PHP, tal y como muestra el listado 16-13.

Listado 16-13 - Ejemplo de archivo por lotes, en lib/miScript.php

```
<?php

require_once(dirname(__FILE__).'../config/ProjectConfiguration.class.php');
$configuration = ProjectConfiguration::getApplicationConfiguration('frontend', 'dev',
true);
sfContext::createInstance($configuration);

// Borra las dos líneas siguientes si no utilizas una base de datos
$dbaseManager = new sfDatabaseManager($configuration);
$dbaseManager->loadConfiguration();

// A continuación añade tu propio código
```

Las líneas del listado anterior se parecen mucho a las primeras líneas de un controlador frontal (ver capítulo 6) porque son procesos similares: inicializar Symfony y procesar la configuración del proyecto y de la aplicación. El método `ProjectConfiguration::getApplicationConfiguration()` requiere tres parámetros:

- El nombre de una aplicación
- El nombre de un entorno de ejecución

- Un valor *booleano* que indica si se activan o no las opciones de depuración de aplicaciones

Para ejecutar el script anterior sólo es necesario llamarlo desde la línea de comandos:

```
| > php lib/miScript.php
```

16.3.2. Creando tareas propias

Una alternativa mejor a la creación de scripts y archivos por lotes es utilizar Symfony para crear nuevas tareas. De la misma forma que utilizas tareas como `cache:clear` y `propel:build-model`, puedes ejecutar tus propias tareas en la línea de comandos mediante `php symfony`. Las ventajas de las tareas propias es que pueden aprovechar el procesamiento de argumentos y opciones de la línea de comandos, pueden mostrar mensajes de ayuda sobre su uso y pueden ampliar las tareas existentes.

Una tarea propia es una clase que hereda de `sfBaseTask` y cuyo código se encuentra en el directorio `lib/task/` de la raíz del proyecto o en el directorio `lib/task` de un plugin. La única restricción es que su nombre debe terminar en `Task.class.php`. El listado 16-14 muestra un ejemplo de tarea propia.

Listado 16-14 - Ejemplo de tarea, en `lib/task/pruebaHolaMundoTask.class.php`

```
class pruebaHolaMundoTask extends sfBaseTask
{
    protected function configure()
    {
        $this->namespace = 'prueba';
        $this->name = 'holaMundo';
        $this->briefDescription = 'Muestra un mensaje de saludo';
    }

    protected function execute()
    {
        // Aquí se incluye el código de la tarea
        $this->log('Hola mundo');
    }
}
```

El código que se incluye en el método `execute()` tiene acceso a todas las librerías de Symfony, como en el caso del script mostrado en la sección anterior. La principal diferencia se produce al ejecutar la tarea:

```
| > php symfony prueba:holaMundo
```

El nombre de la tarea se forma con las propiedades `namespace` y `name` (que son de tipo *protected*) y no se tiene en cuenta el nombre de la clase ni el nombre de los archivos. Además, como la tarea se integra con la línea de comandos de Symfony, se muestra en el listado de tareas cuando se ejecuta el comando:

```
| > php symfony
```

Si no quieres crear el esqueleto de la tarea manualmente, puedes utilizar la tarea `generate:task`. Esta tarea crea todo el código de una tarea vacía y dispone de muchas opciones para personalizar el código generado. Para acceder a sus opciones, simplemente ejecuta el siguiente comando:

```
| > php symfony help generate:task
```

Las tareas pueden aceptar argumentos (parámetros obligatorios y en un orden determinado) y opciones (parámetros opcionales en los que tampoco importa su orden). El listado 16-15 muestra una tarea más compleja que hace uso de todas estas características.

Listado 16-15 - Ejemplo de tarea compleja, en `lib/task/miSegundaTask.class.php`

```
class miSegundaTask extends sfBaseTask
{
    protected function configure()
    {
        $this->namespace      = 'prueba';
        $this->name            = 'miSegundaTask';
        $this->briefDescription = 'Ejemplo de tarea compleja';
        $this->detailedDescription = <<<EOF
La tarea [prueba:miSegundaTask|INFO] realiza algunas cosas interesantes. La puedes
ejecutar de la siguiente manera:

    [php symfony prueba:miSegundaTask frontend|INFO]

Puedes aumentar el nivel de detalle de los mensajes generados mediante la opción
[verbose|COMMENT]:

    [php symfony prueba:miSegundaTask frontend --verbose=on|INFO]
EOF;
        $this->addArgument('aplicacion', sfCommandArgument::REQUIRED, 'El nombre de la
aplicación');
        $this->addOption('verbose', null, sfCommandOption::PARAMETER_REQUIRED, 'Aumenta el
nivel de detalle de los mensajes generados', false);
    }

    protected function execute($arguments = array(), $options = array())
    {
        // Aquí se incluye el código de la tarea
    }
}
```

Nota Si tu tarea quiere acceder a una base de datos, su clase debe heredar de `sfPropelBaseTask` en vez de `sfBaseTask`. De esta forma, la inicialización de la tarea tiene en cuenta que debe cargar todas las clases de Propel. Para crear una conexión con la base de datos en el método `execute()` ejecuta la siguiente instrucción:

```
| $databaseManager = new sfDatabaseManager($this->configuration);
```

Si la configuración de la tarea define los argumentos `application` y `env`, se tienen en cuenta automáticamente cuando se construye la configuración de la tarea, por lo que una tarea puede utilizar cualquiera de las conexiones con bases de datos definidas en el archivo `databases.yml`. Las tareas vacías creadas con la tarea `generate:task` incluyen esta inicialización por defecto.

Una buena forma de descubrir las posibilidades de las tareas consiste en ver el código fuente de las propias tareas de Symfony.

Si creas una tarea con el mismo nombre que alguna tarea existente, tu clase redefine la clase existente. Por lo tanto, un plugin puede redefinir las tareas de Symfony y un proyecto puede redefinir tanto las tareas de Symfony como las tareas de los plugins. Si además se tiene en cuenta que una tarea puede heredar de otra tarea, se comprueba que el mecanismo de línea de comandos de Symfony es realmente flexible.

16.4. Cargando datos en una base de datos

Durante el desarrollo de una aplicación, uno de los problemas recurrentes es el de la carga inicial de datos en la base de datos. Algunos sistemas de bases de datos disponen de soluciones específicas para esta tarea, pero ninguna se puede utilizar junto en el ORM de Symfony. Gracias al uso de YAML y al objeto `sfPropelData`, Symfony puede transferir automáticamente los datos almacenados en un archivo de texto a una base de datos. Aunque puede parecer que crear el archivo de texto con los datos iniciales de la aplicación cuesta más tiempo que insertarlos directamente en la base de datos, a la larga se ahorra mucho tiempo. Se trata de una utilidad muy práctica para la carga automática de datos de prueba para la aplicación.

16.4.1. Sintaxis del archivo de datos

Symfony es capaz de procesar todos los archivos que siguen una sintaxis YAML definida muy simple y que se encuentren en el directorio `data/fixtures/`. Los archivos de datos, también llamados "*fixtures*", se organizan por clases y cada sección de clase utiliza una cabecera con el valor del nombre de la clase. Para cada clase, las filas de datos disponen de una etiqueta que las identifica de forma única y una serie de pares `nombre_campo: valor`. El listado 16-16 muestra un ejemplo de un archivo preparado para cargar sus datos en una base de datos.

Listado 16-16 - Archivo de datos de ejemplo, en `data/fixtures/import_data.yml`

```
Article:                                     ## Crea filas de datos en la tabla blog_article
  first_post:                               ## Etiqueta de la primera fila de datos
    title:      My first memories
    content: |
      For a long time I used to go to bed early. Sometimes, when I had put
      out my candle, my eyes would close so quickly that I had not even time
      to say "I am going to sleep."

  second_post:                               ## Etiqueta de la segunda fila de datos
    title:      Things got worse
    content: |
      Sometimes he hoped that she would die, painlessly, in some accident,
      she who was out of doors in the streets, crossing busy thoroughfares,
      from morning to night.
```

Symfony transforma el nombre indicado para las columnas, en métodos *setter* utilizando la conversión de tipo *camelCase* (la columna `title` se transforma en `setTitle()`, la columna `content` se transforma en `setContent()`, etc.). La ventaja de esta transformación es que se puede definir, por ejemplo, una columna llamada `password` para la que no existe una columna en la

tabla de la base de datos; solamente es necesario definir un método llamado `setPassword()` en el objeto `User` y ya es posible asignar valores a otras columnas de datos en función de este dato, como por ejemplo una columna que guarde la contraseña encriptada.

No es necesario definir el valor de la columna de la clave primaria. Como es un campo cuyo valor se autoincrementa, la capa de base de datos es capaz de determinar su valor.

A las columnas `created_at` tampoco es necesario asignarles un valor, ya que Symfony sabe que a las columnas que se llaman así, les debe asignar la fecha actual del sistema a la hora de crearlas.

16.4.2. Importando los datos

La tarea `propel:data-load` importa los datos de los archivos YAML en una base de datos. Las opciones de conexión con la base de datos se obtienen del archivo de configuración `databases.yml`, por lo que es necesario indicar a la tarea el nombre de una aplicación. Además, es posible indicar el nombre de un entorno de ejecución mediante la opción `--env` (su valor por defecto es `dev`).

```
| > php symfony propel:data-load --env=prod frontend
```

Al ejecutar este comando, se leen todos los archivos de datos YAML del directorio `data/fixtures` y se insertan las filas de datos en la base de datos. Por defecto, se reemplaza todo el contenido existente en la base de datos, aunque si se utiliza la opción llamada `--append`, el comando no borra los datos existentes.

```
| > php symfony propel:data-load --append frontend
```

También es posible especificar otro archivo de datos u otro directorio, indicando su valor como una ruta relativa respecto del directorio del proyecto.

```
| > php symfony propel:data-load frontend --dir[]=data/misfixtures
```

16.4.3. Usando tablas relacionadas

Ahora ya es posible añadir filas de datos a una tabla, pero de esta forma no es posible añadir filas con claves externas que hacen relación a otra tabla. Como los archivos de datos no incluyen la clave primaria, se necesita un método alternativo para relacionar los diferentes registros de datos entre sí.

Volviendo al ejemplo del Capítulo 8, donde la tabla `blog_article` está relacionada con la tabla `blog_comment`, de la forma que se muestra en la figura 16-8.

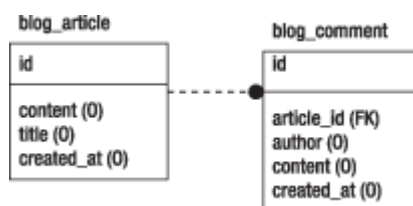


Figura 16.8. Ejemplo de modelo relacional de una base de datos

En esta situación es en la que se utilizan las etiquetas únicas de cada fila de datos. Para añadir un campo de tipo Comment al artículo llamado `first_post`, simplemente es necesario añadir las siguientes líneas del listado 16-17 al archivo de datos `import_data.yml`.

Listado 16-17 - Añadiendo un registro relacionado con otra tabla, en `data/fixtures/import_data.yml`

```
Comment:
  first_comment:
    article_id:  first_post
    author:      Anonymous
    content:     Your prose is too verbose. Write shorter sentences.
```

La tarea `propel:data-load` es capaz de reconocer la etiqueta que se asignó anteriormente al artículo en el archivo `import_data.yml` y es capaz de obtener la clave primaria del registro de tipo `Article` correspondiente en la base de datos, para asignar ese valor al campo `article_id`. No es necesario trabajar con los valores de las columnas de tipo ID, solo es necesario enlazar las filas de datos mediante sus etiquetas, por lo que su funcionamiento es realmente simple.

La única restricción para las filas de datos enlazadas es que los objetos utilizados en una clave externa tienen que estar definidos anteriormente en el archivo; es decir, igual que si se tuvieran que definir uno a uno. Los archivos de datos se procesan desde el principio hasta el final y por tanto, el orden en el que se escriben las filas de datos es muy importante.

A partir de Symfony 1.1, este mecanismo también funciona para las relaciones muchos-a-muchos, en las que dos clases se relacionan a través de una tercera clase. Un ejemplo podría ser una clase `Articulo` que puede tener varios `Autores` y un `Autor` puede tener varios `Articulos`. Normalmente, estas relaciones se resuelven creando una clase `ArticuloAutor` que se corresponde con una tabla llamada `articulo_autor` que tiene las columnas `articulo_id` y `autor_id`. El listado 16-18 muestra un ejemplo de archivo de datos que define una relación de este tipo. Lo más importante es el uso del plural en el nombre de la tabla (`articulo_autors`) que indica que existe una clase intermedia entre las dos clases.

Listado 16-18 - Añadiendo un registro con una relación de tipo muchos-a-muchos, en `data/fixtures/import_data.yml`

```
Autor:
  primer_autor:
    nombre: Juan Pérez
    articulo_autors: [primer_articulo, segundo_articulo]
```

Un solo archivo de datos puede contener la declaración de varias clases diferentes. Sin embargo, si se necesitan insertar muchos datos en muchas tablas diferentes, es posible que el archivo de datos sea demasiado largo como para manejarlo fácilmente.

Como la tarea `propel:data-load` procesa todos los archivos que encuentra en el directorio `fixtures/`, es posible dividir el archivo de datos YAML en otros archivos más pequeños. Lo único que hay que tener en cuenta es que las claves externas obligan a definir un determinado orden al procesar los datos. Para asegurar que los archivos se procesan en el orden adecuado, se

puede añadir un número como prefijo del nombre del archivo, de forma que se procesen en el orden establecido.

```
100_article_import_data.yml
200_comment_import_data.yml
300_rating_import_data.yml
```

16.5. Instalando aplicaciones

Symfony dispone de comandos para sincronizar 2 versiones diferentes de un mismo sitio web. La utilidad de estos comandos es la de poder instalar una aplicación o sitio web desde un servidor de desarrollo hasta un servidor de producción, desde donde los usuarios accederán a la aplicación pública. Este proceso también se conoce como el *"deploy"* de una aplicación, por lo que a veces se utiliza la palabra *"deployar"* (o *"desplegar"*) para referirse a la instalación de una aplicación.

16.5.1. Preparando un proyecto para transferirlo con FTP

La forma habitual de instalar las aplicaciones en los servidores de producción consiste en transferir todos los archivos de la aplicación mediante FTP (o SFTP). Sin embargo, los proyectos desarrollados con Symfony utilizan las librerías internas de Symfony y, salvo que se desarrolle con el archivo de pruebas *sandbox* (lo que no se recomienda) o salvo que los directorios *lib/* y *data/* estén enlazados mediante *svn:externals*, estas librerías no se encuentran dentro de los directorios del proyecto. Independientemente de que se realice una instalación PEAR o se utilicen enlaces simbólicos, trasladar la misma estructura de directorios al servidor de producción suele ser una tarea costosa y no muy sencilla.

Por este motivo, Symfony dispone de una utilidad que *congela* los proyectos, es decir, copia todas las librerías de Symfony necesarias en los directorios *data/*, *lib/* y *web/* del proyecto. Una vez *congelado*, el proyecto se transforma en una aplicación independiente y completamente ejecutable por sí misma, tal y como el entorno de pruebas *sandbox*.

```
| > php symfony project:freeze symfony_data_dir
```

El argumento *symfony_data_dir* es la ruta al directorio *data* de Symfony. Si has instalado Symfony mediante Subversion o mediante un archivo con el código fuente, este directorio coincide con el directorio *lib* de Symfony. Si has instalado Symfony mediante un paquete PEAR, este directorio se encuentra bajo el directorio de datos de PEAR.

Una vez que un proyecto ha sido *congelado*, se puede transferir directamente el directorio raíz completo del proyecto al servidor de producción y funciona sin necesidad de PEAR, enlaces simbólicos o cualquier otro elemento.

Sugerencia En un mismo servidor se pueden ejecutar simultáneamente varios proyectos congelados, cada uno con su propia, e incluso diferente, versión de Symfony.

Para devolver un proyecto a su estado original, se utiliza la tarea *project:unfreeze* (*descongelar*). Esta tarea borra los directorios *data/symfony/*, *lib/symfony/* y *web/sf/*.

```
| > php symfony project:unfreeze
```

Si antes de *congelar* el proyecto existían enlaces simbólicos, Symfony es capaz de reconocerlos y al *descongelar* el proyecto, vuelve a crear los enlaces simbólicos originales.

16.5.2. Usando rsync para transferir archivos incrementalmente

Cuando se realiza el primer traspaso de la aplicación web, es útil transferir mediante FTP (o SFTP) el directorio raíz completo del proyecto, pero cuando se trata de actualizar una aplicación para la que solamente se han modificado unos pocos archivos, la solución mediante FTP no es la ideal. Si se utiliza FTP, o se vuelve a transferir completo el proyecto, con el consiguiente gasto de tiempo y ancho de banda, o se accede manualmente a todos los directorios con archivos modificados y se suben de uno en uno. Este último método, no solo es costoso en tiempo, sino que es muy propenso a cometer errores. Además, el sitio web puede estar no disponible o puede mostrar muchos errores durante el traspaso de las modificaciones.

La solución que propone Symfony es el uso de la herramienta de sincronización rsync mediante SSH. Rsync (<http://samba.anu.edu.au/rsync/>) es una utilidad de la línea de comandos que permite realizar una transferencia incremental de archivos de forma muy rápida, además de que es una herramienta de software libre. En una transferencia incremental, solamente se transfieren los datos modificados. Si un archivo no ha sido modificado desde la última sincronización, no se vuelve a enviar al servidor. Si un archivo solamente tiene un cambio parcial, solamente se envían los cambios realizados. La principal ventaja de rsync es que las sincronizaciones requieren el envío de muy pocos datos y por tanto, son muy rápidas.

Symfony utiliza SSH conjuntamente con rsync para hacer más segura la transferencia de datos. La mayoría de servicios de hosting soportan el uso de SSH para aportar más seguridad a la transferencia de archivos hasta sus servidores.

El cliente SSH utilizado por Symfony utiliza las opciones de conexión del archivo `config/properties.ini`. El listado 16-19 muestra un ejemplo de las opciones de conexión para un servidor de producción. Antes de realizar la sincronización de la aplicación, se deben establecer las opciones de conexión en este archivo. También es posible definir una opción llamada `parameters` para utilizar parámetros propios con rsync.

Listado 16-19 - Opciones de conexión para la sincronización con un servidor, en `miproyecto/config/properties.ini`

```
[symfony]
  name=miproyecto

[production]
  host=frontend.example.com
  port=22
  user=myuser
  dir=/home/myaccount/miproyecto/
```

Nota No debe confundirse el servidor de producción (que es el servidor definido en el archivo `properties.ini` del proyecto) con el entorno de producción (el controlador frontal y la configuración que se utiliza en producción).

Como la sincronización de `rsync` mediante SSH requiere de varios comandos, y la sincronización suele ocurrir muchas veces durante la vida de una aplicación, Symfony automatiza esta tarea mediante un único comando:

```
| > php symfony project:deploy production
```

El comando anterior ejecuta el comando `rsync` en el modo de prueba; es decir, muestra los archivos que tienen que ser sincronizados, pero no los sincroniza realmente. Para realizar la sincronización, se debe indicar explícitamente mediante la opción `--go`.

```
| > php symfony project:deploy production --go
```

No debe olvidarse borrar la cache en el servidor de producción después de la sincronización.

Sugerencia En ocasiones, se producen errores en el servidor de producción que no existían en el servidor de desarrollo. El 90% de las veces el problema reside en una diferencia en las versiones de las aplicaciones (de PHP, del servidor web o de la base de datos) o en la configuración de la aplicación. Para evitar sorpresas desagradables, se debe definir la configuración de PHP del servidor de producción en un archivo llamado `php.yml`, para poder comprobar que el entorno de desarrollo aplica las mismas opciones. El Capítulo 19 incluye más información sobre este archivo de configuración.

Antes de subir la aplicación al servidor de producción, es necesario asegurarse de que está lista para ser utilizada por los usuarios. Antes de instalar la aplicación en el servidor de producción, es recomendable comprobar que se ha completado lo siguiente:

Las páginas de error deberían mostrar un aspecto integrado con el del resto de la aplicación. El Capítulo 19 explica cómo personalizar las páginas del error 500, del error 400 y las de las páginas relacionadas con la seguridad. La sección "Administrando una aplicación en producción" explica, más adelante en este capítulo, cómo personalizar las páginas que se muestran cuando la aplicación no está disponible.

El módulo `default` se debe eliminar del array `enabled_modules` del archivo `settings.yml`, de modo que no se muestren por error páginas del propio Symfony.

El mecanismo de gestión sesiones utiliza una cookie para el navegador del usuario y esta cookie se llama `symfony`. Antes de instalar la aplicación en producción, puede ser una buena idea cambiarle el nombre para no mostrar que la aplicación está desarrollada con Symfony. El Capítulo 6 explica cómo modificar el nombre de la cookie en el archivo `factories.yml`.

Por defecto, el archivo `robots.txt` del directorio `web/` está vacío. Normalmente, es una buena idea modificar este archivo para indicar a los buscadores las partes de la aplicación que pueden acceder y las partes que deberían evitar. También se suele utilizar este archivo para excluir ciertas URL de la indexación realizada por los buscadores, como por ejemplo las URL que consumen mucho tiempo de proceso o las páginas que no interesa indexar.

Los navegadores más modernos buscan un archivo llamado `favicon.ico` cuando el usuario accede por primera vez a la aplicación. Este archivo es el icono que representa a la aplicación en la barra de direcciones y en la carpeta de favoritos. Además de que este icono ayuda a completar el aspecto de la aplicación, evita que se produzcan muchos errores de tipo 404 cuando los navegadores lo solicitan y no se encuentra disponible.

16.5.3. Ignorando los archivos innecesarios

Cuando se sincroniza un proyecto Symfony con un servidor de producción, algunos archivos y directorios no deberían transferirse:

- Todos los directorios del versionado del código (.svn/, CVS/, etc.) y su contenido, solamente es necesario para el desarrollo e integración de la aplicación.
- El controlador frontal del entorno de desarrollo no debería ser accesible por los usuarios finales. Las herramientas de depuración y de log disponibles en este controlador frontal penalizan el rendimiento de la aplicación y proporcionan mucha información sobre las variables internas utilizadas por las acciones. Siempre debería eliminarse este controlador frontal en la aplicación pública.
- Los directorios cache/ y log/ del proyecto no deben borrarse cada vez que se realiza una sincronización. Estos directorios también deberían ignorarse. Si se dispone de un directorio llamado stats/, también debería ignorarse.
- Los archivos subidos por los usuarios tampoco deberían transferirse. Una de las buenas prácticas recomendadas por Symfony es la de guardar los archivos subidos por los usuarios en el directorio web/uploads/. De esta forma, se pueden excluir todos estos archivos simplemente ignorando un directorio durante el traspaso de la aplicación.

Para excluir los archivos en las sincronizaciones de rsync, se edita el archivo rsync_exclude.txt que se encuentra en el directorio miproyecto/config/. Cada fila de ese archivo debe contener el nombre de un archivo, el nombre de un directorio o un patrón con comodines *. La estructura de archivos de Symfony está organizada de forma lógica y diseñada de forma que se minimice el número de archivos o directorios que se deben excluir manualmente de la sincronización. El listado 16-20 muestra un ejemplo.

Listado 16-20 - Ejemplo de exclusiones en una sincronización rsync, en miproyecto/config/rsync_exclude.txt

```
.svn  
/cache/*  
/log/*  
/stats/*  
/web/uploads/*  
/web/frontend_dev.php
```

Nota Los directorios cache/ y log/ no deben sincronizarse con el servidor de producción, pero sí que deben existir en el servidor de producción. Si la estructura de directorios y archivos del proyecto miproyecto/ no los contiene, deben crearse manualmente.

16.5.4. Administrando una aplicación en producción

El comando más utilizado en los servidores de producción es cache:clear. Cada vez que se actualiza Symfony o el proyecto, se debe ejecutar esta tarea (por ejemplo después de ejecutar la tarea project:deploy) y también cada vez que se modifica la configuración en producción.

```
| > php symfony cache:clear
```

Sugerencia Si en el servidor de producción no está disponible la línea de comandos de Symfony, se puede borrar la cache manualmente borrando todos los contenidos del directorio `cache/`.

También es posible deshabilitar temporalmente la aplicación, por ejemplo cuando se necesita actualizar una librería o cuando se tiene que actualizar una gran cantidad de datos.

```
| > php symfony project:disable NOMBRE_APLICACION NOMBRE_ENTORNO
```

Por defecto, una aplicación deshabilitada muestra la página `$sf_symfony_lib_dir/exception/data/unavailable.php`. No obstante, si creas una página llamada `unavailable.php` en el directorio `config/` del proyecto, Symfony utiliza tu página en vez de la página predefinida.

La tarea `project:enable` vuelve a habilitar la aplicación y borra su cache.

```
| > php symfony project:enable NOMBRE_APLICACION NOMBRE_ENTORNO
```

Si se establece el valor `on` a la opción `check_lock` en el archivo `settings.yml`, Symfony bloquea el acceso a la aplicación mientras se borra la cache, y todas las peticiones recibidas mientras se borra la cache se redirigen a una página que muestra que la aplicación está temporalmente no disponible. Se trata de una opción muy recomendable cuando el tamaño de la cache es muy grande y el tiempo empleado para borrarla es mayor que unos milisegundos y el tráfico de usuarios de la aplicación es elevado.

Esta página que indica que la aplicación no está disponible no es la misma que la que se muestra cuando se ejecuta la tarea `project:disable`. El parámetro `check_lock` está desactivado por defecto porque tiene un ligero impacto negativo en el rendimiento.

La tarea `project:clear-controllers` elimina todos los controladores frontales del directorio `web/` que no sean los controladores frontales utilizados en el entorno de producción. Si no se incluyen los controladores frontales de desarrollo en el archivo `rsync_exclude.txt`, este comando garantiza que no se sube al servidor una puerta trasera que revele información interna sobre la aplicación.

```
| > php symfony project:clear-controllers
```

Los permisos de los archivos y directorios del proyecto pueden cambiarse si se realiza un *checkout* desde un repositorio de Subversion. La tarea `project:permissions` arregla los permisos de los directorios y cambia por ejemplo los permisos de `log/` y `cache/` a un valor de `0777` (estos directorios deben tener permiso de escritura para que el framework funcione correctamente).

```
| > php symfony project:permissions
```

16.6. Resumen

Mediante los archivos de log de PHP y los de Symfony, es posible monitorizar y depurar las aplicaciones fácilmente. Durante el desarrollo de la aplicación, el modo debug, las excepciones y la barra de depuración web ayudan a localizar la causa de los problemas. Para facilitar la depuración de la aplicación, es posible incluso insertar mensajes propios en el archivo de log y en la barra de depuración web.

La interfaz de línea de comandos dispone de muchas utilidades para facilitar la gestión y administración de las aplicaciones durante las fases de desarrollo y de producción. Las tareas para cargar de forma masiva datos en la base de datos, la *congelación* de los proyectos y la sincronización de aplicaciones entre servidores, son tareas que ahorran mucho tiempo.

Capítulo 17. Personalizar Symfony

Antes o después, algún proyecto deberá modificar el comportamiento de Symfony. Sea una modificación del comportamiento de una clase o sea una nueva característica que hay que añadir al framework, el momento en el que es necesario modificar Symfony llegará de forma inevitable, ya que todos los clientes para los que se desarrollan aplicaciones tienen requerimientos muy específicos que ningún framework puede predecir.

De hecho, como esta situación es tan común, Symfony dispone de un mecanismo para extender las clases en tiempo de ejecución, algo mucho más avanzado que una simple herencia de clases. Incluso es posible reemplazar las clases del núcleo de Symfony por tus propias clases, utilizando las opciones de las factorías utilizadas por Symfony (las factorías se basan en el patrón de diseño "factories"). Una vez realizadas las modificaciones, se pueden encapsular en forma de plugin para poder reutilizarlas en otras aplicaciones o por parte de otros programadores de Symfony.

17.1. Eventos

Una de las limitaciones actuales de PHP más molestas es que una clase no puede heredar de más de una clase. Además, tampoco se pueden añadir nuevos métodos a una clase ya existente y no se pueden redefinir los métodos existentes. Para paliar estas dos limitaciones y para hacer el framework realmente modificable, Symfony proporciona un sistema de eventos inspirado en el centro de notificación de Cocoa (<http://developer.apple.com/documentation/Cocoa/Conceptual/Notifications/Articles/NotificationCenter.html>), que a su vez se basa en el patrón de diseño Observer (http://es.wikipedia.org/wiki/Observer_%28patr%C3%B3n_de_dise%C3%B1o%29).

17.1.1. Comprendiendo los eventos

Algunas clases de Symfony *notifican un evento* en varios momentos de su ejecución. Por ejemplo cuando un usuario modifica su cultura, el objeto del usuario notifica un evento de tipo `change_culture`. Explicándolo sin palabras técnicas, este evento es como si el objeto le dijera al proyecto *"Acabo de cambiar la cultura del usuario. Si necesitas hacer algo al respecto, este es el momento"*.

Cuando se produce un evento, la aplicación puede responder realizando cualquier proceso. Cuando el usuario modifica su cultura, la aplicación podría responder a la notificación del evento `change_culture` guardando la cultura del usuario en una base de datos para recordar posteriormente la cultura preferida del usuario. Para ello, la aplicación tiene que registrar un *event listener*, que consiste en una función que responde a los eventos producidos. El listado 17-1 muestra cómo registrar un *listener* que responda al evento `change_culture` del usuario:

Listado 17-1 - Registrando un *event listener*

```
$dispatcher->connect('user.change_culture', 'modificaCulturaUsuario');  
  
function modificaCulturaUsuario(sfEvent $evento)
```

```
{
    $usuario = $evento->getSubject();
    $cultura = $evento['culture'];

    // Código que utiliza la cultura del usuario
}
```

La gestión de los eventos y del registro de *listeners* se realiza mediante un objeto especial llamado *event dispatcher*. Este objeto está disponible en cualquier parte del código de la aplicación mediante el *singleton* `sfContext` y la mayoría de objetos de Symfony incluyen un método llamado `getDispatcher()` que permite tener acceso directo a ese objeto. El método `connect()` del *dispatcher* se utiliza para registrar cualquier elemento ejecutable de PHP (el método de una clase o una función) de forma que se ejecute cada vez que se produzca el evento. El primer argumento de `connect()` es el identificador del evento, que es una cadena de texto formada por un *namespace* y el nombre del evento. El segundo argumento es el nombre del elemento ejecutable de PHP.

Las funciones registradas con el *event dispatcher* simplemente esperan a que se produzca el evento para el que han sido registradas. El *event dispatcher* guarda un registro de todos los *listeners* para saber cuáles se deben ejecutar cuando se notifique un evento. Cuando se ejecutan estos métodos o funciones, el *dispatcher* les pasa como argumento un objeto de tipo `sfEvent`.

El objeto del evento almacena información sobre el evento que ha sido notificado. El elemento que ha notificado el evento se puede obtener mediante el método `getSubject()` y los parámetros del evento se pueden acceder mediante el propio objeto del evento utilizando la sintaxis de los arrays. Para obtener por ejemplo el parámetro `culture` de `sfUser` cuando se notifica el evento `user.change_culture`, se utiliza `$evento['culture']`.

En resumen, el sistema de eventos permite añadir nuevas opciones a las clases existentes e incluso permite modificar sus métodos en tiempo de ejecución sin necesidad de utilizar la herencia de clases.

Nota Symfony 1.0 utiliza un mecanismo similar pero con una sintaxis muy diferente. En vez de realizar llamadas a los métodos del *event dispatcher*, en Symfony 1.0 se realizan llamadas a métodos estáticos de la clase `sfMixer` para registrar y notificar eventos. Aunque las llamadas a `sfMixer` se han declarado obsoletas, todavía funcionan correctamente en Symfony 1.1.

17.1.2. Notificando un evento

De la misma forma que las clases de Symfony notifican sus eventos, puedes hacer que tus clases sean fácilmente modificables notificando algunos de sus eventos más importantes. Imagina que tu aplicación realiza peticiones a varios servicios web externos y que has creado una clase llamada `sfRestRequest` para encapsular toda la lógica de tipo REST de estas peticiones. Una buena práctica consiste en notificar un evento cada vez que la clase realice una nueva petición. De esta forma, en el futuro será mucho más fácil añadirle funcionalidades como una cache y un sistema de logs. El listado 17-2 muestra el código que es necesario añadir a un método existente llamado `obtener()` para que notifique un evento.

Listado 17-2 - Notificando un evento


```

class sfRestRequest
{
    protected $dispatcher = null;

    public function __construct(sfEventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    /**
     * Realiza una petición a un servicio web externo
     */
    public function obtener($uri, $parametros = array())
    {
        // Notificar el inicio de la petición
        $this->dispatcher->notify(new sfEvent($this, 'peticion_rest.preparar_peticion',
        array(
            'uri'          => $uri,
            'parameters' => $parametros
        )));

        // Realizar la petición y guardar el resultado en una variable llamada $$resultado
        // ...

        // Notificar la finalización de la petición
        $this->dispatcher->notify(new sfEvent($this, 'peticion_rest.peticion_finalizada',
        array(
            'uri'          => $uri,
            'parametros' => $parametros,
            'resultado' => $resultado
        )));

        return $resultado;
    }
}

```

El método `notify()` del *event dispatcher* requiere como argumento un objeto de tipo `sfEvent`, el mismo tipo de objeto que se pasa a los *event listeners*. Este objeto siempre incluye una referencia al elemento que realiza la notificación (ese es el motivo por el que la instancia del objeto se inicializa con `$this`) y un identificador del evento. De forma opcional también admite un array asociativo de parámetros que permite a los *listeners* interactuar con la lógica del notificador del evento.

Sugerencia Solamente las clases que notifican eventos se pueden modificar mediante el sistema de eventos. Por lo tanto, aunque no estés seguro de si en el futuro necesitarás modificar una clase en tiempo de ejecución, es conveniente que añadas notificaciones en al menos los métodos principales de tus clases.

17.1.3. Notificando un evento hasta que lo procese un listener

El método `notify()` asegura que todos los *listeners* registrados para un evento se van a ejecutar cuando se produzca el evento. Sin embargo, en ocasiones es necesario que un *listener* impida la notificación del evento de forma que ya no se ejecute ninguno de los restantes *listeners*

registrados para ese evento. En este último caso se utiliza el método `notifyUntil()` en vez de `notify()`. De esta forma, el *dispatcher* ejecuta todos los *listeners* hasta que alguno de ellos devuelva un valor `true` y detenga la notificación del evento. Explicándolo sin palabras técnicas, este evento es como si el *listener* le dijera al proyecto *"Ya me encargo yo de responder a este evento, por lo que no se lo notifiques a nadie más"*. El listado 17-3 muestra cómo utilizar esta técnica junto con el método mágico `__call()` para añadir métodos en tiempo de ejecución a una clase existente.

Listado 17-3 - Notificando un evento hasta que un *listener* devuelva true

```
class sfRestRequest
{
    // ...

    public function __call($metodo, $argumentos)
    {
        $evento = $this->dispatcher->notifyUntil(new sfEvent($this,
        'peticion_rest.metodo_no_disponible', array(
            'metodo'      => $metodo,
            'argumentos' => $argumentos
        )));
        if (!$evento->isProcessed())
        {
            throw new sfException(sprintf('Se ha invocado un método que no existe %s::%s.',
            get_class($this), $metodo));
        }

        return $evento->getReturnValue();
    }
}
```

Un *event listener* que se haya suscrito al evento `peticion_rest.metodo_no_disponible` puede comprobar el `$metodo` invocado para decidir si se encarga de el o decide pasarlo al siguiente *event listener*. El listado 17-4 muestra como una clase externa añade los métodos `put()` y `delete()` en la clase `sfRestRequest` en tiempo de ejecución utilizando este truco.

Listado 17-4 - Manejando un evento de tipo `notifyUntil`

```
class frontendConfiguration extends sfApplicationConfiguration
{
    public function configure()
    {
        // ...

        // Registrar el listener
        $this->dispatcher->connect('peticion_rest.metodo_no_disponible',
        array('sfRestRequestExtension', 'listenerMetodoNoDisponible'));
    }
}

class sfRestRequestExtension
{
    static public function listenerMetodoNoDisponible(sfEvent $evento)
```

```

{
    switch ($evento['metodo'])
    {
        case 'put':
            self::put($evento->getSubject(), $evento['argumentos'])

            return true;
        case 'delete':
            self::delete($evento->getSubject(), $evento['argumentos'])

            return true;
        default:
            return false;
    }
}

static protected function put($peticionREST, $argumentos)
{
    // Realizar la petición PUT y guardar el resultado en la variable $resultado
    // ...

    $evento->setReturnValue($resultado);
}

static protected function delete($peticionREST, $argumentos)
{
    // Realizar la petición DELETE y guardar el resultado en la variable $resultado
    // ...

    $evento->setReturnValue($resultado);
}
}

```

El método `notifyUntil()` permite realizar con PHP la herencia múltiple entre clases, más conocida como *mixins* y que consisten en añadir métodos de varias clases en otra clase existente. Ahora es posible inyectar, en tiempo de ejecución, nuevos métodos en los objetos que no se pueden modificar mediante la herencia de clases. Lo mejor de todo es que si utilizas Symfony ya no estás limitado por las características orientadas a objetos de PHP.

Sugerencia Como el primer *listener* que se encarga de un evento de tipo `notifyUntil()` evita que el evento siga notificándose, es importante conocer el orden en el que se ejecutan los *listeners*. El orden que se sigue es el mismo en el que fueron registrados, por lo que el primer *listener* registrado es el primer *listener* que se ejecuta.

En la práctica es difícil que el orden en el que se ejecutan los *listeners* sea un problema. Por lo tanto, si crees que dos *listeners* pueden entrar en conflicto para un determinado evento, es probable que tu clase tenga que notificar varios eventos, por ejemplo uno al principio y otro al final de la ejecución del método.

Por último, si los eventos añaden nuevos métodos a las clases existentes, utiliza nombres únicos de forma que no entren en conflicto con otros métodos añadidos en tiempo de ejecución. Una buena práctica en este sentido consiste en prefijar el nombre de los métodos con el nombre de la clase del *listener*.

17.1.4. Modificando el valor de retorno de un método

Obviamente, los *listener* no sólo pueden utilizar la información que reciben desde el evento, sino que también la pueden modificar para alterar la lógica original del notificador del evento. Para conseguirlo, se utiliza el método `filter()` del *event dispatcher* en vez del método `notify()`. En este caso, todos los *listeners* se invocan con dos parámetros: el objeto que representa al evento y el valor que se va a filtrar. Los *listeners* deben devolver un valor, que puede ser el mismo o completamente diferente. El listado 17-5 muestra cómo utilizar el método `filter()` para filtrar la respuesta recibida de un servicio web de modo que se puedan procesar los caracteres especiales.

Listado 17-5 - Notificando y procesando un evento con filtro

```
class sfRestRequest
{
    // ...

    /**
     * Realiza una petición a un servicio web externo
     */
    public function obtener($uri, $parametros = array())
    {
        // Realizar la petición y guardar el resultado en una variable llamada $$resultado
        // ...

        // Notificar la finalización de la petición
        return $this->dispatcher->filter(new sfEvent($this,
        'peticion_rest.filtrar_respuesta', array(
            'uri' => $uri,
            'parametros' => $parametros,
        )), $resultado));
    }
}

// Aplicar el mecanismo de escape a la respuesta del servicio web
$dispatcher->connect('peticion_rest.filtrar_respuesta', 'rest_htmlespecialchars');

function rest_htmlespecialchars(sfEvent $evento, $resultado)
{
    return htmlspecialchars($resultado, ENT_QUOTES, 'UTF-8');
}
```

17.1.5. Eventos predefinidos

Muchas clases de Symfony incluyen varios eventos, lo que permite modificar las funcionalidades del framework sin tener que modificar sus clases. La tabla 17-1 muestra un listado completo de todos estos eventos junto con su tipo y sus argumentos.

Tabla 17-1 - Eventos de Symfony

Namespace	Nombre	Tipo	Notificadores	Argumentos
application	log	notify	Muchas clases	prioridad

application	throw_exception	notifyUntil	sfException	-
command	log	notify	Las clases sfCommand*	prioridad
command	pre_command	notifyUntil	sfTask	argumentos, opciones
command	post_command	notify	sfTask	-
command	filter_options	filter	sfTask	command_manager
configuration	method_not_found	notifyUntil	sfProjectConfiguration	método, argumentos
component	method_not_found	notifyUntil	sfComponent	método, argumentos
context	load_factories	notify	sfContext	-
controller	change_action	notify	sfController	módulo, acción
controller	method_not_found	notifyUntil	sfController	método, argumentos
controller	page_not_found	notify	sfController	módulo, acción
plugin	pre_install	notify	sfPluginManager	canal, plugin, is_package
plugin	post_install	notify	sfPluginManager	canal, plugin
plugin	pre_uninstall	notify	sfPluginManager	canal, plugin
plugin	post_uninstall	notify	sfPluginManager	canal, plugin
request	filter_parameters	filter	sfWebRequest	path_info
request	method_not_found	notifyUntil	sfRequest	método, argumentos
response	method_not_found	notifyUntil	sfResponse	método, argumentos
response	filter_content	filter	sfResponse	-
routing	load_configuration	notify	sfRouting	-
task	cache.clear	notifyUntil	sfCacheClearTask	aplicación, tipo, entorno
template	filter_parameters	filter	sfViewParameterHolder	-
user	change_culture	notify	sfUser	cultura
user	method_not_found	notifyUntil	sfUser	método, argumentos
view	configure_format	notify	sfView	formato, respuesta, petición
view	method_not_found	notifyUntil	sfView	método, argumentos
view.cache	filter_content	filter	sfViewCacheManager	respuesta, uri, nuevo

Puedes registrar todos los *listeners* que necesites para cada uno de los eventos predefinidos. Lo único que debes tener en cuenta es que los métodos o funciones PHP que registres deben devolver un valor *booleano* para los eventos de tipo *notifyUntil* y deben devolver el valor filtrado en los eventos de tipo *filter*.

Como se puede comprobar en la tabla anterior, los espacios de nombres o *namespaces* de los eventos no siempre coinciden con la función de la clase. Por ejemplo todas las clases de Symfony notifican el evento `application.log` cuando quieren guardar algo en los archivos de log (y también en la barra de depuración web):

```
| $dispatcher->notify(new sfEvent($this, 'application.log', array($mensaje)));
```

Las clases propias de tu proyecto también pueden notificar eventos de Symfony siempre que lo necesiten.

17.1.6. ¿Dónde se registran los listeners?

Los *event listeners* se deben registrar lo antes posible durante la ejecución de una petición. En la práctica, el mejor sitio para registrar los *event listeners* es la clase de configuración de la aplicación. Esta clase dispone de una referencia al *event dispatcher* que se puede utilizar en el método `configure()`. El listado 17-6 muestra cómo registrar un *listener* para uno de los eventos de tipo `petition_rest` de los ejemplos anteriores.

Listado 17-6 - Registrando un *listener* en la clase de configuración de la aplicación, en `apps/frontend/config/ApplicationConfiguration.class.php`

```
class frontendConfiguration extends sfApplicationConfiguration
{
    public function configure()
    {
        $this->dispatcher->connect('petition_rest.metodo_no_disponible',
        array('sfRestRequestExtension', 'listenerMetodoNoDisponible'));
    }
}
```

Los plugins, que se explican más adelante en este capítulo, pueden registrar sus propios *event listeners* en el script `config/config.php` de cada plugin. Este script se ejecuta durante la inicialización de la aplicación y permite acceder al *event dispatcher* mediante `$this->dispatcher`.

Los comportamientos de Propel, que se describieron en el capítulo 8, utilizan el sistema de eventos. Los comportamientos en realidad utilizan el sistema de eventos de Symfony 1.0, aunque eso no es lo importante. Los comportamientos encapsulan el registro y el procesamiento de los eventos para poder extender los objetos generados para Propel, tal y como se explica en el siguiente ejemplo.

Los objetos Propel correspondientes a las tablas de la base de datos tienen un método llamado `delete()`, que se puede utilizar para borrar ese registro de la base de datos. Sin embargo, si estás trabajando con una clase llamada *Factura*, es probable que quieras modificar ese método `delete()` para que no borre el registro sino que simplemente modifique el valor de una columna

llamada borrada. Además, los métodos que obtienen registros (`doSelect()`, `retrieveByPk()`) tendrían en cuenta el valor de esa columna. Además, habría que añadir un método llamado `forzarBorrado()` que permita realmente borrar el registro de la base de datos. Todos estos cambios se pueden juntar en una clase, llamada por ejemplo `ParanoidBehavior` (*comportamiento paranoico*). La clase `Factura` resultante hereda de la clase `BaseFactura` de Propel y se le inyectan en tiempo de ejecución los métodos de la clase `ParanoidBehavior`.

Como se ha visto, un comportamiento es en realidad un *mixin* sobre un objeto Propel. Además, el término *comportamiento* en Symfony implica que el *mixin* se distribuye en forma de plugin. De hecho, la clase `ParanoidBehavior` anterior es realmente un plugin de Symfony llamado `sfPropelParanoidBehaviorPlugin` (<http://trac.symfony-project.com/wiki/sfPropelParanoidBehaviorPlugin>).

El uso de comportamientos implica que los objetos Propel generados deben notificar muchos eventos. Como este comportamiento penaliza el rendimiento de la aplicación si no se utilizan comportamientos, los eventos están deshabilitados por defecto. Para utilizar los comportamientos de Propel, tienes que activarlos cambiando el valor de la propiedad `propel.builder.addBehaviors` a `true` en el archivo `propel.ini` y después tienes que volver a construir las clases del modelo.

17.2. Factorías

Una factoría consiste en la definición de una clase que realiza una determinada tarea. Symfony utiliza las factorías en su funcionamiento interno, como por ejemplo para los controladores y para las sesiones. Cuando el framework necesita por ejemplo crear un nuevo objeto para una petición, busca en la definición de la factoría el nombre de la clase que se debe utilizar para esta tarea. Como la definición por defecto de la factoría para las peticiones es `sfWebRequest`, Symfony crea un objeto de esta clase para tratar con las peticiones. La principal ventaja de utilizar las definiciones de las factorías es que es muy sencillo modificar las características internas de Symfony: simplemente es necesario modificar la definición de la factoría y Symfony utiliza la clase propia indicada en vez de la clase por defecto.

Las definiciones para las factorías se guardan en el archivo de configuración `factories.yml`. El listado 17-7 muestra el contenido por defecto de ese archivo. Cada definición consta del nombre de una clase y opcionalmente, de una serie de parámetros. Por ejemplo, la factoría para el almacenamiento de la sesión (que se indica bajo la clave `storage:`) utiliza un parámetro llamado `session_name` para establecer el nombre de la cookie que se crea para el lado del cliente, de forma que se puedan realizar sesiones persistentes.

Listado 17-7 - Archivo por defecto para las factorías, en `frontend/config/factories.yml`

```
prod:
  logger:
    class:  sfNoLogger
    param:
      level:  err
      loggers: ~
```

```

cli:
  controller:
    class: sfConsoleController
  request:
    class: sfConsoleRequest
  response:
    class: sfConsoleResponse

test:
  storage:
    class: sfSessionTestStorage
  param:
    session_path: %SF_TEST_CACHE_DIR%/sessions

#all:
#  controller:
#    class: sfFrontWebController
#
#  request:
#    class: sfWebRequest
#    param:
#      formats:
#        txt: text/plain
#        js: [application/javascript, application/x-javascript, text/javascript]
#        css: text/css
#        json: [application/json, application/x-json]
#        xml: [text/xml, application/xml, application/x-xml]
#        rdf: application/rdf+xml
#        atom: application/atom+xml
#
#  response:
#    class: sfWebResponse
#    param:
#      logging: %SF_LOGGING_ENABLED%
#      charset: %SF_CHARSET%
#
#  user:
#    class: myUser
#    param:
#      timeout: 1800
#      logging: %SF_LOGGING_ENABLED%
#      use_flash: true
#      default_culture: %SF_DEFAULT_CULTURE%
#
#  storage:
#    class: sfSessionStorage
#    param:
#      session_name: symfony
#
#  view_cache:
#    class: sfFileCache
#    param:
#      automatic_cleaning_factor: 0
#      cache_dir: %SF_TEMPLATE_CACHE_DIR%
#      lifetime: 86400

```



```

#     prefix:                %SF_APP_DIR%
#
# i18n:
#   class: sfI18N
#   param:
#     source:                XLIFF
#     debug:                 off
#     untranslated_prefix:   "[T]"
#     untranslated_suffix:   "[/T]"
#     cache:
#       class: sfFileCache
#       param:
#         automatic_cleaning_factor: 0
#         cache_dir:             %SF_I18N_CACHE_DIR%
#         lifetime:              86400
#         prefix:                %SF_APP_DIR%
#
# routing:
#   class: sfPatternRouting
#   param:
#     load_configuration: true
#     suffix:              .
#     default_module:      default
#     default_action:      index
#     variable_prefixes:   [':']
#     segment_separators:  ['/', '.']
#     variable_regex:      '[_w\d_]+'
#     debug:                %SF_DEBUG%
#     logging:              %SF_LOGGING_ENABLED%
#     cache:
#       class: sfFileCache
#       param:
#         automatic_cleaning_factor: 0
#         cache_dir:             %SF_CONFIG_CACHE_DIR%/routing
#         lifetime:              31556926
#         prefix:                %SF_APP_DIR%
#
# logger:
#   class: sfAggregateLogger
#   param:
#     level: debug
#     loggers:
#       sf_web_debug:
#         class: sfWebDebugLogger
#         param:
#           condition:      %SF_WEB_DEBUG%
#           xdebug_logging: true
#       sf_file_debug:
#         class: sfFileLogger
#         param:
#           file: %SF_LOG_DIR%/ %SF_APP% %SF_ENVIRONMENT%.Log

```

La mejor forma de crear una nueva factoría consiste en crear una nueva clase que herede de la clase por defecto y añadirle nuevos métodos. La factoría para las sesiones de usuario se establece a la clase `myUser` (localizada en `frontend/lib`) y hereda de la clase `sfUser`. Se puede

utilizar el mismo mecanismo para aprovechar las factorías ya existentes. El listado 17-8 muestra el ejemplo de una factoría para el objeto de la petición.

Listado 17-8 - Redefiniendo factorías

```
// Se crea la clase miRequest.class.php en un directorio para
// el que esté activada la carga automática de clases, por ejemplo
// frontend/lib/
<?php

class miRequest extends sfRequest
{
    // El código de la nueva factoría
}
# Se declara en el archivo factories.yml que esta nueva
# clase es la factoría para las peticiones
all:
    request:
        class: miRequest
```

17.3. Integrando componentes de otros frameworks

Si se requiere utilizar una clase externa y no se copia esa clase en algún directorio lib/ de Symfony, la clase se encontrará en algún directorio en el que Symfony no la puede encontrar. En este caso, si se utiliza esta clase en el código, es necesario incluir manualmente una instrucción require, a menos que se utilicen las propiedades de Symfony para enlazar y permitir la carga automática de otros componentes externos.

Symfony de momento no proporciona utilidades y herramientas para resolver cualquier tipo de problema. Si se necesita un generador de archivos PDF, una API para interactuar con los mapas de Google o una implementación en PHP del motor de búsqueda Lucene, es necesario hacer uso de algunas librerías del framework de Zend (<http://framework.zend.com/>). Si se quieren manipular imágenes directamente con PHP, conectarse con una cuenta POP3 para obtener los emails o diseñar una interfaz para la consola de comandos, seguramente se utilizarán los eZcomponents (<http://ez.no/ezcomponents>). Afortunadamente, si se utilizan las opciones correctas, se pueden utilizar directamente en Symfony todos los componentes de estas librerías externas.

Lo primero que hay que hacer es declarar la ruta al directorio raíz de cada librería en el archivo app.yml, a menos que las librerías se hayan instalado mediante PEAR.

```
all:
    zend_lib_dir:    /usr/local/zend/library/
    ez_lib_dir:      /usr/local/ezcomponents/
    swift_lib_dir:   /usr/local/swiftmailer/
```

A continuación, se extiende el mecanismo de carga automática de clases de PHP indicando que librería se debe utilizar cuando falle la carga automática de Symfony. Para ello, se registran las clases que se cargan de forma automática en la clase de configuración de la aplicación (el capítulo 19 lo explica en detalle) tal y como se muestra en el listado 17-9.

Listado 17-9 - Extendiendo el mecanismo de carga automática de clases para permitir el uso de componentes externos, en apps/frontend/config/ApplicationConfiguration.class.php

```
class frontendConfiguration extends sfApplicationConfiguration
{
    public function initialize()
    {
        parent::initialize(); // primero la carga automática de Symfony

        // Integrando el Zend Framework
        if ($sf_zend_lib_dir = sfConfig::get('app_zend_lib_dir'))
        {
            set_include_path($sf_zend_lib_dir.PATH_SEPARATOR.get_include_path());
            require_once($sf_zend_lib_dir.'/Zend/Loader.php');
            spl_autoload_register(array('Zend_Loader', 'Zend_Loader'));
        }

        // Integrando eZ Components
        if ($sf_ez_lib_dir = sfConfig::get('app_ez_lib_dir'))
        {
            set_include_path($sf_ez_lib_dir.PATH_SEPARATOR.get_include_path());
            require_once($sf_ez_lib_dir.'/Base/base.php');
            spl_autoload_register(array('ezcBase', 'autoload'));
        }

        // Integrando Swift Mailer
        if ($sf_swift_lib_dir = sfConfig::get('app_swift_lib_dir'))
        {
            set_include_path($sf_swift_lib_dir.PATH_SEPARATOR.get_include_path());
            require_once($sf_swift_lib_dir.'/Swift/ClassLoader.php');
            spl_autoload_register(array('Swift_ClassLoader', 'load'));
        }
    }
}
```

A continuación se describe lo que sucede cuando se crea un nuevo objeto de una clase que no ha sido cargada:

1. La función de Symfony encargada de la carga automática de clases busca la clase en las rutas especificadas en el archivo autoload.yml.
2. Si no se encuentra ninguna clase, se invocan uno a uno los métodos registrados con `spl_autoload_register()` hasta que uno de ellos devuelva el valor `true`. Por lo tanto, se invocan los métodos `Zend_Loader::Zend_Loader()`, `ezcBase::autoload()` y `Swift_ClassLoader::load()` hasta que alguno de ellos encuentre la clase.
3. Si todos los métodos anteriores devuelven `false`, PHP genera un error.

De esta forma, los componentes de otros frameworks pueden aprovecharse también del mecanismo de carga automática de clases, por lo que es incluso más sencillo que utilizarlos dentro de los frameworks originales. El siguiente código muestra por ejemplo cómo utilizar el componente `Zend_Search` (que implementa el motor de búsqueda Lucene en PHP) desde el propio framework Zend:

```
require_once 'Zend/Search/Lucene.php';
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
// ...
```

Utilizando Symfony y la carga automática de clases, es mucho más fácil utilizar este componente. De hecho, no es necesario utilizar ningún `require` y por tanto no tienes que preocuparte de las rutas de los archivos y de las clases:

```
$doc = new Zend_Search_Lucene_Document(); // The class is autoloaded
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
// ...
```

17.4. Plugins

En ocasiones, es necesario reutilizar una porción de código desarrollada para alguna aplicación Symfony. Si se puede encapsular ese código en una clase, tan sólo es necesario guardar la clase en algún directorio `lib/` para que otras aplicaciones puedan encontrarla. Sin embargo, si el código se encuentra desperdigado en varios archivos, como por ejemplo un tema para el generador de administraciones o una serie de archivos JavaScript y *helpers* que permiten utilizar fácilmente un efecto visual complejo, es muy complicado copiar todo este código en una clase.

Los plugins permiten agrupar todo el código diseminado por diferentes archivos y reutilizar este código en otros proyectos. Los plugins permiten encapsular clases, filtros, *event listeners*, *helpers*, archivos de configuración, tareas, módulos, esquemas y extensiones para el modelo, *fixtures*, archivos estáticos, etc. Los plugins son fáciles de instalar, de actualizar y de desinstalar. Se pueden distribuir en forma de archivo comprimido `.tgz`, un paquete PEAR o directamente desde el repositorio de código. La ventaja de los paquetes PEAR es que pueden controlar las dependencias, lo que simplifica su actualización. La forma en la que Symfony carga los plugins permite que los proyectos puedan utilizarlos como si fueran parte del propio framework.

Básicamente, un plugin es una extensión encapsulada para un proyecto Symfony. Los plugins permiten no solamente reutilizar código propio, sino que permiten aprovechar los desarrollos realizados por otros programadores y permiten añadir al núcleo de Symfony extensiones realizadas por otros desarrolladores.

17.4.1. Plugins disponibles para Symfony

El sitio web del proyecto Symfony dispone de una página dedicada a los plugins de Symfony. La página se encuentra dentro del wiki de Symfony, en la dirección: <http://trac.symfony-project.com/wiki/SymfonyPlugins>

Cada plugin que se muestra en ese listado, cuenta con su propia página con instrucciones para su instalación y toda la documentación necesaria.

Algunos plugins están desarrollados por voluntarios de la comunidad Symfony y otros han sido desarrollados por los mismos creadores de Symfony. Entre estos últimos se encuentran los siguientes:

- `sfFeed2Plugin`: automatiza la manipulación de los canales RSS y Atom.

- `sfThumbnailPlugin`: crea imágenes en miniatura, por ejemplo para las imágenes subidas por los usuarios.
- `sfMediaLibraryPlugin`: permite gestionar la subida de archivos multimedia, incluyendo una extensión para los editores avanzados de texto que permite incluir las imágenes dentro de los textos creados.
- `sfShoppingCartPlugin`: permite gestionar un carrito de la compra.
- `sfPagerNavigationPlugin`: dispone de controles para paginar elementos de forma clásica y mediante Ajax, basados en el objeto `sfPager`.
- `sfGuardPlugin`: permite incluir autenticación, autorización y otras opciones de gestión de usuarios más avanzadas que las que proporciona por defecto Symfony.
- `sfPrototypePlugin`: permite incluir los archivos de prototype y script.aculo.us como librerías de JavaScript independientes.
- `sfSuperCachePlugin`: crea versiones cacheadas de las páginas web en el directorio de la cache bajo el directorio web raíz del proyecto, de forma que el servidor web pueda servirlos lo más rápidamente posible.
- `sfOptimizerPlugin`: optimiza el código fuente de la aplicación para que se ejecute más rápidamente en el entorno de producción (el próximo capítulo muestra los detalles).
- `sfErrorLoggerPlugin`: guarda un registro de todos los errores de tipo 404 y 500 en una base de datos e incluye un módulo de administración para gestionar estos errores.
- `sfSslRequirementPlugin`: proporciona soporte para la encriptación SSL en las acciones.

El wiki también contiene otros plugins utilizados para extender los objetos Propel, que también se suelen llamar behaviors. Entre otros, están disponibles los siguientes:

- `sfPropelParanoidBehaviorPlugin`: deshabilita el borrado de los objetos y lo reemplaza por la actualización de una columna llamada `deleted_at`.
- `sfPropelOptimisticLockBehaviorPlugin`: implementa la estrategia *optimistic locking* para los objetos Propel.

Se recomienda visitar de forma habitual el wiki de Symfony, ya que se añaden plugins constantemente y normalmente proporcionan utilidades muy empleadas en el desarrollo de aplicaciones web.

Además del wiki de Symfony, también se pueden distribuir los plugins en forma de archivo para bajar, se puede crear un canal PEAR o se pueden almacenar en un repositorio público.

17.4.2. Instalando un plugin

El proceso de instalación de los plugins varía en función del método utilizado para distribuirlo. Siempre es recomendable leer el archivo README incluido en el plugin o las instrucciones de instalación disponibles en la página de descarga del plugin.

Los plugins se instalan en cada proyecto. Todos los métodos descritos en las siguientes secciones resultan en la copia de los archivos de cada plugin en el directorio `miproyecto/plugins/[NOMBRE PLUGIN]/`.

17.4.2.1. Plugins PEAR

Los plugins listados en el wiki de Symfony se distribuyen en forma de paquetes PEAR asociados con una página del wiki y disponibles en el canal PEAR oficial de los plugins de Symfony: `plugins.symfony-project.org`. Para instalar un plugin de este tipo, se utiliza la tarea `plugin:install` indicando el nombre del plugin, tal y como muestra el listado 17-10.

Listado 17-10 - Instalando un plugin del wiki de Symfony a través del canal PEAR oficial de los plugins de Symfony

```
> cd miproyecto
> php symfony plugin:install nombrePlugin
```

También es posible descargar los archivos del plugin e instalarlo desde un directorio del sistema. En este caso, se utiliza la ruta absoluta hasta el archivo del paquete descargado, como se muestra en el listado 17-11.

Listado 17-11 - Instalando un plugin mediante un paquete PEAR descargado

```
> cd miproyecto
> php symfony plugin:install /ruta/hasta/el/archivo/descargado/nombrePlugin.tgz
```

Algunos plugins están alojados en canales PEAR externos. En este caso, se pueden instalar mediante la tarea `plugin:install` después de registrar el canal PEAR e indicando el nombre del canal, como se muestra en el listado 17-12.

Listado 17-12 - Instalando un plugin desde un canal PEAR externo

```
> cd miproyecto
> php symfony plugin:add-channel canal.symfony.pear.ejemplo.com
> php symfony plugin:install --channel=canal.symfony.pear.ejemplo.com nombrePlugin
```

Estas tres formas de instalar plugins utilizan paquetes PEAR, por lo que se utiliza el término "*Plugins PEAR*" para referirse de forma indistinta a los plugins del canal PEAR oficial, los de canales PEAR externos y los que se descargan en forma de paquete PEAR.

La tarea `plugin:install` dispone de varias opciones, tal y como se muestra en el listado 17-13.

Listado 17-13 - Utilizando opciones al instalar un plugin

```
> php symfony plugin:install --stability=beta nombrePlugin
> php symfony plugin:install --release=1.0.3 nombrePlugin
> php symfony plugin:install --install-deps nombrePlugin
```

Sugerencia Como sucede con todas las tareas de Symfony, puedes obtener ayuda sobre las opciones y argumentos de `plugin:install` ejecutando el comando `php symfony help plugin:install`

17.4.2.2. Plugins de archivo

Algunos plugins se distribuyen en forma de un archivo o un conjunto de archivos. Para instalarlos, simplemente se descomprimen los archivos en el directorio `plugins/` del proyecto. Si el plugin contiene un subdirectorio llamado `web/`, se copia o se realiza un enlace simbólico a este directorio desde el directorio `web/` del proyecto, como se muestra en el listado 17-14. Por último, siempre se debe borrar la cache después de instalar el plugin.

Listado 17-14 - Instalando un plugin desde un archivo

```
> cd plugins
> tar -zxpf miPlugin.tgz
> cd ..
> ln -sf plugins/miPlugin/web web/miPlugin
> php symfony cc
```

17.4.2.3. Instalando plugins desde un repositorio de código

En ocasiones, los plugins disponen de su propio repositorio de código para el versionado de su código fuente. Estos plugins se pueden instalar simplemente descargando el código desde el repositorio hasta el directorio `plugins/`, pero este método puede ser problemático si el propio proyecto también utiliza el versionado de su código fuente.

Un método alternativo consiste en declarar el plugin como una dependencia externa, de forma que cada vez que se actualice el código fuente del proyecto, también se actualice el código fuente del plugin. Los repositorios de tipo Subversion, guardan las dependencias externas en la propiedad `svn:externals`. Como se muestra en el listado 17-15, se puede añadir un plugin simplemente editando esta propiedad y actualizando posteriormente el código fuente del proyecto.

Listado 17-15 - Instalando un plugin desde un repositorio de código

```
> cd miproyecto
> svn propedit svn:externals plugins
    nombrePlugin http://svn.ejemplo.com/nombrePlugin/trunk
> svn up
> php symfony cc
```

Nota Si el plugin contiene un directorio llamado `web/`, se debe crear un enlace simbólico de la misma forma que la explicada para los plugins de archivos.

17.4.2.4. Activando el módulo de un plugin

Algunos plugins contienen módulos enteros. La única diferencia entre los módulos de plugins y los módulos normales es que los de los plugins no se guardan en el directorio `miproyecto/apps/frontend/modules/` (para facilitar su actualización). Además, se deben activar en el archivo `settings.yml`, como se muestra en el listado 17-16.

Listado 17-16 - Activando un módulo de plugin, en `frontend/config/settings.yml`

```
all:
  .settings:
    enabled_modules: [default, sfMiPluginModule]
```

Este funcionamiento se ha establecido para evitar las situaciones en las que los módulos de un plugin se puedan habilitar de forma errónea para una aplicación que no los requiere, lo que podría provocar un agujero de seguridad. Si un plugin dispone de dos módulos llamados frontend y backend, se debería habilitar el módulo frontend solamente para la aplicación frontend y el módulo backend en la aplicación backend. Este es el motivo por el que los módulos de los plugins no se activan automáticamente.

Sugerencia El módulo `default` es el único módulo activado por defecto. Realmente no es un módulo de plugin, ya que es del propio framework (se guarda en el directorio `$sf_symfony_lib_dir/controller/default/`). Este módulo se encarga de mostrar las páginas de bienvenida, las páginas de error 404 y las de los errores de seguridad por no haber proporcionado las credenciales adecuadas. Si no se quieren utilizar las páginas por defecto de Symfony, se puede eliminar este módulo de la opción `enabled_modules`.

17.4.2.5. Listando los plugins instalados

Accediendo al directorio `plugins/` del proyecto, se pueden observar los plugins instalados, pero la tarea `plugin:list` proporciona más información: el número de versión y el nombre del canal para cada plugin instalado (ver el listado 17-17).

Listado 17-17 - Listando los plugins instalados

```
> cd miproyecto
> php symfony plugin:list

Installed plugins:
sfPrototypePlugin      1.0.0-stable # plugins.symfony-project.com (symfony)
sfSuperCachePlugin     1.0.0-stable # plugins.symfony-project.com (symfony)
sfThumbnail            1.1.0-stable # plugins.symfony-project.com (symfony)
```

17.4.2.6. Actualizando y desinstalando plugins

Los plugins PEAR se pueden desinstalar ejecutando la tarea `plugin:uninstall` desde el directorio raíz del proyecto, como muestra el listado 17-18. Si el plugin se instaló desde un canal PEAR diferente al canal oficial de Symfony, para desinstalar el plugin también se debe indicar el nombre del canal (se puede obtener el nombre del canal mediante la tarea `plugin:list`).

Listado 17-18 - Desinstalando un plugin

```
> cd miproyecto
> php symfony plugin:uninstall sfPrototypePlugin
> php symfony cc
```

Para desinstalar un plugin de archivo o un plugin instalado desde un repositorio, se borran manualmente los archivos del plugin que se encuentran en los directorios `plugins/` y `web/` y se borra la cache.

Para actualizar un plugin, se puede utilizar la tarea `plugin:upgrade` (para los plugins PEAR) o se puede ejecutar directamente `svn update` (si el plugin se ha instalado desde un repositorio de código). Los plugins de archivo no se pueden actualizar de una forma tan sencilla.

17.4.3. Estructura de un plugin

Los plugins se crean mediante el lenguaje PHP. Si se entiende la forma en la que se estructura una aplicación, es posible comprender la estructura de un plugin.

17.4.3.1. Estructura de archivos de un plugin

El directorio de un plugin se organiza de forma muy similar al directorio de un proyecto. Los archivos de un plugin se deben organizar de forma adecuada para que Symfony pueda cargarlos automáticamente cuando sea necesario. El listado 17-19 muestra la estructura de archivos de un plugin.

Listado 17-19 - Estructura de archivos de un plugin

```
nombrePlugin/
  config/
    *schema.yml          // Esquema de datos
    *schema.xml
    config.php           // Configuración específica del plugin
  data/
    generator/
      sfPropelAdmin
      */                 // Temas para el generador de administraciones
      template/
      skeleton/
    fixtures/
      *.yml              // Archivos de fixtures
  lib/
    *.php                // Clases
    helper/
      *.php              // Helpers
    model/
      *.php              // Clases del modelo
    task/
      *Task.class.php    // Tareas de la línea de comandos
  modules/
    */                  // Módulos
    actions/
      actions.class.php
    config/
      module.yml
      view.yml
      security.yml
    templates/
      *.php
    validate/
      *.yml
  web/
    *                    // Archivos estáticos
```

17.4.3.2. Posibilidades de los plugins

Los plugins pueden contener numerosos elementos. Su contenido se tiene en consideración durante la ejecución de la aplicación y cuando se ejecutan tareas mediante la línea de comandos.

Sin embargo, para que los plugins funcionen correctamente, es necesario seguir una serie de convenciones:

- Los esquemas de bases de datos los detectan las tareas propel-. Cuando se ejecuta la tarea propel-build-model para el proyecto, se reconstruye el modelo del proyecto y los modelos de todos los plugins que dispongan de un modelo. Los esquemas de los plugins siempre deben contener un atributo package que siga la notación plugins.nombrePlugin.lib.model, como se muestra en el listado 17-20.

Listado 17-20 - Ejemplo de declaración de un esquema de un plugin, en miPlugin/config/schema.yml

```
propel:
  _attributes:      { package: plugins.miPlugin.lib.model }
  mi_plugin_foobar:
    _attributes:    { phpName: miPluginFoobar }
    id:
      name:         { type: varchar, size: 255, index: unique }
    ...
```

- La configuración del plugin se incluye en el script de inicio del plugin (config.php). Este archivo se ejecuta después de las configuraciones de la aplicación y del proyecto, por lo que Symfony ya se ha iniciado cuando se procesa esta configuración. Se puede utilizar este archivo por ejemplo para extender las clases existentes con *event listeners* y comportamientos.
- Los archivos de datos o *fixtures* del directorio data/fixtures/ del plugin se procesan mediante la tarea propel:data-load.
- Las clases propias se cargan automáticamente de la misma forma que las clases que se guardan en las carpetas lib/ del proyecto.
- Cuando se realiza una llamada a use_helper() en las plantillas, se cargan automáticamente los *helpers* de los plugins. Estos *helpers* deben encontrarse en un subdirectorio llamado helper/ dentro de cualquier directorio lib/ del plugin.
- Las clases del modelo en el directorio miplugin/lib/model/ se utilizan para especializar las clases del modelo generadas por Propel (en miplugin/lib/model/om/ y miplugin/lib/model/map/). Todas estas clases también se cargan automáticamente. Las clases del modelo generado para un plugin no se pueden redefinir en los directorios del proyecto.
- Las tareas del plugin están disponibles en la línea de comandos de Symfony tan pronto como se instala el plugin. Los plugins pueden crear nuevas tareas o redefinir el comportamiento de las tareas existentes. Una buena práctica consiste en utilizar el nombre del plugin como *namespace* de sus tareas. Si se ejecuta el comando php symfony en la línea de comandos, se puede ver la lista completa de tareas disponibles, incluyendo las tareas proporcionadas por todos los plugins instalados.
- Los módulos proporcionan nuevas acciones, siempre que se declaren en la opción enabled_modules de la aplicación.

- Los archivos estáticos (imágenes, scripts, hojas de estilos, etc.) se sirven como el resto de archivos estáticos del proyecto. Cuando se instala un plugin mediante la línea de comandos, Symfony crea un enlace simbólico al directorio web/ del proyecto si el sistema operativo lo permite, o copia el contenido del directorio web/ del módulo en el directorio web/ del proyecto. Si el plugin se instala mediante un archivo o mediante un repositorio de código, se debe copiar manualmente el directorio web/ del plugin (como debería indicar el archivo README incluido en el plugin).

Sugerencia Registrar rutas con un plugin

Los plugins pueden añadir nuevas rutas al sistema de enrutamiento, pero no pueden utilizar un archivo de configuración similar a `routing.yml` para hacerlo. El motivo es que el orden en el que se definen las reglas es muy importante y la configuración en cascada de Symfony mezclaría todas las rutas de los archivos YAML. Por lo tanto, los plugins que añaden rutas deben registrar un *event listener* para el evento `routing.load_configuration` y deben añadir las rutas directamente en el *listener*:

```
// en plugins/miPlugin/config/config.php
$this->dispatcher->connect('routing.load_configuration', array('miPluginEnrutamiento',
'listenerEventoCargaConfiguracionEnrutamiento'));

// en plugins/miPlugin/lib/miPluginEnrutamiento.php
class miPluginEnrutamiento
{
    static public function listenerEventoCargaConfiguracionEnrutamiento(sfEvent $evento)
    {
        $enrutamiento = $evento->getSubject();
        // add plug-in routing rules on top of the existing ones
        $enrutamiento->prependRoute('mi_ruta', '/mi_plugin/:action', array('module' =>
'miPluginInterfazAdministracion'));
    }
}
```

17.4.3.3. Configuración manual de plugins

Algunas tareas no las puede realizar automáticamente el comando `plugin:install`, por lo que se deben realizar manualmente durante la instalación del plugin:

- El código de los plugins puede hacer uso de una configuración propia (por ejemplo mediante `sfConfig::get('app_miplugin_opcion')`), pero no se pueden indicar los valores por defecto en un archivo de configuración `app.yml` dentro del directorio `config/` del plugin. Para trabajar con valores por defecto, se utilizan los segundos argumentos opcionales en las llamadas a los métodos `sfConfig::get()`. Las opciones de configuración se pueden redefinir en el nivel de la aplicación (el listado 17-26 muestra un ejemplo).
- Las reglas de enrutamiento propias se deben añadir manualmente en el archivo `routing.yml`.
- Los filtros propios también se deben añadir manualmente al archivo `filters.yml` de la aplicación.

- Las factorías propias se deben añadir manualmente al archivo `factories.yml` de la aplicación.

En general, todas las configuraciones que deben realizarse sobre los archivos de configuración de las aplicaciones, se tienen que añadir manualmente. Los plugins que requieran esta instalación manual, deberían indicarlo en el archivo README incluido.

17.4.3.4. Personalizando un plugin para una aplicación

Cuando se necesita personalizar el funcionamiento de un plugin, nunca se debería modificar el código del directorio `plugins/`. Si se realizan cambios en ese directorio, se perderían todos los cambios al actualizar el plugin. Los plugins disponen de opciones y la posibilidad de redefinir su funcionamiento, de forma que se puedan personalizar sus características.

Los plugins que están bien diseñados disponen de opciones que se pueden modificar en el archivo `app.yml` de la aplicación, tal y como se muestra en el listado 17-21.

Listado 17-21 - Personalizando un plugin que utiliza la configuración de la aplicación

```
// Ejemplo de código del plugin
$foo = sfConfig::get('app_mi_plugin_opcion', 'valor');
# Modificar el valor por defecto de 'opcion' en el archivo
# app.yml de la aplicación
all:
  mi_plugin:
    opcion:      otrovalor
```

Las opciones del módulo y sus valores por defecto normalmente se describen en el archivo README del plugin.

Se pueden modificar los contenidos por defecto de un módulo del plugin creando un módulo con el mismo nombre en la aplicación. Realmente no se redefine el comportamiento del módulo original, sino que se sustituye, ya que se utilizan los elementos del módulo de la aplicación y no los del plugin. Funciona correctamente si se crean plantillas y archivos de configuración con el mismo nombre que los del plugin.

Por otra parte, si un plugin quiere ofrecer un módulo cuyo comportamiento se pueda redefinir, el archivo `actions.class.php` del módulo del plugin debe estar vacío y heredar de una clase que se cargue automáticamente, de forma que esta clase pueda ser heredada también por el `actions.class.php` del módulo de la aplicación. El listado 17-22 muestra un ejemplo.

Listado 17-22 - Personalizando la acción de un plugin

```
// En miPlugin/modules/mimodulo/lib/miPluginmimoduloActions.class.php
class miPluginmimoduloActions extends sfActions
{
  public function executeIndex()
  {
    // Instrucciones y código
  }
}

// En miPlugin/modules/mimodulo/actions/actions.class.php
```

```

require_once dirname(__FILE__).'/../lib/miPluginmimoduloActions.class.php';

class mimoduloActions extends miPluginmimoduloActions
{
    // Vacío
}

// En frontend/modules/mimodulo/actions/actions.class.php
class mimoduloActions extends miPluginmimoduloActions
{
    public function executeIndex()
    {
        // Aquí se redefine el código del plugin
    }
}

```

A partir de Symfony 1.1, cuando se construyen las clases del modelo, Symfony también busca los posibles archivos YAML utilizados para personalizar cada esquema de datos. La búsqueda se realiza incluso para los plugins, siguiendo la nomenclatura que se muestra a continuación:

Nombre original del esquema	Nombre del esquema personalizado
config/schema.yml	schema.custom.yml
config/loquesea_schema.yml	loquesea_schema.custom.yml
plugins/miPlugin/config/schema.yml	miPlugin_schema.custom.yml
plugins/miPlugin/config/loquesea_schema.yml	miPlugin_loquesea_schema.custom.yml

Los esquemas personalizados se buscan en los directorios config/ de la aplicación y de los plugins, por lo que un plugin puede redefinir el esquema de otros plugins y puede haber varias modificaciones para cada esquema de datos.

A continuación, Symfony fusiona los dos esquemas en base al valor phpName de cada tabla. El proceso de fusión de esquemas permite añadir o modificar tablas, columnas y atributos de columnas. El siguiente listado muestra por ejemplo cómo un esquema de datos personalizado añade columnas a una tabla definida en el esquema de datos de un plugin.

```

# Esquema original, en plugins/miPlugin/config/schema.yml
propel:
  articulo:
    _attributes: { phpName: Articulo }
    titulo:      varchar(50)
    usuario_id:  { type: integer }
    created_at:

# Esquema personalizado, en miPlugin_schema.custom.yml
propel:
  articulo:
    _attributes: { phpName: Articulo, package: valor1.valor2.lib.model }
    titulo_corto: varchar(50)

# Esquema resultante de la fusión realizada por Symfony
# Este esquema es el que se utiliza para generar el modelo

```

```

propel:
  articulo:
    _attributes:    { phpName: Articulo, package: valor1.valor2.lib.model }
    titulo:        varchar(50)
    usuario_id:    { type: integer }
    created_at:
    titulo_corto:  varchar(50)

```

Como la fusión de los esquemas se realiza mediante el valor del atributo `phpName` de la tabla, es posible incluso modificar el nombre de la tabla de la base de datos que utiliza el plugin, siempre que se mantenga el mismo valor en el atributo `phpName` del esquema.

17.4.4. Cómo crear un plugin

Solamente los plugins creados como paquetes PEAR se pueden instalar mediante la tarea `plugin:install`. Este tipo de plugins se pueden distribuir mediante el wiki de Symfony, mediante un canal PEAR o mediante la descarga de un archivo. Por tanto, si que quiere crear un plugin, es mejor publicarlo como paquete PEAR en vez de como archivo normal y corriente. Además, los plugins instalados mediante paquetes PEAR son más fáciles de actualizar, pueden declarar las dependencias que tienen y copian automáticamente los archivos estáticos en el directorio `web/`.

17.4.4.1. Organización de archivos

Si se ha creado una nueva característica para Symfony, puede ser útil encapsularla en un plugin para poder reutilizarla en otros proyectos. El primer paso es el de organizar los archivos de forma lógica para que los mecanismos de carga automática de Symfony puedan cargarlos cuando sea necesario. Para ello, se debe seguir la estructura de archivos mostrada en el listado 17-19. El listado 17-23 muestra un ejemplo de estructura de archivos para un plugin llamado `sfSamplePlugin`.

Listado 17-23 - Ejemplo de los archivos que se encapsulan en un plugin

```

sfSamplePlugin/
  README
  LICENSE
  config/
    schema.yml
  data/
    fixtures/
      fixtures.yml
  lib/
    model/
      sfSampleFooBar.php
      sfSampleFooBarPeer.php
    tasks/
      sfSampleTask.class.php
    validator/
      sfSampleValidator.class.php
  modules/
    sfSampleModule/
      actions/

```

```

    actions.class.php
  config/
    security.yml
  lib/
    BasesfSampleModuleActions.class.php
  templates/
    indexSuccess.php
web/
  css/
    sfSampleStyle.css
  images/
    sfSampleImage.png

```

Para la creación de los plugins, no es importante la localización del directorio del plugin (sfSamplePlugin/ en el caso del listado 17-23), ya que puede encontrarse en cualquier sitio del sistema de archivos.

Sugerencia Se aconseja ver la estructura de archivos de los plugins existentes antes de crear plugins propios, de forma que se puedan utilizar las mismas convenciones para el nombrado de archivos y la misma estructura de archivos.

17.4.4.2. Creando el archivo package.xml

El siguiente paso en la creación del plugin es añadir un archivo llamado package.xml en el directorio raíz del plugin. El archivo package.xml sigue la misma sintaxis de PEAR. El listado 17-24 muestra el aspecto típico de un archivo package.xml de un plugin.

Listado 17-24 - Ejemplo de archivo package.xml de un plugin de Symfony

```

<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.6" version="2.0"
  xmlns="http://pear.php.net/dtd/package-2.0"
  xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0 http://pear.php.net/dtd/
tasks-1.0.xsd http://pear.php.net/dtd/package-2.0 http://pear.php.net/dtd/
package-2.0.xsd">
  <name>sfSamplePlugin</name>
  <channel>plugins.symfony-project.org</channel>
  <summary>symfony sample plugin</summary>
  <description>Just a sample plugin to illustrate PEAR packaging</description>
  <lead>
    <name>Fabien POTENCIER</name>
    <user>fabpot</user>
    <email>fabien.potencier@symfony-project.com</email>
    <active>yes</active>
  </lead>
  <date>2006-01-18</date>
  <time>15:54:35</time>
  <version>
    <release>1.0.0</release>
    <api>1.0.0</api>
  </version>
  <stability>

```

```

    <release>stable</release>
    <api>stable</api>
</stability>
<license uri="http://www.symfony-project.org/license">MIT license</license>
<notes>-</notes>
<contents>
  <dir name="/">
    <file role="data" name="README" />
    <file role="data" name="LICENSE" />
    <dir name="config">
      <!-- model -->
      <file role="data" name="schema.yml" />
    </dir>
    <dir name="data">
      <dir name="fixtures">
        <!-- fixtures -->
        <file role="data" name="fixtures.yml" />
      </dir>
    </dir>
    <dir name="lib">
      <dir name="model">
        <!-- model classes -->
        <file role="data" name="sfSampleFooBar.php" />
        <file role="data" name="sfSampleFooBarPeer.php" />
      </dir>
      <dir name="tasks">
        <!-- tasks -->
        <file role="data" name="sfSampleTask.class.php" />
      </dir>
      <dir name="validator">
        <!-- validators -->
        <file role="data" name="sfSampleValidator.class.php" />
      </dir>
    </dir>
    <dir name="modules">
      <dir name="sfSampleModule">
        <file role="data" name="actions/actions.class.php" />
        <file role="data" name="config/security.yml" />
        <file role="data" name="lib/BasesfSampleModuleActions.class.php" />
        <file role="data" name="templates/indexSuccess.php" />
      </dir>
    </dir>
    <dir name="web">
      <dir name="css">
        <!-- stylesheets -->
        <file role="data" name="sfSampleStyle.css" />
      </dir>
      <dir name="images">
        <!-- images -->
        <file role="data" name="sfSampleImage.png" />
      </dir>
    </dir>
  </dir>
</contents>
<dependencies>

```



```

    <required>
      <php>
        <min>5.1.0</min>
      </php>
      <pearinstaller>
        <min>1.4.1</min>
      </pearinstaller>
      <package>
        <name>symfony</name>
        <channel>pear.symfony-project.com</channel>
        <min>1.1.0</min>
        <max>1.2.0</max>
        <exclude>1.2.0</exclude>
      </package>
    </required>
  </dependencies>
  <phprelease />
  <changelog />
</package>

```

Las partes más interesantes del archivo anterior son las etiquetas `<contents>` y `<dependencies>`, que se describen a continuación. Como el resto de etiquetas no son específicas de Symfony, se puede consultar la documentación de PEAR (<http://pear.php.net/manual/en/>) para obtener más información sobre el formato de `package.xml`.

17.4.4.3. Contenidos

La etiqueta `<contents>` se utiliza para describir la estructura de archivos de los plugins. Mediante esta etiqueta se dice a PEAR los archivos que debe copiar y el lugar en el que los debe copiar. La estructura de archivos se define mediante etiquetas `<dir>` y `<file>`. Todas las etiquetas de tipo `<file>` deben contener un atributo `role="data"`. La sección `<contents>` del listado 17-24 describe la estructura de directorios exacta del listado 17-23.

Nota El uso de etiquetas `<dir>` no es obligatorio, ya que se pueden utilizar rutas relativas como valor de los atributos `name` de las etiquetas `<file>`. No obstante, se recomienda utilizarlas para que el archivo `package.xml` sea fácil de leer.

17.4.4.4. Dependencias de los plugins

Los plugins están diseñados para funcionar con una serie de versiones de PHP, PEAR, Symfony, paquetes PEAR y otros plugins. La etiqueta `<dependencies>` declara todas estas dependencias y ayuda a PEAR a comprobar si se encuentran instalados todos los paquetes requeridos, lanzando una excepción si alguno no está disponible.

Siempre se deberían declarar las dependencias de PHP, PEAR y Symfony; al menos se deberían declarar las correspondientes a la instalación propia del autor del plugin, como requerimiento mínimo de instalación. Si no se sabe qué requerimientos establecer, se pueden indicar como requisitos PHP 5.1, PEAR 1.4 y Symfony 1.1.

También es recomendable añadir un número correspondiente a la versión más avanzada de Symfony para la que el plugin funciona correctamente. De esta forma, se producirá un error al

intentar utilizar un plugin con una versión muy avanzada de Symfony. Así, el autor del plugin se ve obligado a asegurar que el plugin funciona con las nuevas versiones de Symfony antes de lanzar una nueva versión del plugin. Siempre es mejor que se muestre un mensaje de error y se obligue a actualizar el plugin, que no simplemente dejar que el plugin no funcione y no avise de ninguna manera.

Si especificas los plugins como dependencias, los usuarios pueden instalar tu plugin y todas sus dependencias con un solo comando:

```
| > php symfony plugin:install --install-deps sfSamplePlugin
```

17.4.4.5. Construyendo el plugin

PEAR dispone de un comando (`pear package`) que construye un archivo comprimido de tipo `.tgz` con los contenidos del paquete, siempre que se ejecute el comando desde un directorio que contiene un archivo `package.xml`, tal y como muestra el listado 17-25:

Listado 17-25 - Creando un paquete PEAR para el plugin

```
| > cd sfSamplePlugin
| > pear package
|
| Package sfSamplePlugin-1.0.0.tgz done
```

Una vez construido el plugin, se puede comprobar que funciona correctamente instalándolo en el propio sistema, como se muestra en el listado 17-26.

Listado 17-26 - Instalando el plugin

```
| > cp sfSamplePlugin-1.0.0.tgz /home/production/miproyecto/
| > cd /home/production/miproyecto/
| > php symfony plugin:install sfSamplePlugin-1.0.0.tgz
```

Según la descripción de la etiqueta `<contents>`, los archivos del plugin se instalarán en diferentes directorios del proyecto. El listado 17-27 muestra donde acaban los archivos del plugin `sfSamplePlugin` después de su instalación.

Listado 17-27 - Los archivos del plugin se instalan en los directorios `plugins/` y `web/`

```
| plugins/
|   sfSamplePlugin/
|     README
|     LICENSE
|     config/
|       schema.yml
|     data/
|       fixtures/
|         fixtures.yml
|     lib/
|       model/
|         sfSampleFooBar.php
|         sfSampleFooBarPeer.php
|     tasks/
|       sfSampleTask.class.php
```

```
    validator/
        sfSampleValidator.class.php
modules/
    sfSampleModule/
        actions/
            actions.class.php
        config/
            security.yml
        lib/
            BasesfSampleModuleActions.class.php
        templates/
            indexSuccess.php
web/
    sfSamplePlugin/  ## Copia o enlace simbólico, dependiendo del sistema operativo
    css/
        sfSampleStyle.css
    images/
        sfSampleImage.png
```

Posteriormente, se comprueba si el plugin funciona correctamente dentro de la aplicación. Si todo funciona bien, el plugin ya está listo para ser utilizado en otros proyectos y para compartirlo con el resto de la comunidad de Symfony.

17.4.4.6. Distribuir un plugin desde el sitio web del proyecto Symfony

La mejor forma de publicitar un plugin es distribuirlo desde el sitio web `symfony-project.com`. Cualquier plugin desarrollado por cualquier programador se puede distribuir desde este sitio web, siempre que se realicen los siguientes pasos:

1. El archivo README del plugin debe describir la instalación y uso del plugin y el archivo LICENSE debe indicar el tipo de licencia de uso del plugin. El archivo README debe escribirse con el formato común de los wikis (<http://trac.symfony-project.com/wiki/WikiFormatting>).
2. Se crea un paquete PEAR para el plugin mediante el comando `pear package` y se prueba su funcionamiento. El nombre del paquete PEAR debe seguir la notación `sfSamplePlugin-1.0.0.tgz` (1.0.0 es la versión del plugin).
3. Se crea una nueva página en el wiki de Symfony llamada `sfSamplePlugin` (es obligatorio utilizar el sufijo `Plugin`). En esta página, se describe el uso del plugin, la licencia, las dependencias y el proceso de instalación. Se pueden reutilizar los contenidos del archivo README del plugin. Se pueden comprobar las páginas de los plugins existentes para utilizarlas como ejemplo.
4. Se adjunta el paquete PEAR a la página del wiki (`sfSamplePlugin-1.0.0.tgz`).
5. Se añade la página del plugin (`[wiki:sfSamplePlugin]`) a la lista de plugins disponibles, que también es una página del wiki y está disponible en (<http://trac.symfony-project.com/wiki/SymfonyPlugins>).

Si se siguen todos estos pasos, cualquier usuario puede instalar el plugin ejecutando el siguiente comando en el directorio de un proyecto Symfony:

```
| > php symfony plugin:install sfSamplePlugin
```

17.4.4.7. Convenciones sobre el nombre de los plugins

Para mantener el directorio `plugins/` limpio, todos los nombres de los plugins deberían seguir la notación *camelCase* y deben contener el sufijo `Plugin`, como por ejemplo `carritoCompraPlugin`, `feedPlugin`, etc. Antes de elegir el nombre de un plugin, se debe comprobar que no exista otro plugin con el mismo nombre.

Nota Los plugins relacionados con Propel deberían contener la palabra `Propel` en su nombre. Un plugin que por ejemplo se encargue de la autenticación mediante el uso de objetos Propel, podría llamarse `sfPropelAuth`.

Los plugins siempre deberían incluir un archivo `LICENSE` que describa las condiciones de uso del plugin y la licencia seleccionada por su autor. También se debería incluir en el archivo `README` información sobre los cambios producidos en cada versión, lo que realiza el plugin, las instrucciones sobre su instalación y configuración, etc.

17.5. Resumen

Las clases de Symfony contienen *hooks* utilizados por `sfMixer` para permitir ser modificadas a nivel de la aplicación. El mecanismo de *mixins* permite la herencia múltiple y la redefinición de métodos durante la ejecución de la aplicación, aunque las limitaciones de PHP no lo permitirían. De esta forma, es fácil extender las características de Symfony, incluso cuando se quieren reemplazar por completo las clases internas de Symfony, para lo que se dispone del mecanismo de factorías.

Muchas de las extensiones que se pueden realizar ya existen en forma de plugins, que se pueden instalar, actualizar y desinstalar fácilmente desde la línea de comandos de Symfony. Crear un plugin es tan sencillo como crear un paquete de PEAR y permite reutilizar un mismo código en varias aplicaciones diferentes.

El wiki de Symfony incluye muchos plugins y también es posible añadir plugins propios. Ahora que se sabe cómo hacerlo, los creadores de Symfony esperan que muchos programadores realicen mejoras a Symfony y las distribuyan a toda la comunidad de Symfony.

Capítulo 18. Rendimiento

Si una aplicación está pensada para ser utilizada por muchos usuarios, su optimización y su rendimiento son factores muy importantes a tener en cuenta durante su desarrollo. Como es lógico, el rendimiento siempre ha sido una de las máximas preocupaciones de los creadores de Symfony.

Aunque la gran ventaja de reducir el tiempo de desarrollo de una aplicación gracias a Symfony conlleva una disminución de su rendimiento, los programadores de Symfony siempre han tenido presente los requerimientos de rendimiento habituales. De esta forma, se ha analizado cada clase y cada método para que sean lo más rápidos posible.

La penalización mínima en el rendimiento de la aplicación (que se puede medir mostrando simplemente un mensaje tipo *"Hola Mundo"* con y sin Symfony) es muy reducida. Por tanto, el framework es escalable y responde correctamente a las pruebas de carga, también llamadas *pruebas de stress*. La prueba definitiva de su buen rendimiento es que algunos sitios con muchísimo tráfico (varios millones de usuarios y muchas interacciones Ajax) utilizan Symfony y están muy satisfechos con su rendimiento. La lista de sitios web desarrollados con Symfony se puede obtener en el wiki del proyecto: <http://trac.symfony-project.com/wiki/ApplicationsDevelopedWithSymfony>.

Evidentemente, los sitios web con millones de usuarios tienen los recursos necesarios para crear granjas de servidores y para mejorar el hardware de los servidores. No obstante, si no se dispone de este tipo de recursos, existen unos pequeños trucos que se pueden seguir para mejorar el rendimiento de las aplicaciones Symfony. En este capítulo se muestran algunas de las optimizaciones recomendadas para mejorar el rendimiento en todos los niveles del framework, aunque la mayoría están pensadas para usuarios avanzados. Aunque alguna técnica ya se ha comentado en otros capítulos anteriores, siempre es conveniente reunir todas las técnicas en un único lugar.

18.1. Optimizando el servidor

Una aplicación bien optimizada debería ejecutarse en un servidor que también estuviera muy optimizado. Para asegurar que no existe un cuello de botella en los elementos externos a Symfony, se deberían conocer las técnicas básicas para optimizar los servidores. A continuación se muestran una serie de opciones que se deben comprobar para que el rendimiento del servidor no se vea penalizado.

Si la opción `magic_quotes_gpc` del archivo `php.ini` tiene asignado un valor de `on`, el rendimiento de la aplicación disminuye, ya que PHP añade mecanismos de escape a todas las comillas de los parámetros de la petición y Symfony después aplica los mecanismos inversos, por lo que el único efecto de esta opción es una pérdida de rendimiento y posibles problemas en algunos sistemas. Si se tiene acceso a la configuración de PHP, se debería desactivar esta opción.

Cuanto más reciente sea la versión de PHP que se utiliza, mayor será el rendimiento. La versión PHP 5.2 es más rápida que PHP 5.1, que a su vez es mucho más rápida que PHP 5.0. De forma que

es una buena idea actualizar a la última versión de PHP para obtener una gran mejora en su rendimiento.

El uso de un acelerador de PHP (como por ejemplo, APC, XCache, o eAccelerator) es casi obligatorio en un servidor de producción, ya que mejora el rendimiento de PHP en un 50% y no tiene ningún inconveniente. Para disfrutar de la auténtica velocidad de ejecución de PHP, es necesario instalar algún acelerador.

Por otra parte, se deben desactivar en el servidor de producción todas las herramientas de depuración, como las extensiones Xdebug y APD.

Nota Si te estás preguntando sobre la penalización en el rendimiento causada por el uso de la extensión `mod_rewrite`, su efecto es despreciable. Aunque es evidente que cargar por ejemplo una imagen mediante la reglas de reescritura de este módulo es más lento que cargar la imagen directamente, la penalización producida es muchas órdenes de magnitud inferior a la ejecución de cualquier sentencia PHP.

Sugerencia Cuando un solo servidor no es suficiente, se puede añadir otro servidor y hacer un balanceo de la carga entre ellos. Mientras que el directorio `uploads/` sea compartido y se almacenen las sesiones en una base de datos, Symfony funciona igual de bien en una arquitectura de balanceo de carga.

18.2. Optimizando el modelo

En Symfony, la capa del modelo tiene fama de ser el componente más lento. Si las pruebas de rendimiento demuestran que se debe optimizar esta capa para una aplicación, a continuación se muestran las posibles mejoras que se pueden realizar.

18.2.1. Optimizando la integración de Propel

Inicializar la capa del modelo (las clases internas de Propel) requiere cierto tiempo, ya que se deben cargar algunas clases y se deben construir algunos objetos. No obstante, por la forma en la que Symfony integra Propel, esta inicialización solamente se produce cuando una acción requiere realmente utilizar el modelo, por lo que si sucede, se realiza lo más tarde posible. Las clases Propel se inicializan solamente cuando un objeto del modelo generado se carga automáticamente. Por tanto, las páginas que no utilizan el modelo no se ven penalizadas por la capa del modelo.

Si una aplicación no necesita la capa del modelo, se puede evitar la inicialización del objeto `sfDatabaseManager` desactivando por completo la capa del modelo mediante la siguiente opción del archivo `settings.yml`:

```
all:
  .settings:
    use_database: off
```

Las clases generadas para el modelo (en `lib/model/om/`) ya están optimizadas porque se les han eliminado los comentarios y también se cargan de forma automática cuando es necesario. Utilizar el sistema de carga automática en vez de incluir las clases a mano, garantiza que las clases se cargan solamente cuando son realmente necesarias. Por tanto, si una clase del modelo

no se utiliza, el mecanismo de carga automática ahorra tiempo de ejecución, mientras que la alternativa de utilizar sentencias `include` de PHP no podría ahorrarlo. En lo que respecta a los comentarios, se utilizan para documentar el uso de los métodos generados, pero aumentan mucho el tamaño de los archivos, lo que disminuye el rendimiento en los sistemas con discos duros lentos. Como los métodos de las clases generadas tienen nombres muy explícitos, los comentarios se desactivan por defecto.

Estas 2 mejoras son específicas de Symfony, pero se puede volver a las opciones por defecto de Propel cambiando estas 2 opciones en el archivo `propel.ini`, como se muestra a continuación:

```
propel.builder.addIncludes = true  # Añadir sentencias "include" en las clases
                                # en vez de utiliza la carga automática de clases
propel.builder.addComments = true  # Añadir comentarios a las clases generadas
```

18.2.2. Limitando el número de objetos que se procesan

Cuando se utiliza un método de una clase *peer* para obtener los objetos, el resultado de la consulta pasa el proceso de "hidratación" ("hydrating" en inglés) en el que se crean los objetos y se cargan con los datos de las filas devueltas en el resultado de la consulta. Para obtener por ejemplo todas las filas de la tabla `articulo` mediante Propel, se ejecuta la siguiente instrucción:

```
$articulos = ArticuloPeer::doSelect(new Criteria());
```

La variable `$articulos` resultante es un array con los objetos de tipo `Article`. Cada objeto se crea e inicializa, lo que requiere cierta cantidad de tiempo. La consecuencia de este comportamiento es que, al contrario de lo que sucede con las consultas a la base de datos, la velocidad de ejecución de una consulta Propel es directamente proporcional al número de resultados que devuelve. De esta forma, los métodos del modelo deberían optimizarse para devolver solamente un número limitado de resultados. Si no se necesitan todos los resultados devueltos por `Criteria`, se deberían limitar mediante los métodos `setLimit()` y `setOffset()`. Si solamente se necesitan por ejemplo las filas de datos de la 10 a la 20 para una consulta determinada, se puede refinar el objeto `Criteria` como se muestra en el listado 18-1.

Listado 18-1 - Limitando el número de resultados devueltos por `Criteria`

```
$c = new Criteria();
$c->setOffset(10); // Posición de la primera fila que se obtiene
$c->setLimit(10);  // Número de filas devueltas
$articulos = ArticuloPeer::doSelect($c);
```

El código anterior se puede automatizar utilizando un paginador. El objeto `sfPropelPager` gestiona de forma automática los valores `offset` y `limit` para una consulta Propel, de forma que solamente se crean los objetos mostrados en cada página. La documentación del paginador (http://www.symfony-project.org/cookbook/1_1/pager) dispone de más información sobre esta clase.

18.2.3. Minimizando el número de consultas mediante Joins

Mientras se desarrolla una aplicación, se debe controlar el número de consultas a la base de datos que realiza cada petición. La barra de depuración web muestra el número de consultas realizadas para cada página y al pulsar sobre el pequeño icono de una base de datos, se muestra el código SQL de las consultas realizadas. Si el número de consultas crece de forma desproporcionada, seguramente es necesario utilizar una Join.

Antes de explicar los métodos para Joins, se muestra lo que sucede cuando se recorre un array de objetos y se utiliza un método *getter* de Propel para obtener los detalles de la clase relacionada, como se ve en el listado 18-2. Este ejemplo supone que el esquema describe una tabla llamada *articulo* con una clave externa relacionada con la tabla *autor*.

Listado 18-2 - Obteniendo los detalles de una clase relacionada dentro de un bucle

```
// En la acción
$this->articulos = ArticuloPeer::doSelect(new Criteria());

// Consulta realizada en la base de datos por doSelect()
SELECT articulo.id, articulo.titulo, articulo.autor_id, ...
FROM   articulo

// En la plantilla
<ul>
<?php foreach ($articulos as $articulo): ?>
  <li><?php echo $articulo->getTitulo() ?>,
    escrito por <?php echo $articulo->getAutor()->getNombre() ?></li>
<?php endforeach; ?>
</ul>
```

Si el array `$articulos` contiene 10 objetos, el método `getAutor()` se llama 10 veces, lo que implica una consulta con la base de datos cada vez que se tiene que crear un objeto de tipo *Autor*, como se muestra en el listado 18-3.

Listado 18-3 - Los métodos *getter* de las claves externas, implican una consulta a la base de datos

```
// En la plantilla
$articulo->getAutor()

// Consulta a la base de datos producida por getAutor()
SELECT autor.id, autor.nombre, ...
FROM   autor
WHERE  autor.id = ?           // ? es articulo.autor_id
```

Por tanto, la página que genera el listado 18-2 implica 11 consultas a la base de datos: una consulta para construir el array de objetos *Articulo* y otras 10 consultas para obtener el objeto *Autor* asociado a cada objeto anterior. Evidentemente, se trata de un número de consultas muy grande para mostrar simplemente un listado de los artículos disponibles y sus autores.

Si se utiliza directamente SQL, es muy fácil reducir el número de consultas a solamente 1, obteniendo las columnas de la tabla *articulo* y las de la tabla *autor* mediante una única

consulta. Esto es exactamente lo que hace el método `doSelectJoinAutor()` de la clase `ArticuloPeer`. Este método realiza una consulta más compleja que un simple `doSelect()`, y las columnas adicionales que están presentes en el resultado obtenido permiten a Propel "hidratar" tanto los objetos de tipo `Articulo` como los objetos de tipo `Autor`. El código del listado 18-4 produce el mismo resultado que el del listado 18-2, pero solamente requiere 1 consulta con la base de datos en vez de 11 consultas, por lo que es mucho más rápido.

Listado 18-4 - Obteniendo los detalles de los artículos y sus autores en la misma consulta

```
// En la acción
$this->articulos = ArticuloPeer::doSelectJoinAutor(new Criteria());

// Consulta a la base de datos realizada por doSelectJoinAutor()
SELECT articulo.id, articulo.titulo, articulo.autor_id, ...
      autor.id, autor.name, ...
FROM   articulo, autor
WHERE  articulo.autor_id = autor.id

// En la plantilla no hay cambios
<ul>
<?php foreach ($articulos as $articulo): ?>
  <li><?php echo $articulo->getTitulo() ?>,
    escrito por <?php echo $articulo->getAutor()->getNombre() ?></li>
<?php endforeach; ?>
</ul>
```

No existen diferencias entre el resultado devuelto por `doSelect()` y el resultado devuelto por `doSelectJoinXXX()`; los dos métodos devuelven el mismo array de objetos (de tipo `Articulo` en este ejemplo). La diferencia se hace evidente cuando se utiliza un método *getter* asociado con una clave externa. En el caso del método `doSelect()`, se realiza una consulta a la base de datos y se crea un nuevo objeto con el resultado; en el caso del método `doSelectJoinXXX()`, el objeto asociado ya existe y no se realiza la consulta con la base de datos, por lo que el proceso es mucho más rápido. Por tanto, si se sabe de antemano que se van a utilizar los objetos relacionados, se debe utilizar el método `doSelectJoinXXX()` para reducir el número de consultas a la base de datos y por tanto, para mejorar el rendimiento de la página.

El método `doSelectJoinAutor()` se genera automáticamente cuando se ejecuta la tarea `propel-build-model`, debido a la relación entre las tablas `articulo` y `autor`. Si existen otras claves externas en la tabla del artículo, por ejemplo una tabla de categorías, la clase `BaseArticuloPeer` generada contendría otros métodos `Join`, como se muestra en el listado 18-5.

Listado 18-5 - Ejemplo de métodos `doSelect` disponibles para una clase `ArticuloPeer`

```
// Obtiene objetos "Articulo"
doSelect()

// Obtiene objetos "Articulo" y crea los objetos "Autor" relacionados
doSelectJoinAutor()

// Obtiene objetos "Articulo" y crea los objetos "Categoria" relacionados
doSelectJoinCategoria()
```

```
// Obtiene objetos "Articulo" y crea todos los objetos relacionados salvo "Autor"
doSelectJoinAllExceptAutor()

// Obtiene objetos "Articulo" y crea todos los objetos relacionados
doSelectJoinAll()
```

Las clases *peer* también disponen de métodos Join para `doCount()`. Las clases que soportan la internacionalización (ver Capítulo 13) disponen de un método `doSelectWithI18n()`, que se comporta igual que los métodos Join, pero con los objetos de tipo `i18n`. Para descubrir todos los métodos de tipo Join generados para las clases del modelo, es conveniente inspeccionar las clases *peer* generadas en el directorio `lib/model/om/`. Si no se encuentra el método Join necesario para una consulta (por ejemplo no se crean automáticamente los métodos Join para las relaciones muchos-a-muchos), se puede crear un método propio que extienda el modelo.

Sugerencia Evidentemente, la llamada al método `doSelectJoinXXX()` es un poco más lenta que la llamada a un método simple `doSelect()`, por lo que solamente mejora el rendimiento global de la página si se utilizan los objetos relacionados.

18.2.4. Evitar el uso de arrays temporales

Cuando se utiliza Propel, los objetos creados ya contienen todos los datos, por lo que no es necesario crear un array temporal de datos para la plantilla. Los programadores que no están acostumbrados a trabajar con ORM suelen caer en este error. Estos programadores suelen preparar un array de cadenas de texto o de números para las plantillas, mientras que, en realidad, las plantillas pueden trabajar directamente con los arrays de objetos. Si la plantilla por ejemplo muestra la lista de títulos de todos los artículos de la base de datos, un programador que no está acostumbrado a trabajar de esta forma puede crear un código similar al del listado 18-6.

Listado 18-6 - Crear un array temporal en la acción es inútil si ya se dispone de un array de objetos

```
// En la acción
$articulos = ArticuloPeer::doSelect(new Criteria());
$titulos = array();
foreach ($articulos as $articulo)
{
    $titulos[] = $articulo->getTitulo();
}
$this->titulos = $titulos;

// En la plantilla
<ul>
<?php foreach ($titulos as $titulo): ?>
    <li><?php echo $titulo ?></li>
<?php endforeach; ?>
</ul>
```

El problema del código anterior es que el proceso de creación de objetos del método `doSelect()` hace que crear el array `$titulos` sea inútil, ya que el mismo código se puede reescribir como muestra el listado 18-7. De esta forma, el tiempo que se pierde creando el array `$titulos` se puede aprovechar para mejorar el rendimiento de la aplicación.

Listado 18-7 - Utilizando el array de objetos, no es necesario crear un array temporal

```
// En la acción
$this->articulos = ArtículoPeer::doSelect(new Criteria());

// En la plantilla
<ul>
<?php foreach ($articulos as $articulo): ?>
    <li><?php echo $articulo->getTitulo() ?></li>
<?php endforeach; ?>
</ul>
```

Si realmente es necesario crear un array temporal porque se realiza cierto procesamiento con los objetos, la mejor solución es la de crear un nuevo método en la clase del modelo que devuelva directamente ese array. Si por ejemplo se necesita un array con los títulos de los artículos y el número de comentarios de cada artículo, la acción y la plantilla deberían ser similares a las del listado 18-8.

Listado 18-8 - Creando un método propio para preparar un array temporal

```
// En la acción
$this->articulos = ArtículoPeer::getArticuloTitulosConNumeroComentarios();

// En la plantilla
<ul>
<?php foreach ($articulos as $articulo): ?>
    <li><?php echo $articulo[0] ?> (<?php echo $articulo[1] ?> comentarios)</li>
<?php endforeach; ?>
</ul>
```

Solamente falta crear un método `getArticuloTitulosConNumeroComentarios()` muy rápido en el modelo, que se puede crear saltándose por completo el ORM y todas las capas de abstracción de bases de datos.

18.2.5. Saltándose el ORM

Cuando no se quieren utilizar los objetos completos, sino que solamente son necesarias algunas columnas de cada tabla (como en el ejemplo anterior) se pueden crear métodos específicos en el modelo que se salten por completo la capa del ORM. Se puede utilizar por ejemplo Creole para acceder directamente a la base de datos y devolver un array con un formato propio, como se muestra en el listado 18-9.

Listado 18-9 - Accediendo directamente con Creole para optimizar los métodos del modelo, en `lib/model/ArticuloPeer.php`

```
class ArtículoPeer extends BaseArticuloPeer
{
    public static function getArticuloTitulosConNumeroComentarios()
    {
        $conexion = Propel::getConnection();
        $consulta = 'SELECT %s as titulo, COUNT(%s) AS num_comentarios FROM %s LEFT JOIN %s
ON %s = %s GROUP BY %s';
        $consulta = sprintf($consulta,
            ArtículoPeer::TITULO, ComentarioPeer::ID,
```

```

        ArtículoPeer::TABLE_NAME, ComentarioPeer::TABLE_NAME,
        ArtículoPeer::ID, ComentarioPeer::ARTICULO_ID,
        ArtículoPeer::ID
    );
    $sentencia = $conexion->prepareStatement($consulta);
    $resultset = $sentencia->executeQuery();
    $resultados = array();
    while ($resultset->next())
    {
        $resultados[] = array($resultset->getString('titulo'),
        $resultset->getInt('num_comentarios'));
    }

    return $resultados;
}
}

```

Si se crean muchos métodos de este tipo, se puede acabar creando un método específico para cada acción, perdiendo la ventaja de la separación en capas y la abstracción de la base de datos.

Sugerencia Si Propel no es adecuado para la capa del modelo de algún proyecto, es mejor considerar el uso de otros ORM antes de escribir todas las consultas a mano. El plugin `sfDoctrine` proporciona una interfaz para el ORM `PhpDoctrine`. Además, se puede utilizar otra capa de abstracción de bases de datos en vez de `Creole`. Desde la versión PHP 5.1, se encuentra incluida en PHP la capa de abstracción `PDO`, que ofrece una alternativa mucho más rápida que `Creole`.

18.2.6. Optimizando la base de datos

Existen numerosas técnicas para optimizar la base de datos y que pueden ser aplicadas independientemente de `Symfony`. En esta sección, se repasan brevemente algunas de las estrategias más utilizadas, aunque es necesario un buen conocimiento de motores de bases de datos para optimizar la capa del modelo.

Sugerencia Recuerda que la barra de depuración web muestra el tiempo de ejecución de cada consulta realizada por la página, por lo que cada cambio que se realice debería comprobarse para ver si realmente reduce el tiempo de ejecución.

A menudo, las consultas a las bases de datos se realizan sobre columnas que no son claves primarias. Para aumentar la velocidad de ejecución de esas consultas, se deben crear índices en el esquema de la base de datos. Para añadir un índice a una columna, se añade la propiedad `index: true` a la definición de la columna, tal y como muestra el listado 18-10.

Listado 18-10 - Añadiendo un índice a una sola columna, en `config/schema.yml`

```

propel:
  articulo:
    id:
    autor_id:
    titulo: { type: varchar(100), index: true }

```

Se puede utilizar de forma alternativa el valor `index: unique` para definir un índice único en vez de un índice normal. El archivo `schema.yml` también permite definir índices sobre varias columnas (el Capítulo 8 contiene más información sobre la sintaxis de los índices). El uso de índices es muy recomendable, ya que es una buena forma de acelerar las consultas más complejas.

Después de añadir el índice al esquema, se debe añadir a la propia base de datos: directamente mediante una sentencia de tipo `ADD INDEX` o mediante el comando `propel-build-all` (que no solamente reconstruye la estructura de la tabla, sino que borra todos los datos existentes).

Sugerencia Las consultas de tipo `SELECT` son más rápidas cuando se utilizan índices, pero las sentencias de tipo `INSERT`, `UPDATE` y `DELETE` son más lentas. Además, los motores de bases de datos solamente utilizan 1 índice en cada consulta y determinan el índice a utilizar en cada consulta mediante métodos heurísticos internos. Por tanto, se deben medir las mejoras producidas por la creación de los índices, ya que en ocasiones las mejoras producidas en el rendimiento son muy escasas.

A menos que se especifique lo contrario, en Symfony cada petición utiliza una conexión con la base de datos y esa conexión se cierra al finalizar la petición. Se pueden habilitar conexiones persistentes con la base de datos, de forma que se cree un *pool* de conexiones abiertas con la base de datos y se reutilicen en las diferentes peticiones. La opción que se debe utilizar es `persistent: true` en el archivo `databases.yml`, como muestra el listado 18-11.

Listado 18-11 - Activar las conexiones persistentes con la base de datos, en `config/databases.yml`

```
prod:
  propel:
    class:      sfPropelDatabase
    param:
      persistent: true
      dsn:       mysql://login:passwd@localhost/blog
```

Esta opción puede mejorar el rendimiento de la base de datos o puede no hacerlo, dependiendo de numerosos factores. En Internet existe mucha documentación sobre las posibles mejoras que produce. Por tanto, es conveniente hacer pruebas de rendimiento sobre la aplicación antes y después de modificar el valor de esta opción.

Muchas de las opciones de configuración de MySQL, que se encuentran en el archivo `my.cnf`, pueden modificar el rendimiento de la base de datos. Por este motivo, es conveniente leer la documentación disponible al respecto en <http://dev.mysql.com/doc/refman/5.0/en/option-files.html>.

Una de las herramientas disponibles en MySQL es el archivo de log de las consultas lentas. Todas las consultas que tardan más de `long_query_time` segundos en ejecutarse (esta opción se modifica en el archivo `my.cnf`) se guardan en un archivo de log que es difícil de analizar manualmente, pero que el comando `mysqldumpslow` resume de forma muy útil. Se trata de una herramienta excelente para detectar las consultas que requieren ser optimizadas.

18.3. Optimizando la vista

En función del diseño y la implementación realizada en la capa de la vista, se pueden producir mejoras o pérdidas de rendimiento en la aplicación. En esta sección se describen diferentes alternativas y sus inconvenientes.

18.3.1. Utilizando el fragmento de código más rápido

Si no se utiliza el mecanismo de cache, se debe tener en cuenta que `include_component()` es un poco más lento que `include_partial()`, que a su vez, es un poco más lento que un simple `include` de PHP. El motivo es que Symfony instancia una vista para incluir un elemento parcial e instancia un objeto de tipo `sfComponent` para incluir un componente, que a su vez requiere un procesamiento ligeramente superior al necesario para incluir directamente un archivo.

De todas formas, la pérdida de rendimiento es insignificante, a menos que se incluyan muchos elementos parciales o muchos componentes en una plantilla. Por tanto, este caso se puede dar en listados y en tablas o cuando se utiliza la llamada al *helper* `include_partial()` dentro de una sentencia `foreach`. Si se incluyen muchos elementos parciales o componentes en una plantilla y ello reduce notablemente el rendimiento de la página, se debería utilizar el mecanismo de cache (ver Capítulo 12) y si no es posible hacerlo, utilizar sentencias `include` de PHP.

En lo que respecta a los *slots* y a los *slots de componentes*, su diferencia de rendimiento sí que es apreciable. El tiempo de procesamiento necesario para incluir un *slot* es despreciable, ya que es equivalente al tiempo requerido para instanciar una variable. Sin embargo, los *slots de componentes* se basan en una configuración de la vista y necesitan instanciar unos cuantos objetos para funcionar. No obstante, los *slots de componentes* se pueden guardar en la cache de forma independiente a la plantilla, mientras que los *slots* siempre se guardan en la cache junto con la plantilla que los incluye.

18.3.2. Optimizando el sistema de enrutamiento

Como se explica en el capítulo 9, todas las llamadas a los *helpers* de enlaces realizadas por las plantillas utilizan el sistema de enrutamiento para transformar una URI interna en una URL externa. El proceso consiste en encontrar un patrón en el archivo `routing.yml` que coincida con la URI indicada. Symfony realiza este proceso de forma muy sencilla: comprueba la primera regla del sistema de enrutamiento y si no coincide con la URI interna, continua probando las siguientes reglas. Como cada comprobación requiere el uso de expresiones regulares, puede ser un proceso que consume mucho tiempo de procesamiento.

Afortunadamente, existe una solución muy sencilla: utilizar el nombre de la regla en vez de los pares `modulo/accion`. Con este método, se indica a Symfony qué regla debe utilizar y por tanto el sistema de enrutamiento no pierde tiempo intentando encontrar la regla que coincida con la URI.

Si se considera por ejemplo la siguiente regla de enrutamiento definida en el archivo `routing.yml`:

```
articulo_segund_id:
  url:          /articulo/:id
  param:        { module: articulo, action: leer }
```

En este caso, en vez de utilizar el siguiente enlace:

```
<?php echo link_to('mi articulo', 'articulo/leer?id='.$articulo->getId()) ?>
```

Se debería utilizar esta otra versión mucho más rápida:

```
<?php echo link_to('mi articulo', '@articulo_segund_id?id='.$articulo->getId()) ?>
```

Cuando la página incluye docenas de enlaces creados con reglas de enrutamiento, las diferencias se hacen muy notables.

18.3.3. Saltándose la plantilla

Normalmente, la respuesta se compone de una serie de cabeceras y el contenido, aunque algunas respuestas no necesitan contenidos. Las interacciones Ajax por ejemplo, normalmente sólo requieren enviar unos pocos datos desde el servidor a un programa de JavaScript que se encarga de actualizar diferentes partes de la página. En este tipo de respuestas muy cortas, es mucho más rápido enviar sólo las cabeceras. Como se vio en el Capítulo 11, una acción puede devolver una sola cabecera JSON. El listado 18-12 muestra el ejemplo del Capítulo 11.

Listado 18-12 - Ejemplo de acción que devuelve una cabecera JSON

```
public function executeActualizar()
{
    $salida = '[[{"titulo", "Mi carta normal"}, {"nombre", "Sr. Pérez"}]]';
    $this->getResponse()->setHttpHeader("X-JSON", '('.$salida.')');

    return sfView::HEADER_ONLY;
}
```

El código anterior no utiliza ni plantillas ni layout y la respuesta se envía de una sola vez. Como sólo contiene cabeceras, la respuesta es mucho más corta y tarda mucho menos en llegar hasta el navegador del cliente.

El Capítulo 6 explica otra forma de evitar el uso de las plantillas y devolver el contenido en forma de texto directamente desde la acción. Aunque esta técnica rompe con la separación impuesta por el modelo MVC, aumenta significativamente la capacidad de respuesta de una acción. El listado 18-13 muestra un ejemplo.

Listado 18-13 - Ejemplo de acción que devuelve el contenido directamente en forma de texto

```
public function executeAccionRapida()
{
    return $this->renderText("<html><body>Hola Mundo</body></html>");
}
```

18.3.4. Reduciendo los helpers por defecto

En cada petición se cargan los grupos de *helpers* estándar (Partial, Cache y Form). Si se está seguro de que no se van a utilizar los *helpers* de algún grupo, se puede eliminar este grupo de la lista de *helpers* estándar, lo que evita que se tenga que procesar el archivo del *helper* en cada petición. En concreto, el grupo de *helpers* de formularios (Form) es bastante grande y por tanto, ralentiza la ejecución de las páginas que no utilizan formularios. Por tanto, es una buena idea modificar la opción `standard_helpers` del archivo `settings.yml` para no incluirlo por defecto:

```
all:
  .settings:
    standard_helpers: [Partial, Cache]    # Se elimina "Form"
```

El único inconveniente es que todas las plantillas que utilicen formularios tienen que declarar explícitamente que utilizan los *helpers* del grupo Form mediante la instrucción `use_helper('Form')`.

18.3.5. Comprimiendo la respuesta

Symfony comprime la respuesta antes de enviarla al navegador del cliente. Esta característica hace uso del módulo `zlib` de PHP. Si se quiere ahorrar el ligerísimo consumo de CPU que implica esta opción, se puede desactivar desde el archivo `settings.yml`:

```
all:
  .settings:
    compressed: off
```

Toda la mejora producida en la CPU se ve contrarrestada por una gran pérdida en el ancho de banda y en el tiempo de transmisión de la respuesta, por lo que esta opción no mejora el rendimiento en todas las aplicaciones.

Sugerencia Si se desactiva la compresión en PHP, se puede habilitar en el nivel del servidor. Apache dispone de su propia extensión para comprimir los contenidos.

18.4. Optimizando la cache

El Capítulo 12 describe cómo guardar en la cache partes de la respuesta o incluso la respuesta completa. Como guardar la respuesta en la cache mejora mucho el rendimiento de la aplicación, esta técnica debería ser una de las primeras a considerar para optimizar las aplicaciones. En esta sección se muestra cómo sacar el máximo partido a la cache e incluye algunos trucos muy interesantes.

18.4.1. Borrando partes de la cache de forma selectiva

Durante el desarrollo de una aplicación, se dan muchas situaciones en las que se debe borrar la cache:

- Cuando se crea una clase nueva: añadir la clase a un directorio para el que funciona la carga automática de clases (cualquier directorio `lib/` del proyecto) no es suficiente para que Symfony sea capaz de encontrarla en los entornos de ejecución que no sean el de

desarrollo. En este caso, es preciso borrar la cache de la carga automática para que Symfony recorrer otra vez todos los directorios indicados en el archivo `autoload.yml` y pueda encontrar las nuevas clases.

- Cuando se modifica la configuración en el entorno de producción: en producción, la configuración de la aplicación solamente se procesa durante la primera petición. Las siguientes peticiones utilizan la versión guardada en la cache. Por lo tanto, cualquier cambio en la configuración no tiene efecto en el entorno de producción (o en cualquier otro entorno donde la depuración de aplicaciones esté desactivada) hasta que se borre ese archivo de la cache.
- Cuando se modifica una plantilla en un entorno en el que la cache de plantillas está activada: en producción siempre se utilizan las plantillas guardadas en la cache, por lo que todos los cambios introducidos en las plantillas se ignoran hasta que la plantilla guardada en la cache se borra o caduca.
- Cuando se actualiza una aplicación mediante el comando `project:deploy`: este caso normalmente comprende las 3 modificaciones descritas anteriormente.

El problema de borrar la cache entera es que la siguiente petición tarda bastante tiempo en ser procesada, porque se debe regenerar la cache de configuración. Además, también se borran de la cache las plantillas que no han sido modificadas, por lo que se pierde la ventaja de haberlas guardado en la cache.

Por este motivo, es una buena idea borrar de la cache solamente los archivos que hagan falta. Las opciones de la tarea `cache:clear` pueden definir un subconjunto de archivos a borrar de la cache, como muestra el listado 18-14.

Listado 18-14 - Borrando solamente algunas partes de la cache

```
// Borrar sólo la cache de la aplicación "frontend"
> php symfony cache:clear frontend

// Borrar sólo la cache HTML de la aplicación "frontend"
> php symfony cache:clear frontend template

// Borrar sólo la cache de configuración de la aplicación "frontend"
> php symfony cache:clear frontend config
```

También es posible borrar a mano algunos archivos del directorio `cache/` o borrar las plantillas guardadas en la cache desde la acción mediante el método `$cacheManager->remove()`, como se describe en el capítulo 12.

Todas estas técnicas minimizan el impacto negativo sobre el rendimiento de todos los cambios mostrados anteriormente.

Sugerencia Cuando se actualiza Symfony, la cache se borra de forma automática, sin intervención manual (si se establece la opción `check_symfony_version` a `true` en el archivo de configuración `settings.yml`).

18.4.2. Generando páginas para la cache

Cuando se instala una nueva aplicación en producción, la cache de las plantillas está vacía. Para que una página se guarde en la cache, se debe esperar a que algún usuario visite esa página. En algunas aplicaciones críticas, no es admisible el tiempo de procesamiento de esa primera petición, por lo que se debe disponer de la versión de la página en la cache desde la primera petición.

La solución consiste en navegar de forma automática por las páginas de la aplicación en un entorno intermedio que se suele llamar *"staging"* y que dispone de una configuración similar a la del entorno de producción. De esta forma, se genera la cache completa de páginas y plantillas. Después, se puede transferir la aplicación a producción junto con la cache llena.

Para navegar de forma automática por todas las páginas de la aplicación, una opción consiste en utilizar un script de consola que navegue por una serie de URL mediante un navegador de texto (como por ejemplo "curl"). Otra opción mejor y más rápida consiste en utilizar un script de Symfony que utilice el objeto `sfBrowser` mostrado en el capítulo 15. Se trata de un navegador interno escrito en PHP y que utiliza el objeto `sfTestBrowser` para las pruebas funcionales. A partir de una URL externa, devuelve una respuesta, teniendo en cuenta la cache de las plantillas, como haría cualquier otro navegador. Como sólo se inicializa Symfony una vez y no pasa por la capa HTTP, este método es mucho más rápido.

El listado 18-15 muestra un script que genera la cache de plantillas en un entorno de tipo *"staging"*. Se puede ejecutar mediante `php batch/generar_cache.php`.

Listado 18-15 - Generando la cache de las plantillas, en `batch/generar_cache.php`

```
<?php

require_once(dirname(__FILE__).'../config/ProjectConfiguration.class.php');
$config = ProjectConfiguration::getApplicationConfiguration('frontend',
'staging', false);
sfContext::createInstance($config);

// Array de URL a navegar
$uris = array(
    '/foo/index',
    '/foo/bar/id/1',
    '/foo/bar/id/2',
    ...
);

$b = new sfBrowser();
foreach ($uris as $uri)
{
    $b->get($uri);
}
```

18.4.3. Guardando los datos de la cache en una base de datos

Por defecto, los datos de la cache de plantillas se guardan en el sistema de archivos: los trozos de HTML y los objetos serializados de la respuesta se guardan en el directorio `cache/` del proyecto. Symfony también incluye un método de almacenamiento alternativo para la cache: la base de datos SQLite. Este tipo de base de datos consiste en un archivo simple que PHP es capaz de reconocer como base de datos para buscar información en el archivo de forma muy eficiente.

Para indicar a Symfony que debería utilizar el almacenamiento de SQLite en vez del sistema de archivos, se debe modificar la opción `view_cache` del archivo de configuración `factories.yml`:

```
view_cache:
  class: sfSQLiteCache
  param:
    database: %SF_TEMPLATE_CACHE_DIR%/cache.db
```

La ventaja de utilizar el almacenamiento en SQLite es que la cache de las plantillas es mucho más fácil de leer y de escribir cuando el número de elementos de la cache es muy grande. Si la aplicación hace un uso intensivo de la cache, los archivos almacenados en la cache acaban en una estructura de directorios muy profunda, por lo que utilizar el almacenamiento de SQLite mejora el rendimiento de la aplicación.

Además, borrar una cache almacenada en el sistema de archivos requiere eliminar muchos archivos, por lo que es una operación que puede durar algunos segundos, durante los cuales la aplicación no está disponible. Si se utiliza el almacenamiento de SQLite, el proceso de borrado de la cache consiste en borrar un solo archivo, precisamente el archivo que se utiliza como base de datos SQLite. Independientemente del número de archivos en la cache, el borrado es instantáneo.

18.4.4. Saltándose Symfony

La mejor forma de mejorar el rendimiento de Symfony consiste en saltárselo por completo, aunque sea de forma parcial. Algunas páginas no cambian con cada petición, por lo que no es necesario procesarlas cada vez mediante el framework. Aunque la cache de las plantillas acelera el procesamiento de las páginas, todavía debe hacer uso de Symfony.

El capítulo 12 muestra algunos trucos con los que se puede evitar Symfony por completo para algunas páginas. El primer truco consiste en utilizar las cabeceras HTTP 1.1 para solicitar a los proxies y a los navegadores de los usuarios que guarden la página en sus propias caches y que no la soliciten la próxima vez que el usuario quiera acceder a la página. El segundo truco es la cache super rápida (que se puede automatizar mediante el plugin `sfSuperCachePlugin`) que consiste en guardar una copia de la respuesta en el directorio `web/` y la modificación de las reglas de reescritura de URL para que Apache busque en primer lugar la versión de la página en la cache antes de enviar la petición a Symfony.

Estos dos métodos son muy efectivos, aunque solamente se puedan aplicar a las páginas estáticas, ya que evita que estas páginas sean procesadas por Symfony, permitiendo a los servidores dedicarse al procesamiento de las peticiones complejas.

18.4.5. Guardando en la cache el resultado de una función

Si una función no depende del contexto de ejecución ni de variables aleatorias, al ejecutar 2 veces la misma función con los mismos parámetros, el resultado será el mismo. De esta forma, se podría evitar la segunda ejecución de la función si se ha almacenado el resultado de la primera ejecución. Esto es exactamente lo que permite hacer la clase `sfFunctionCache`. Esta clase dispone de un método llamado `call()`, al que se le pasa un elemento de PHP que se pueda ejecutar y un array de parámetros con los argumentos. Cuando se ejecuta, este método crea una huella digital mediante el método MD5 de todos los argumentos que se le han pasado y busca en la cache una clave que coincida con esta huella digital. Si se encuentra la clave, se devuelve el resultado almacenado en la cache. Si no se encuentra, `sfFunctionCache` ejecuta la función, almacena su respuesta en la cache y devuelve esta respuesta. Por tanto, la segunda ejecución del código del listado 18-16 es más rápida que la primera.

Listado 18-16 - Guardando el resultado de una función en la cache

```
$cache = new sfFileCache(array('cache_dir' =>
    sfConfig::get('sf_cache_dir').'/function'));
$fc = new sfFunctionCache($cache);
$resultado1 = $fc->call('cos', array(M_PI)
$resultado2 = $fc->call('preg_replace', array('/\s\s+/', ' ', $input));
```

El constructor de `sfFunctionCache` espera como argumento un objeto de tipo cache. El primer argumento del método `call()` debe ser cualquier elemento PHP que se pueda ejecutar, por lo que se puede indicar el nombre de una función, un array con el nombre de una clase y un método estático o un array con el nombre de un objeto y el de un método público. Respecto al otro argumento que se puede pasar al método `call()`, se trata de un array con todos los argumentos que se pasan al método o función.

Cuidado Si utilizas una cache basada en archivos como en el ejemplo anterior, es mejor indicar como directorio de la cache un directorio que se encuentre dentro de `cache/`, ya que de esta forma se borrará automáticamente cuando se ejecute la tarea `cache:clear`. Si guardas la cache de las funciones en otro sitio, no se borra automáticamente cuando borras la cache mediante la línea de comandos.

18.4.6. Guardando datos en la cache del servidor

Los aceleradores de PHP proporcionan unas funciones especiales para almacenar datos en la memoria, de forma que se puedan reutilizar entre diferentes peticiones. El problema es que cada acelerador utiliza su propia sintaxis y cada uno realiza esta tarea de una forma diferente. La cache de Symfony abstraer todas las diferencias en el funcionamiento de los diferentes aceleradores. Su sintaxis se muestra en el listado 18-17.

Listado 18-17 - Utilizando un acelerador de PHP para guardar datos en la cache

```
$cache = new sfAPCCache();

// Guardando datos en la cache
$cache->set($nombre, $valor, $tiempoDeVida);
```

```
// Accediendo a Los datos
$valor = $cache->get($nombre);

// Comprobando si un valor existe en La cache
$existe_valor = $cache->has($nombre);

// Borrar La cache
$cache->clear();
```

El método `set()` devuelve un valor `false` si no funciona la cache. El valor guardado en la cache puede ser de cualquier tipo (cadena, array, objeto); la clase `sfProcessCache` se encarga de la serialización automática. El método `get()` devuelve un valor `null` si la variable solicitada no existe en la cache.

Sugerencia Si se quiere profundizar en el uso de la cache en memoria, se debería utilizar la clase `sfMemcacheCache`. Esta clase dispone de la misma interfaz que el resto de las clases de cache y permite reducir la carga en la base de datos para las aplicaciones en las que se aplica el balanceo de carga.

18.5. Desactivando las características que no se utilizan

La configuración por defecto de Symfony activa las características más habituales para las aplicaciones web. No obstante, si no se necesitan todas estas características, es posible desactivarlas para ahorrar el tiempo requerido en inicializarlas durante cada petición.

Si por ejemplo una aplicación no utiliza el mecanismo de las sesiones o si se quiere realizar la gestión de sesiones manualmente, se debería establecer la opción `auto_start` a `false` bajo la clave `storage` del archivo de configuración `factories.yml`, como muestra el listado 18-19.

Listado 18-19 - Desactivando las sesiones, en `frontend/config/factories.yml`

```
all:
  storage:
    class: sfSessionStorage
    param:
      auto_start: false
```

Lo mismo se puede aplicar a la opción de base de datos (como se explicó en la sección anterior "Optimizando el modelo"). Si la aplicación no utiliza una base de datos, se puede desactivar esta opción para conseguir una ligera mejora en el rendimiento de la aplicación. Estas dos opciones se configuran en el archivo `settings.yml` (ver listado 18-20).

Listado 18-20 - Desactivando la opción de la base de datos, en `frontend/config/settings.yml`

```
all:
  .settings:
    use_database:      off      # Base de datos y modelo
```

Las opciones de seguridad (ver capítulo 6) se pueden desactivar en el archivo `filters.yml`, tal y como se muestra en el listado 18-21.

Listado 18-21 - Desactivando algunas características, en frontend/config/filters.yml

```
rendering: ~
security:
  enabled: off

# generally, you will want to insert your own filters here

cache:      ~
common:     ~
execution:  ~
```

Algunas opciones sólo son útiles durante el desarrollo de la aplicación, por lo que no se deberían activar en producción. Por defecto Symfony optimiza el rendimiento del entorno de producción deshabilitando todo lo innecesario. Entre las opciones que penalizan el rendimiento, el modo de depuración de aplicaciones es la más importante. Los archivos de log de Symfony también se desactivan por defecto en el entorno de producción.

Si los archivos de log se deshabilitan para las peticiones del entorno de producción, puede ser complicado solucionar los errores que se produzcan en este entorno. Afortunadamente, Symfony dispone de un plugin llamado `sfErrorLoggerPlugin`, que se ejecuta en segundo plano en el entorno de producción y guarda el log de los errores 404 y 500 en una base de datos. Se trata de un método mucho más rápido que los logs tradicionales, ya que los métodos del plugin sólo se ejecutan cuando falla una petición, mientras que el mecanismo de log penaliza el rendimiento en cualquier caso. Las instrucciones de instalación y el manual del plugin se pueden encontrar en <http://www.symfony-project.com/wiki/sfErrorLoggerPlugin>.

Sugerencia Se deben comprobar de forma regular los archivos de log de los errores del servidor, ya que contienen información muy útil sobre los errores 404 y 500.

18.6. Optimizando el código fuente

También es posible mejorar el rendimiento de la aplicación optimizando el código fuente de la propia aplicación. En esta sección se ofrecen algunos consejos al respecto.

18.6.1. Compilación del núcleo de Symfony

Cargar 10 archivos requieren muchas más operaciones de entrada/salida que cargar un solo archivo grande, sobre todo en discos lentos. Además, cargar un archivo muy grande consume muchos más recursos que cargar un archivo menor, sobre todo si una gran parte del archivo grande contiene información ignorada por PHP, por ejemplo los comentarios.

Por lo tanto, una operación que mejora mucho el rendimiento consiste en juntar una serie de archivos en un solo archivo y eliminar todos sus comentarios. Symfony ya realiza esta optimización y se llama compilación del núcleo de Symfony. Al principio de la primera petición (o después de que se haya borrado la cache) la aplicación Symfony concatena todas las clases del núcleo del framework Symfony (`sfActions`, `sfRequest`, `sfView`, etc.) en un solo archivo, optimiza el tamaño del archivo eliminando los comentarios y los espacios en blanco sobrantes y lo almacena en la cache, en un archivo llamado `config_core_compile.yml.php`. Las siguientes

peticiones solamente cargan este archivo optimizado en lugar de los 30 archivos individuales que lo componen.

Si la aplicación dispone de clases que deben cargarse siempre y sobre todo si son clases grandes con muchos comentarios, puede ser muy beneficioso añadirlas a la compilación del núcleo de Symfony. Para ello, se crea un archivo llamado `core_compile.yml` en el directorio `config/` de la aplicación y se listan las clases que se quieren añadir, como se muestra en el listado 18-22.

Listado 18-22 - Añadiendo las clases al archivo de compilación del núcleo de Symfony, en `frontend/config/core_compile.yml`

```
- %SF_ROOT_DIR%/lib/miClase.class.php
- %SF_ROOT_DIR%/apps/frontend/lib/miToolkit.class.php
- %SF_ROOT_DIR%/plugins/miPlugin/lib/miPluginCore.class.php
...
```

18.6.2. El plugin `sfOptimizer`

Symfony dispone de otra herramienta de optimización llamada `sfOptimizer`. Esta herramienta aplica varias estrategias de administración sobre el código de Symfony y el código de la aplicación, lo que permite acelerar la ejecución de la aplicación.

El código de Symfony realiza muchas comprobaciones sobre las opciones de configuración, y puede que la aplicación también lo haga. Si se observa el código de las clases de Symfony, se encuentran por ejemplo muchas comprobaciones del valor de la opción de configuración `sf_logging_enabled` antes de realizar una llamada al objeto `sfLogger`:

```
if (sfConfig::get('sf_logging_enabled'))
{
    $this->getContext()->getLogger()->info('Ha pasado por aquí');
}
```

Incluso aunque el registro creado con `sfConfig` está muy optimizado, el número de llamadas realizadas al método `get()` durante el procesamiento de cada petición es muy importante, lo que penaliza el rendimiento de la aplicación. Una de las estrategias de optimización de `sfOptimizer` consiste en reemplazar las constantes de configuración por su valor real, siempre que estas constantes no varíen durante la ejecución de la aplicación. Este es el caso de la opción `sf_logging_enabled`; si el valor de esta opción se establece a `false`, el plugin `sfOptimizer` transforma el código anterior en lo siguiente:

```
if (0)
{
    $this->getContext()->getLogger()->info('Ha pasado por aquí');
}
```

Y eso no es todo, ya que una comprobación tan evidente como la anterior, se transforma en una cadena de texto vacía.

Para aplicar las optimizaciones, se instala el plugin desde <http://trac.symfony-project.com/wiki/sfOptimizerPlugin> y después se ejecuta la tarea `optimize`, especificando el nombre de una aplicación y de un entorno:

```
| > php symfony optimize frontend prod
```

Si se quieren aplicar otras estrategias de optimización al código fuente, el plugin `sfOptimizer` puede ser un buen punto de partida.

18.7. Resumen

Symfony es un framework muy bien optimizado y que puede manejar los sitios web con mucho tráfico sin problemas. Sin embargo, si se quiere optimizar aún más el rendimiento de una aplicación, se puede modificar la configuración: la del servidor, la de PHP o la de la aplicación.

También se deberían seguir las buenas prácticas al escribir los métodos del modelo; y como la base de datos suele ser el cuello de botella de las aplicaciones web, se trata de uno de los puntos más importantes. Las plantillas también pueden utilizar algunos trucos interesantes, aunque la mejora más importante se consigue mediante la cache. Por último, existen algunos plugins que ofrecen técnicas bastante innovadoras para mejorar el rendimiento de las aplicaciones web (`sfSuperCache` y `sfOptimizer`).

Capítulo 19. Configuración avanzada

Ahora que se conoce Symfony muy bien, es posible adentrarse en lo más profundo de su código para comprender su arquitectura interna y para descubrir nuevas características. Sin embargo, antes de extender las clases de Symfony para adaptarlas a los requerimientos propios, se deberían analizar en detalle los archivos de configuración. Symfony incluye multitud de características que se pueden activar mediante una opción de configuración. Por tanto, se puede redefinir el comportamiento interno de Symfony sin necesidad de crear nuevas clases. En este capítulo se muestran en detalle todos los archivos de configuración y todo lo que se puede hacer con ellos.

19.1. Opciones de Symfony

El archivo `frontend/config/settings.yml` contiene la configuración principal de Symfony para la aplicación llamada `frontend`. Aunque ya se ha visto la utilidad de varias de sus opciones en los capítulos anteriores, a continuación se repasan todas estas opciones.

Como se explicó en el capítulo 5, este archivo es dependiente del entorno, lo que significa que cada opción puede tomar un valor diferente en cada entorno de ejecución. Todas las opciones definidas en este archivo son accesibles desde el código de PHP mediante la clase `sfConfig`. El nombre del parámetro que se debe utilizar está formado por el nombre de la opción y el prefijo `sf_`. Si se quiere obtener el valor de la opción `cache`, se utiliza el parámetro `sf_cache` y se obtiene su valor mediante `sfConfig::get('sf_cache')`.

19.1.1. Acciones y módulos por defecto

Symfony proporciona páginas por defecto para varias situaciones. Si se produce un error en el enrutamiento, Symfony ejecuta una acción del módulo `default` que se encuentra en el directorio `$sf_symfony_lib_dir/controller/default/`. El archivo `settings.yml` define la acción que se ejecuta en función del error producido:

- `error_404_module` y `error_404_action`: acción que se ejecuta cuando la URL solicitada por el usuario no cumple con ninguna de las rutas establecidas, o cuando se produce una excepción de tipo `sfError404Exception`. Su valor por defecto es `default/error404`.
- `login_module` y `login_action`: acción que se ejecuta cuando un usuario que no se ha autenticado intenta acceder a una página definida como segura (opción `secure`) en el archivo `security.yml` (el Capítulo 6 muestra los detalles). Su valor por defecto es `default/login`.
- `secure_module` y `secure_action`: acción que se ejecuta cuando un usuario no dispone de las credenciales requeridas para una ejecutar una acción. Su valor por defecto es `default/secure`.

- `module_disabled_module` y `module_disabled_action`: acción que se ejecuta cuando un usuario solicita un módulo que sido deshabilitado mediante el archivo `module.yml`. Su valor por defecto es `default/disabled`.

Antes de instalar una aplicación en producción, se deberían personalizar todas esas acciones, ya que las plantillas del módulo `default` incluyen el logotipo de Symfony en todas las páginas. La figura 19-1 muestra el aspecto de una de estas páginas, la página del error 404.

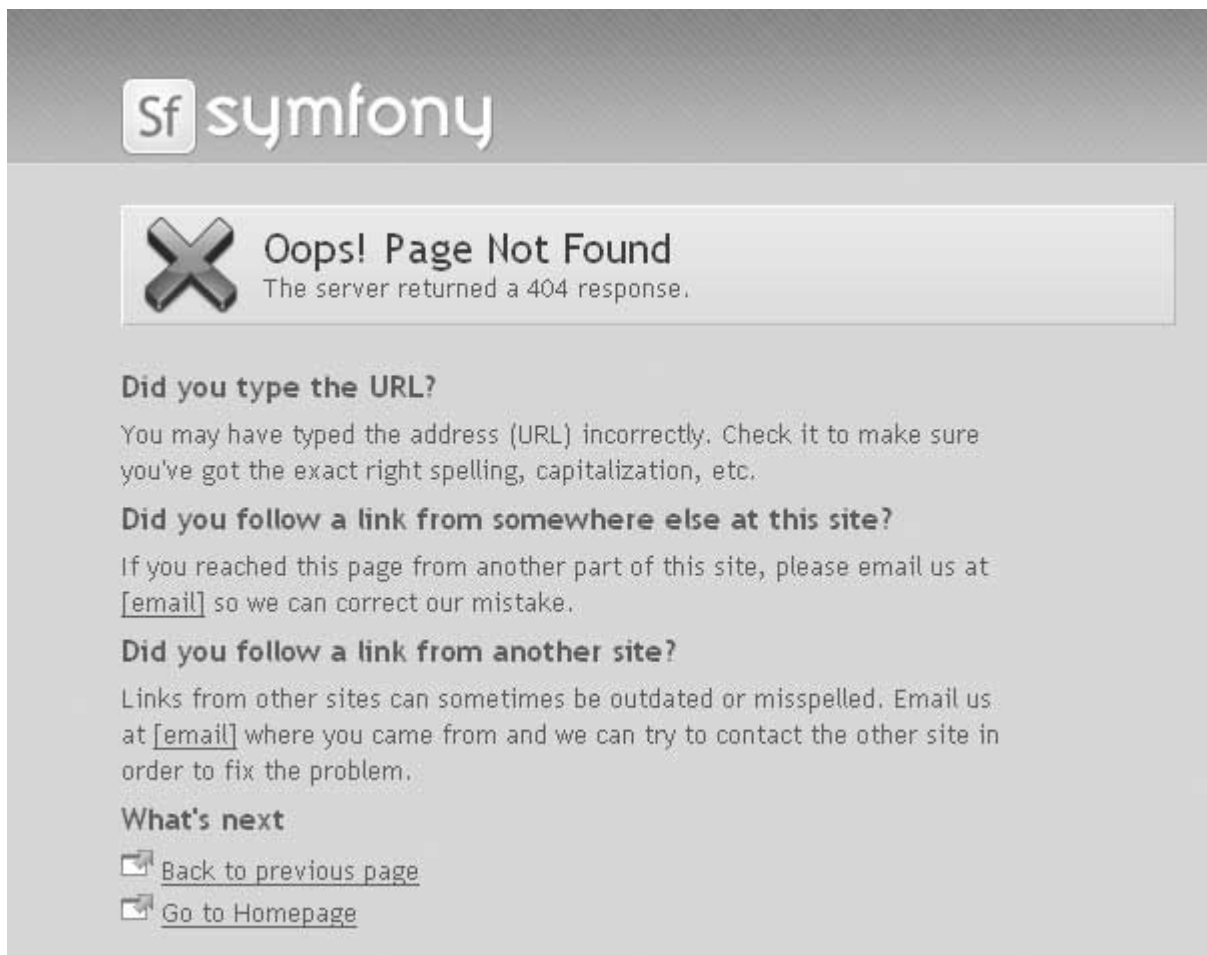


Figura 19.1. Página por defecto para el error 404

Se pueden modificar las páginas por defecto de 2 formas:

- Se puede crear un módulo llamado `default` dentro del directorio `modules/` de la aplicación y redefinir todas las acciones definidas en el archivo `settings.yml` (`index`, `error404`, `login`, `secure` y `disabled`) y todas las plantillas relacionadas (`indexSuccess.php`, `error404Success.php`, `loginSuccess.php`, `secureSuccess.php` y `disabledSuccess.php`).
- Se pueden modificar las opciones del módulo y acción por defecto del archivo `settings.yml` para utilizar páginas de la propia aplicación.

Existen otras dos páginas que muestran el mismo aspecto que el resto de páginas de Symfony y que también se deben modificar antes de instalar la aplicación en producción. Estas páginas no se encuentran en el módulo `default`, ya que se muestran cuando Symfony no se ejecuta

correctamente. Estas 2 páginas se encuentran en el directorio `$sf_symfony_lib_dir/exception/data/`:

- `error500.php`: página que se muestra cuando ocurre un error en el entorno de producción. En otros entornos en los que la depuración de aplicaciones está activada, Symfony muestra en estos casos un mensaje de error explícito y la traza completa de la ejecución (ver los detalles en el capítulo 16).
- `unavailable.php`: página que se muestra cuando un usuario solicita una página mientras la aplicación está deshabilitada (mediante la tarea `disable`). También se muestra esta página mientras se está borrando la cache (es decir, durante el tiempo que transcurre entre la ejecución de la tarea `php symfony cache:clear` y la finalización de esta tarea). Los sistemas que disponen de una cache muy grande, pueden tardar varios segundos en borrarla entera. Como Symfony no puede ejecutar una petición con una cache a medio borrar, las peticiones que se reciben antes del borrado completo se redirigen a esta página.

Para personalizar el aspecto de estas páginas, se crean los archivos `error500.php` y `unavailable.php` en el directorio `config/` del proyecto o de la aplicación. Si están disponibles en ese directorio, Symfony las utiliza en vez de sus propias páginas.

Nota Para redireccionar las peticiones a la página `unavailable.php` cuando se necesite, se debe establecer la opción `check_lock` a `on` en el archivo `settings.yml` de la aplicación. Esta opción está desactivada por defecto porque reduce muy ligeramente el rendimiento para cada petición.

19.1.2. Activando características opcionales

Algunas de las opciones del archivo `settings.yml` controlan las características opcionales del framework que se pueden activar y desactivar. Como desactivar las opciones que no se utilizan mejora el rendimiento de las aplicaciones, es conveniente repasar las opciones de la tabla 19-1 antes de instalar la aplicación en producción.

Tabla 19-1 - Características opcionales que se pueden activar mediante `settings.yml`

Opción	Descripción	Valor por defecto
<code>use_database</code>	Activa el gestor de bases de datos. Se debe establecer a <code>off</code> si no se utilizan bases de datos	<code>on</code>
<code>i18n</code>	Activa la traducción de la interfaz de la aplicación (ver capítulo 13). En las aplicaciones multidioma debería establecerse su valor a <code>on</code>	<code>off</code>
<code>logging_enabled</code>	Activa el sistema de log de eventos de Symfony. Si se establece su valor a <code>off</code> , no se tienen en cuenta las opciones del archivo <code>logging.yml</code> y se desactiva por completo el uso de archivos de log en Symfony	<code>on</code>
<code>escaping_strategy</code>	Activa y establece la política utilizada por el mecanismo de escape (ver capítulo 7). Utiliza el valor <code>on</code> para que se aplique el mecanismo de escape a los datos que se pasan a las plantillas	<code>off</code>

cache	Activa el mecanismo de cache para las plantillas (ver capítulo 12). Si algún módulo define un archivo <code>cache.yml</code> , su valor debe ser <code>on</code> . El filtro de la cache (<code>sfCacheFilter</code>) solamente se activa si esta opción vale <code>on</code>	<code>off</code> en desarrollo, <code>on</code> en producción
web_debug	Activa la barra de depuración web para depurar fácilmente las aplicaciones (ver capítulo 16). Para mostrar la barra en todas las páginas, se establece su valor a <code>on</code> .	<code>on</code> en desarrollo, <code>off</code> en producción
check_symfony_version	Activa la comprobación de la versión de Symfony para cada petición. Si se quiere borrar la cache automáticamente después de actualizar el framework, su valor debe ser <code>on</code> . Si se borra manualmente la cache después de cada actualización, su valor debe ser <code>off</code>	<code>off</code>
check_lock	Activa el sistema de bloqueo de la aplicación, que se inicia mediante las tareas <code>cache:clear</code> y <code>project:disable</code> (ver sección anterior). Si se establece su valor a <code>on</code> , todas las peticiones a una aplicación deshabilitada se redirigen a la página <code>\$sf_symfony_lib_dir/exception/data/unavailable.php</code>	<code>off</code>
compressed	Activa la compresión de la respuesta mediante PHP. Si se establece a <code>on</code> , se comprime el código HTML generado antes de enviar la respuesta mediante las opciones de compresión de PHP	<code>off</code>

19.1.3. Configuración de cada característica

Symfony utiliza algunas opciones del archivo `settings.yml` para modificar el comportamiento de algunas de sus características, como la validación de formularios, la cache, los módulos externos, etc.

19.1.3.1. Opciones del mecanismo de escape

Las opciones del mecanismo de escape controlan la forma en la que las plantillas acceden a las variables (ver capítulo 7). El archivo `settings.yml` incluye dos opciones para esta característica:

- La opción `escaping_strategy` puede tomar los valores `on` o `off`.
- La opción `escaping_method` puede valer `ESC_RAW`, `ESC_SPECIALCHARS`, `ESC_ENTITIES`, `ESC_JS` o `ESC_JS_NO_ENTITIES`.

19.1.3.2. Opciones del sistema de enrutamiento

Las opciones del sistema de enrutamiento (ver capítulo 9) se definen en el archivo de configuración `factories.yml`, bajo la clave `routing`. El listado 19-1 muestra la configuración por defecto del sistema de enrutamiento.

Listado 19-1 - Opciones de configuración del sistema de enrutamiento, en `frontend/config/factories.yml`

```
routing:
  class: sfPatternRouting
```

```

param:
  load_configuration: true
  suffix:             .
  default_module:     default
  default_action:     index
  variable_prefixes:  [ ':' ]
  segment_separators: [ '/', '.' ]
  variable_regex:     '[\w\d_]+'
  debug:              %SF_DEBUG%
  logging:             %SF_LOGGING_ENABLED%
  cache:
    class: sfFileCache
    param:
      automatic_cleaning_factor: 0
      cache_dir:                 %SF_CONFIG_CACHE_DIR%/routing
      lifetime:                  31556926
      prefix:                    %SF_APP_DIR%

```

- La opción `suffix` establece el sufijo por defecto para las URL generadas. Su valor por defecto es un punto (`.`), lo que significa que no se añade ningún sufijo. Si se establece su valor a `.html`, todas las URL generadas parecerán páginas estáticas.
- Cuando una regla de enrutamiento no define los parámetros `module` o `action`, se utilizan los valores por defecto de `factories.yml`:
 - `default_module`: valor por defecto del parámetro `module`. Su valor por defecto es `default`.
 - `default_action`: valor por defecto del parámetro `action`. Su valor por defecto es `index`.
- Los patrones de las rutas identifican los comodines con nombre mediante un prefijo formado por dos puntos (`:`). Se puede modificar ese valor por defecto para utilizar un formato más parecido a PHP. Para ello, se añade el símbolo del dólar (`$`) en el array de la opción `variable_prefixes`. De esta forma, se pueden utilizar patrones como `/articulo/$ano/$mes/$dia/$titulo` en vez de `/articulo/:ano/:mes/:dia/:titulo`
- Los patrones de cada regla separan los diferentes comodines con nombre mediante los separadores. Por defecto, los separadores permitidos son la barra (`/`) y el punto (`.`). Se pueden añadir todos los separadores que se necesiten en la opción `segment_separators`. Si por ejemplo se añade el guión medio (`-`), se pueden crear patrones como `/articulo/:ano-:mes-:dia/:titulo`
- En el entorno de producción, el sistema de enrutamiento utiliza una cache para mejorar el rendimiento en la transformación de URI internas en URL externas. Por defecto esta cache utiliza el sistema de archivos, pero se puede utilizar cualquier clase de cache, siempre que se declare esa clase y sus opciones en la opción `cache`. El capítulo 15 describe la lista completa de clases de cache. Para desactivar la cache del sistema de enrutamiento en el entorno de producción, se establece el valor `on` en la opción `debug`.

Las opciones anteriores son todas las opciones disponibles para la clase `sfPatternRouting`. Además, es posible utilizar otra clase para el sistema de enrutamiento de la aplicación, ya sea

una clase propia o una de las factorías incluidas por Symfony (`sfNoRouting` y `sfPathInfoRouting`). Estas dos factorías hacen que las URL tengan el aspecto `modulo/accion?clave1=valor1`. La desventaja es que no se pueden personalizar, pero su gran ventaja es que son muy rápidas. La diferencia entre las dos es que la primera utiliza el GET de PHP y la segunda utiliza el `PATH_INFO`. Se pueden utilizar sobre todo en las interfaces de administración de las aplicaciones.

Existe una última opción relacionada con el sistema de enrutamiento, pero se define en el archivo `settings.yml`:

- La opción `no_script_name` activa o desactiva la aparición del nombre del controlador frontal en las URL generadas. La opción `no_script_name` solamente se puede activar para una sola aplicación dentro de un mismo proyecto, a no ser que se guarden los controladores frontales en varios directorios diferentes y se modifiquen las reglas de enrutamiento por defecto para las URL. En el entorno de producción, esta opción suele establecerse a `on` y suele vale `off` en el resto de entornos.

19.1.3.3. Opciones para la validación de formularios

Nota Las opciones que se describen en esta sección han sido declaradas obsoletas en Symfony 1.1 por lo que sólo funcionan si se habilita el plugin `sfCompat10`.

Las opciones de validación de formularios controlan la forma en la que se muestran los mensajes de error de los *helpers* del grupo `Validation` (ver capítulo 10). Este tipo de errores se muestran dentro de etiquetas `<div>`, y utilizan el valor de la opción `validation_error_class` como atributo `class` del `<div>` y el valor de la opción `validation_error_id_prefix` para construir el atributo `id`. Los valores por defecto son `form_error` y `error_for_`, por lo que los atributos generados por la llamada al *helper* `form_error()` para un campo de formulario llamado `campo` serían `class="form_error" id="error_for_campo"`.

Otras dos opciones determinan los caracteres que se muestran delante y detrás de los mensajes de error: `validation_error_prefix` y `validation_error_suffix`. Cambiando su valor, se modifica el aspecto de todos los mensajes de error generados.

19.1.3.4. Opciones para la cache

Las mayoría de opciones de la cache se definen en el archivo `cache.yml`, salvo dos opciones incluidas en el archivo `settings.yml`: `cache` que activa el mecanismo de cache de las plantillas y `etag` que controla la etiqueta `Etag` en el lado del servidor (ver capítulo 15). También es posible configurar en el archivo `factories.yml` el tipo de almacenamiento que se utiliza en todas las caches (la cache de la vista, la del sistema de enrutamiento y la de la internacionalización). El listado 19-2 muestra la configuración por defecto de la factoría de la cache de la vista.

Listado 19-2 - Opciones de configuración de la cache de la vista, en `frontend/config/factories.yml`

```
view_cache:
  class: sfFileCache
  param:
```

```

    automatic_cleaning_factor: 0
    cache_dir:                 %SF_TEMPLATE_CACHE_DIR%
    lifetime:                  86400
    prefix:                    %SF_APP_DIR%/template

```

La opción `class` puede tomar los valores `sfFileCache`, `sfAPCCache`, `sfEAcceleratorCache`, `sfXCacheCache`, `sfMemcacheCache` y `sfSQLiteCache`. También puedes utilizar tu propia clase, siempre que herede de la clase `sfCache` y proporcione los mismos métodos genéricos para guardar, obtener y borrar elementos en la cache mediante claves. Las opciones de esta factoría dependen de cada clase, aunque algunas opciones son comunes para todas:

- `lifetime` establece el tiempo de expiración en segundos de los elementos de la cache.
- `prefix` establece el prefijo que se añade a cada clave de la cache. Para compartir una misma cache entre dos aplicaciones, es necesario utilizar el mismo prefijo.

Además, para cada factoría es necesario definir el lugar en el que se va a almacenar la cache:

- `sfFileCache` utiliza el parámetro `cache_dir` para establecer la ruta absoluta hasta el directorio de la cache.
- `sfAPCCache`, `sfEAcceleratorCache` y `sfXCacheCache` no necesitan una opción para indicar el lugar en el que se almacena la cache, ya que utilizan las funciones nativas de PHP para comunicarse con los sistemas de cache de APC, EAccelerator y XCache respectivamente.
- `sfMemcacheCache` utiliza el parámetro `host` para establecer el hostname del servidor de Memcached. También se puede utilizar el parámetro `servers` para indicar un array de servidores.
- `sfSQLiteCache` utiliza el parámetro `database` para indicar la ruta absoluta hasta el archivo de la base de datos de tipo SQLite.

La documentación de la API de cada clase de cache contiene más información sobre todas sus opciones.

La vista no es el único componente que puede utilizar una cache. Las factorías `routing` y `I18N` incluyen un parámetro llamado `cache` en el que se puede indicar cualquier factoría de cache, de la misma forma que en la cache de la vista. El listado 19-1 por ejemplo muestra cómo el sistema de enrutamiento utiliza por defecto la cache para mejorar su rendimiento, pero se puede modificar esa opción por cualquier otro valor de los mostrados anteriormente.

19.1.3.5. Opciones para los archivos de log

El archivo `settings.yml` incluye 2 opciones relacionadas con los archivos de log (ver capítulo 16):

- `error_reporting` especifica los eventos que se guardan en los archivos de log de PHP. Su valor por defecto es `E_PARSE | E_COMPILE_ERROR | E_ERROR | E_CORE_ERROR | E_USER_ERROR` para el entorno de producción (por lo que los eventos que se guardan en el log son `E_PARSE`, `E_COMPILE_ERROR`, `E_ERROR`, `E_CORE_ERROR` y `E_USER_ERROR`) y `E_ALL | E_STRICT` para el entorno de desarrollo.

- La opción `web_debug` activa la barra de depuración web. su valor debería ser `on` solamente en los entornos de desarrollo y pruebas.

19.1.3.6. Opciones para las rutas a los archivos estáticos

El archivo `settings.yml` también permite indicar la ruta a los archivos estáticos, también llamados "*assets*". Si se quiere utilizar una versión específica de un componente que es diferente a la que se incluye en Symfony, estas opciones permiten establecer la ruta a la nueva versión:

- Los archivos JavaScript del editor de textos avanzado se configuran mediante la opción `rich_text_js_dir` (por defecto, `js/tiny_mce`)
- Las librerías de Prototype se configuran mediante `prototype_web_dir` (por defecto, `/sf/prototype`)
- Los archivos del generador de administraciones se configuran mediante `admin_web_dir`
- Los archivos de la barra de depuración web se configuran mediante `web_debug_web_dir`
- Los archivos JavaScript del calendario avanzado se configuran mediante `calendar_web_dir`

19.1.3.7. Helpers por defecto

Los *helpers* por defecto se cargan en todas las plantillas y se configuran mediante la opción `standard_helpers` (ver capítulo 7). Por defecto, se incluyen los grupos de *helpers* `Partial`, `Cache` y `Form`. Si algún grupo de *helpers* se utiliza en todas las plantillas, es mejor añadirlo a la opción `standard_helpers` para evitar tener que declarar su uso en todas las plantillas mediante `use_helper()`.

19.1.3.8. Módulos activos

La opción `enabled_modules` establece los módulos de los plugins o del propio núcleo de Symfony que están activos. Aunque un plugin incluya un módulo, los usuarios no pueden acceder a este módulo a menos que haya sido incluido en la lista de la opción `enabled_modules`. Por defecto solamente se encuentra activado el módulo `default` de Symfony, que muestra las páginas por defecto de bienvenida, de errores como "página no encontrada", etc.

19.1.3.9. Juego de caracteres

El juego de caracteres utilizado en las respuestas es una opción global de la aplicación, ya que se utiliza en muchos componentes del framework (plantillas, mecanismo de escape, *helpers*, etc.). Su valor se define en la opción `charset`, cuyo valor por defecto y recomendado es `utf-8`.

19.1.3.10. Otras configuraciones

El archivo `settings.yml` contiene otras opciones que Symfony utiliza internamente para definir su comportamiento. El listado 19-3 muestra todas las opciones en el mismo orden en el que aparecen en el archivo de configuración.

Listado 19-3 - Otras opciones de configuración, en frontend/config/settings.yml

```
# Eliminar los comentarios de las clases internas de Symfony, tal y como se define
# en el archivo core_compile.yml
strip_comments:      on
# Máximo número de forwards que se realizan antes de lanzar una excepción
max_forwards:        5
# Constantes globales
path_info_array:      SERVER
path_info_key:        PATH_INFO
url_format:           PATH
```

El archivo `settings.yml` se utiliza para indicar las opciones de Symfony para una determinada aplicación. Como se vio en el capítulo 5, cuando se quieren añadir opciones específicas para una aplicación, lo mejor es añadirlas en el archivo de configuración `frontend/config/app.yml`. Este archivo también depende del entorno de ejecución, y las opciones definidas en ese archivo se pueden acceder con la clase `sfConfig` mediante el prefijo `app_`.

```
all:
  tarjetascredito:
    falsa:            off    # app_tarjetascredito_falsa
    visa:             on     # app_tarjetascredito_visa
    americanexpress:  on     # app_tarjetascredito_americanexpress
```

También se puede crear un archivo `app.yml` en el directorio de configuración del proyecto, lo que permite establecer opciones para todas las aplicaciones del proyecto. El mecanismo de configuración en cascada también se aplica a este archivo, por lo que las opciones establecidas en el archivo `app.yml` de cada aplicación tienen preferencia y pueden redefinir las opciones establecidas en el archivo `app.yml` del proyecto.

19.2. Extendiendo la carga automática de clases

La carga automática de clases, explicada brevemente en el capítulo 2, evita tener que incluir manualmente las clases que se utilizan en el código, siempre que esas clases se encuentren en un directorio específico. De esta forma, el framework se encarga automáticamente de cargar solamente las clases que hacen falta y de incluirlas en el momento en el que se necesitan.

El archivo `autoload.yml` almacena un listado de todas las rutas en las que se encuentran las clases que se cargan automáticamente. La primera vez que se procesa este archivo de configuración, Symfony recorre todos los directorios indicados en el archivo. Cada vez que se encuentra un archivo terminado en `.php` en alguno de estos directorios, la ruta del archivo y los nombres de las clases que se encuentran en el archivo se añaden a un listado interno de las clases que se cargan automáticamente. El listado se guarda en la cache, en un archivo llamado `config/config/autoload.yml.php`. Posteriormente, durante la ejecución de la aplicación, cuando se necesita una clase, Symfony busca en esta lista la ruta hasta la clase y añade el archivo `.php` de forma automática.

La carga automática de clases funciona para todos los archivos de tipo `.php` que contengan clases y/o interfaces.

Por defecto, las clases que se encuentran en los siguientes directorios de los proyectos se benefician directamente de la carga automática de clases:

- `miproyecto/lib/`
- `miproyecto/lib/model`
- `miproyecto/apps/frontend/lib/`
- `miproyecto/apps/frontend/modules/mimodulo/lib`

En el directorio de configuración de la aplicación, no existe por defecto un archivo llamado `autoload.yml`. Si se quieren modificar las opciones del framework, por ejemplo para cargar automáticamente las clases que se encuentran en otro directorio, se crea un archivo `autoload.yml` vacío y se redefinen las opciones del archivo `$sf_symfony_lib_dir/config/config/autoload.yml` o se crean nuevas opciones.

El archivo `autoload.yml` comienza con la clave `autoload:` e incluye un listado de los directorios en los que Symfony debe buscar las clases existentes. Para cada directorio se debe indicar una etiqueta, de forma que sea posible redefinir las opciones por defecto de Symfony. Para cada directorio se indica un nombre (`name`) (que aparecerá en forma de comentario en `config/autoload.yml.php`) y una ruta absoluta (`path`). A continuación, se define si la búsqueda que realiza Symfony debe ser recursiva (`recursive`) y por tanto, debe buscar archivos de tipo `.php` en todos los subdirectorios del directorio indicado; también se pueden indicar los subdirectorios que se excluyen (mediante `exclude`). El listado 19-4 muestra los directorios utilizados por defecto y la sintaxis empleada.

Listado 19-4 - Configuración por defecto de la carga automática de clases, en `$sf_symfony_lib_dir/config/config/autoload.yml`

```
autoload:
  # plugins
  plugins_lib:
    name:          plugins lib
    path:          %SF_PLUGINS_DIR%/*/lib
    recursive:     on

  plugins_module_lib:
    name:          plugins module lib
    path:          %SF_PLUGINS_DIR%/*/modules/*/lib
    prefix:        2
    recursive:     on

  # project
  project:
    name:          project
    path:          %SF_LIB_DIR%
    recursive:     on
    exclude:       [model, symfony]

  project_model:
    name:          project model
    path:          %SF_LIB_DIR%/model
```

```

    recursive:      on

# application
application:
  name:            application
  path:            %SF_APP_LIB_DIR%
  recursive:      on

modules:
  name:            module
  path:            %SF_APP_DIR%/modules/*/lib
  prefix:          1
  recursive:      on

```

Las rutas indicadas pueden utilizar comodines y también pueden utilizar los parámetros definidos en las clases de configuración para las rutas más utilizadas (que se explica en la siguiente sección). Si se utilizan estos valores en el archivo de configuración, se deben escribir en mayúsculas y encerrados por caracteres %

Aunque modificar el archivo `autoload.yml` permite indicar nuevas rutas en las que Symfony debe utilizar la carga automática de clases, también es posible extender el mecanismo utilizado por Symfony y añadir un gestor propio para realizar la carga automática de clases. Como Symfony utiliza la función estándar `spl_autoload_register()` para gestionar la carga automática de clases, resulta muy sencillo registrar otros gestores en la clase de configuración de la aplicación:

```

class frontendConfiguration extends sfApplicationConfiguration
{
    public function initialize()
    {
        parent::initialize(); // primero se carga Symfony

        // aquí se definen los gestores propios para la carga automática de clases
        spl_autoload_register(array('miToolkit', 'autoload'));
    }
}

```

Cuando el mecanismo de carga automática de clases de PHP encuentra una clase nueva, en primer lugar intenta utilizar el sistema de carga automática de clases de Symfony (utilizando los directorios definidos en el archivo `autoload.yml`). Si no encuentra la clase, prueba las otras funciones de carga automática registradas con `spl_autoload_register()`, hasta que encuentra la clase. Por tanto, se pueden añadir tantos mecanismos de carga automática de clases como sean necesarios, por ejemplo para proporcionar enlaces o *puentes* con los componentes de otros frameworks (ver capítulo 17).

19.3. Estructura de archivos propia

Cada vez que el framework requiere de una ruta para buscar algo (las clases internas de Symfony, las plantillas, los plugins, los archivos de configuración, etc.) utiliza una variable que almacena la ruta. Modificando el valor de estas variables, se puede modificar por completo la

estructura de directorios de un proyecto Symfony, para adaptarla a las necesidades específicas de cualquier cliente.

Sugerencia Aunque es posible modificar por completo la estructura de directorios de los proyectos Symfony, no se recomienda hacerlo. Uno de los puntos fuertes de los frameworks como Symfony es que cualquier programador puede comprender fácilmente cualquier proyecto desarrollado con Symfony, debido al uso de las convenciones. Por tanto, debe considerarse seriamente las ventajas y desventajas de modificar la estructura de directorios antes de hacerlo.

19.3.1. La estructura de archivos básica

Las variables que almacenan las rutas utilizadas se definen en las clases `sfProjectConfiguration` y `sfApplicationConfiguration` y se almacenan en el objeto `sfConfig`. El listado 19-5 muestra las variables que almacenan las rutas y el directorio al que hacen referencia.

Listado 19-5 - Variables de la estructura de archivos por defecto, definidas en `sfProjectConfiguration` y `sfApplicationConfiguration`

```
sf_root_dir           # myproject/
sf_apps_dir           # apps/
sf_app_dir             # frontend/
sf_app_config_dir      # config/
sf_app_i18n_dir        # i18n/
sf_app_lib_dir         # lib/
sf_app_module_dir     # modules/
sf_app_template_dir   # templates/
sf_cache_dir          # cache/
sf_app_base_cache_dir # frontend/
sf_app_cache_dir       # prod/
sf_template_cache_dir # templates/
sf_i18n_cache_dir      # i18n/
sf_config_cache_dir    # config/
sf_test_cache_dir      # test/
sf_module_cache_dir    # modules/
sf_config_dir          # config/
sf_data_dir            # data/
sf_doc_dir             # doc/
sf_lib_dir             # lib/
sf_log_dir             # log/
sf_test_dir            # test/
sf_plugins_dir         # plugins/
sf_web_dir             # web/
sf_upload_dir          # uploads/
```

Todas las rutas a los directorios principales de Symfony se obtienen a través de opciones acabadas en `_dir`. Siempre se deberían utilizar las variables en vez de las rutas reales (absolutas o relativas), de forma que se puedan modificar posteriormente si es necesario. Si se quiere por ejemplo mover un archivo al directorio `uploads/` de la aplicación, se debería utilizar como ruta el valor `sfConfig::get('sf_upload_dir')` en vez de `sfConfig::get('sf_root_dir').'/web/uploads/'`

19.3.2. Modificando la estructura de archivos

Si se desarrolla una aplicación para un cliente que ya dispone de una estructura de directorios definida y que no quiere cambiarla para adaptarse a Symfony, será necesario modificar la estructura de archivos por defecto. Redefiniendo el valor de la variable `sf_XXX_dir` mediante `sfConfig`, se puede conseguir que Symfony funcione correctamente con una estructura de directorios completamente diferente a la de por defecto. El mejor lugar para realizar esta modificación es la clase `ProjectConfiguration` de la aplicación para los directorios del proyecto y la clase `XXXConfiguration` para los directorios de las aplicaciones.

Si por ejemplo se necesita que todas las aplicaciones compartan un directorio común para los layouts de las plantillas, se añade la siguiente línea en el método `configure()` de la clase `ProjectConfiguration` para redefinir la opción `sf_app_template_dir`:

```
| sfConfig::set('sf_app_template_dir',  
| sfConfig::get('sf_root_dir').DIRECTORY_SEPARATOR.'templates');
```

Nota Aunque se puede modificar la estructura de directorios del proyecto mediante el método `sfConfig::set()`, es mejor utilizar los métodos definidos por las clases de configuración del proyecto y de las aplicaciones, ya que también se encargan de modificar todas las rutas relacionadas. El método `setCacheDir()` por ejemplo modifica los valores de `sf_cache_dir`, `sf_app_base_cache_dir`, `sf_app_cache_dir`, `sf_template_cache_dir`, `sf_i18n_cache_dir`, `sf_config_cache_dir`, `sf_test_cache_dir` y `sf_module_cache_dir`.

19.3.3. Modificando el directorio raíz del proyecto

Todas las rutas definidas en las clases de configuración se basan en el directorio raíz del proyecto, que se define en el archivo `ProjectConfiguration` incluido en el controlador frontal. Normalmente, el directorio raíz se encuentra un nivel por encima del directorio `web/`, pero se puede utilizar una estructura diferente. Si se utiliza una estructura principal de directorios formada por dos directorios, uno puede ser el directorio público y otro el privado, tal y como muestra el listado 19-7. Esta estructura es muy típica cuando se utiliza un servicio de hosting compartido.

Listado 19-7 - Ejemplo de estructura de directorios propia en un hosting compartido

```
| symfony/      # Area privada  
|   apps/  
|   config/  
|   ...  
| www/         # Area pública  
|   images/  
|   css/  
|   js/  
|   index.php
```

En este caso, el directorio raíz sería el directorio `symfony/`. De esta forma, para que la aplicación funcione correctamente, en el controlador frontal `index.php` se debe incluir el archivo `config/ProjectConfiguration.class.php` de la siguiente forma:

```
| require_once(dirname(__FILE__).'/../symfony/config/ProjectConfiguration.class.php');
```

Además, como el área pública es `www/` en vez del tradicional `web/`, se debe redefinir su valor con el método `setWebDir()`:

```
class ProjectConfiguration extends sfProjectConfiguration
{
    public function configure()
    {
        // ...

        $this->setWebDir($this->getRootDir().'../www');
    }
}
```

19.3.4. Enlazando las librerías de Symfony

La ruta a los archivos del framework se define en la clase `ProjectConfiguration`, que se encuentra en el directorio `config/`, tal y como se muestra en el listado 19-8.

Listado 19-8 - Las ruta a los archivos del framework, en `miproyecto/config/ProjectConfiguration.class.php`

```
<?php

require_once '/ruta/hasta/symfony/lib/autoload/sfCoreAutoload.class.php';
sfCoreAutoload::register();

class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
    }
}
```

La ruta se inicializa cuando se ejecuta la tarea `php symfony generate:project` desde la línea de comandos y hace referencia a la instalación de Symfony que se ha utilizado para construir el proyecto. La ruta se utilizan tanto en la línea de comandos como en la arquitectura MVC.

Por tanto, se puede utilizar otra instalación de Symfony simplemente modificando la ruta a los archivos de Symfony.

Aunque esta ruta puede ser absoluta, también es posible utilizar `dirname(FILE)` para hacer referencia a archivos dentro de la estructura del proyecto y para mantener la independencia respecto al directorio elegido para instalar el proyecto. Muchos proyectos prefieren por ejemplo que el directorio `lib/` de Symfony aparezca como un enlace simbólico en el directorio `lib/vendor/symfony/` del proyecto:

```
miproyecto/
  lib/
    vendor/
      symfony/ => /ruta/hasta/symfony/
```

En este caso, la clase `ProjectConfiguration` sólo debe definir el directorio `lib/` de Symfony de la siguiente manera:

```
<?php

require_once dirname(__FILE__).'/../lib/vendor/symfony/lib/autoload/
sfCoreAutoload.class.php';
sfCoreAutoload::register();

class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
    }
}
```

El mismo principio se aplica si se quieren incluir los archivos de Symfony como `svn:externals` en el directorio `lib/vendor/` del proyecto:

```
miproyecto/
  lib/
    vendor/
      svn:externals symfony http://svn.symfony-project.com/branches/1.1
```

Sugerencia En ocasiones, los diferentes servidores que ejecutan las aplicaciones no tienen las librerías de Symfony en las mismas rutas. Una forma de conseguirlo es excluir el archivo `ProjectConfiguration.class.php` del proceso de sincronización (añadiéndolo a la lista del archivo `rsync_exclude.txt`). Otra forma de hacerlo es mantener las mismas rutas en la versión de desarrollo y en la versión de producción del archivo `ProjectConfiguration.class.php` y que las rutas apunten a enlaces simbólicos que cambian en cada servidor.

19.4. Comprendiendo el funcionamiento de los manejadores de configuración

Cada archivo de configuración dispone de su propio manejador. La responsabilidad de los manejadores de configuración consiste en la gestión de la configuración en cascada y la transformación de los archivos de configuración en archivos PHP optimizados para ser utilizados durante la ejecución de la aplicación.

19.4.1. Manejadores de configuración por defecto

La configuración de los manejadores por defecto se guarda en el archivo `$sf_symfony_lib_dir/config/config/config_handlers.yml`. En este archivo se relacionan los manejadores y los archivos de configuración según su ruta. El listado 19-9 muestra un extracto de este archivo.

Listado 19-9 - Extracto de `$sf_symfony_lib_dir/config/config/config_handlers.yml`

```
config/settings.yml:
  class:    sfDefineEnvironmentConfigHandler
  param:
    prefix: sf_

config/app.yml:
  class:    sfDefineEnvironmentConfigHandler
  param:
```

```
    prefix: app_

config/filters.yml:
  class:    sfFilterConfigHandler

modules/*/config/module.yml:
  class:    sfDefineEnvironmentConfigHandler
  param:
    prefix: mod_
    module: yes
```

Para cada archivo de configuración (`config_handlers.yml` identifica cada archivo mediante una ruta que puede hacer uso de comodines) se especifica bajo la clave `class` la clase del manejador que se debe utilizar.

Las opciones de los archivos de configuración manejados por `sfDefineEnvironmentConfigHandler` se pueden acceder directamente desde el código de la aplicación mediante la clase `sfConfig`, utilizando como prefijo el valor indicado en la clave `param/prefix`.

Se pueden modificar o crear nuevos manejadores para procesar cada archivo de configuración, de forma que por ejemplo se puedan utilizar archivos de tipo INI o XML en vez de archivos YAML.

Nota El manejador del archivo de configuración `config_handlers.yml` se denomina `sfRootConfigHandler` y, obviamente, no se puede modificar.

Si se necesita cambiar la forma en la que se procesa la configuración, se crea un archivo vacío llamado `config_handlers.yml` en el directorio `config/` de la aplicación y se redefine el valor de las líneas `class` por las clases propias que se han creado.

19.4.2. Creando un manejador propio

El uso de manejadores para procesar los archivos de configuración implica 2 grandes ventajas:

- El archivo de configuración se transforma en código PHP ejecutable y este código se guarda en la cache. Esto significa que en producción, la configuración se procesa una sola vez y el rendimiento es óptimo.
- El archivo de configuración se puede definir en varios niveles (proyecto y aplicación) y los valores finales de los parámetros dependen de la configuración en cascada. De esta forma se pueden definir parámetros a nivel de proyecto y redefinir su valor en cada aplicación.

Si se quiere crear un manejador propio, se puede seguir como ejemplo la estructura utilizada por el framework en el directorio `$sf_symfony_lib_dir/config/`.

En el siguiente ejemplo, se supone que la aplicación dispone de una clase llamada `myMapAPI`, que proporciona una interfaz con un servicio web externo de mapas. Como muestra el listado 19-10, esta clase se debe inicializar con una URL y un nombre de usuario.

Listado 19-10 - Ejemplo de inicialización de la clase `myMapAPI`


```
$mapApi = new myMapAPI();
$mapApi->setUrl($url);
$mapApi->setUser($usuario);
```

Estos dos parámetros de configuración se pueden guardar en un archivo de configuración específico llamado `map.yml` y guardado en el directorio `config/`. El contenido de este archivo de configuración puede ser:

```
api:
  url:  map.api.ejemplo.com
  user: foobar
```

Para transformar estas opciones de configuración en un código equivalente al del listado 19-9, se debe crear un manejador de archivos de configuración. Todos los manejadores definidos deben extender la clase `sfConfigHandler` y deben proporcionar un método llamado `execute()`, que espera como parámetro un array de rutas a archivos de configuración y que devuelve los datos que se deben escribir en un archivo de la cache. Los manejadores de archivos de tipo YAML deberían extender la clase `sfYamlConfigHandler`, que proporciona algunas utilidades para el procesamiento de archivos YAML. Para el archivo `map.yml` anterior, el manejador de configuración más típico sería el que se muestra en el listado 19-11.

Listado 19-11 - Un manejador de configuraciones propio, en `frontend/lib/myMapConfigHandler.class.php`

```
<?php

class myMapConfigHandler extends sfYamlConfigHandler
{
    public function execute($configFiles)
    {
        // Procesar el archivo YAML
        $config = $this->parseYamls($configFiles);

        $data = "<?php\n";
        $data. = "\$mapApi = new myMapAPI();\n";

        if (isset($config['api']['url'])
        {
            $data. = sprintf("\$mapApi->setUrl('%s');\n", $config['api']['url']);
        }

        if (isset($config['api']['user'])
        {
            $data. = sprintf("\$mapApi->setUser('%s');\n", $config['api']['user']);
        }

        return $data;
    }
}
```

El array `$configFiles` que pasa Symfony al método `execute()` contiene una ruta a cada archivo `map.yml` que se encuentre en los directorios `config/`. El método `parseYamls()` se encarga de realizar la configuración en cascada.

Para asociar este nuevo manejador con los archivos de tipo `map.yml`, se crea un nuevo archivo de configuración `config_handlers.yml` con el siguiente contenido:

```
config/map.yml:
  class: myMapConfigHandler
```

Nota La clase indicada en `class` debe cargarse de forma automática (como en este caso) o encontrarse en el archivo cuya ruta se indica en el parámetro `file` bajo la clave `param`.

Como sucede con muchos otros archivos de configuración de Symfony, también se puede registrar un manejador de configuración directamente desde el código PHP:

```
$sfContext::getInstance()->getConfigCache()->registerConfigHandler('config/map.yml',
myMapConfigHandler, array());
```

Cuando se necesita el código basado en el archivo `map.yml` y que ha generado el manejador `myMapConfigHandler`, se puede ejecutar la siguiente instrucción dentro del código de la aplicación:

```
include($sfContext::getInstance()->getConfigCache()->checkConfig('config/map.yml'));
```

Cuando se ejecuta el método `checkConfig()`, Symfony busca todos los archivos `map.yml` existentes en los directorios de configuración y los procesa con el manejador especificado en el archivo `config_handlers.yml` si no existe el archivo `map.yml.php` correspondiente en la cache o si el archivo `map.yml` es más reciente que el de la cache.

Sugerencia Si se quieren soportar diferentes entornos en un archivo de configuración YAML, su manejador debe extender la clase `sfDefineEnvironmentConfigHandler` en vez de la clase `sfYamlConfigHandler`. Después de ejecutar el método `parseYaml()` para obtener la configuración, se debe ejecutar el método `mergeEnvironment()`. Este proceso se puede hacer en una sola línea mediante `$config = $this->mergeEnvironment($this->parseYamls($configFiles));`.

Si solamente se necesita que los usuarios puedan obtener los valores desde el código mediante `sfConfig`, se puede utilizar la clase del manejador de configuración `sfDefineEnvironmentConfigHandler`. Si por ejemplo se quiere obtener el valor de los parámetros `url` y `user` mediante `sfConfig::get('map_url')` y `sfConfig::get('map_user')`, se puede definir el manejador de la siguiente forma:

```
config/map.yml:
  class: sfDefineEnvironmentConfigHandler
  param:
    prefix: map_
```

Se debe tener cuidado en no utilizar un prefijo que ya se esté utilizando por otro manejador. Los prefijos existentes por defecto son `sf_`, `app_` y `mod_`.

19.5. Resumen

Los archivos de configuración pueden modificar por completo el funcionamiento del framework. Como Symfony utiliza la configuración incluso para sus características internas y para la carga de los archivos, se puede adaptar fácilmente a muchos más entornos que los tradicionales hostings dedicados.

Esta gran "*configurabilidad*" es uno de los puntos fuertes de Symfony. Aunque a veces echa para atrás a los programadores que están empezando con Symfony, porque son muchos archivos de configuración y hay que aprender muchas convenciones, lo cierto es que permite que las aplicaciones Symfony sean compatibles con un gran número de sistemas y entornos diferentes. Una vez que se dominan los archivos de configuración de Symfony, se pueden ejecutar las aplicaciones en cualquier servidor del mundo.