El tutorial Jobeet

Fabien Potencier

Sobre este libro...

- **Esta versión impresa se creó el 30 de marzo de 2009 y todavía está incompleta**. La versión más actualizada de los contenidos de este libro se puede encontrar en http://www.librosweb.es/jobeet

- Si quieres aportar sugerencias, comentarios, críticas o informar sobre errores, puedes enviarnos un mensaje a **contacto@librosweb.es**

Capítulo 1. Comenzando el proyecto

1.1. Introducción

El framework Symfony comenzó hace más de tres años como un proyecto de software libre y se ha convertido en uno de los frameworks de PHP más populares gracias a sus características avanzadas y su gran documentación. Y esto último ha sido así desde el principio.

En diciembre de 2005, justo después de publicar la primera versión oficial de Symfony, se publicó el tutorial Askeet (http://www.symfony-project.org/askeet/1_0/en/) , un conjunto de 24 tutoriales que se publicaron todos los días desde el 1 de diciembre hasta el día de Navidad.

Ese tutorial se ha convertido en una herramienta muy valiosa para promocionar el uso del framework entre los principiantes. Muchos programadores han aprendido a desarrollar aplicaciones con Symfony gracias al tutorial Askeet y muchas empresas siguen utilizándolo como su principal herramienta de formación.

No obstante, el tutorial Askeet se ha quedado un poco obsoleto y aprovechando el lanzamiento de Symfony 1.2, hemos decidido publicar un nuevo tutorial llamado Jobeet y que también está dividido en 24 capítulos.

El tutorial original se publicó durante 24 días seguidos en el blog oficial de Symfony y lo que estás leyendo es su adaptación al formato de un libro.

1.2. El desafío

Cada capítulo está preparado para que dure una hora y para que aprendas a programar con Symfony creando un sitio web real, desde el principio hasta el final.

Si multiplicas una hora por los 24 tutoriales que se van a publicar, el resultado es 24 horas o un día, que es el tiempo que creemos que necesita un programador para aprender los fundamentos de Symfony. Cada día se añadirán características a la aplicación, lo que va a permitir presentar algunas de las nuevas características de Symfony y algunas de las mejores prácticas en el desarrollo profesional de aplicaciones Symfony.

En el tutorial de Askeet, decidimos que el tema del día 21 lo eligieran los usuarios. La iniciativa fue un éxito rotundo y la comunidad de usuarios decidió que añadiéramos un buscador a la aplicación. Y lo añadimos. El tutorial del día 21 se ha convertido además en uno de los tutoriales de Askeet más famosos.

Durante la publicación del tutorial Jobeet, celebramos la llegada del invierno el día 21 de diciembre celebrando un concurso para elegir el diseño gráfico de la aplicación. El diseño ganador fue el creado por la empresa `centre{source}` y es el que se utiliza en

este tutorial como diseño por defecto. Además, este diseño es el que está disponible en el sitio web oficial de Jobeet.

1.3. Este tutorial es diferente

¿Recuerdas cómo fueron los primeros días de PHP4? ¡La época dorada del desarrollo web! PHP fue uno de los primeros lenguajes específicamente pensados para la web y uno de los más sencillos de aprender.

Sin embargo, como las tecnologías web evolucionan muy rápidamente, los programadores web tienen que reciclarse y adaptarse a las últimas herramientas y buenas prácticas disponibles. La mejor forma de aprender consiste normalmente en leer blogs, tutoriales y libros. Nosotros mismos hemos leído muchos libros y blogs sobre PHP, Python, Java, Ruby y Perl y nos hemos dado cuenta de que la mayoría se quedan atrás cuando el autor empieza a mostrar trozos de código.

¿Quién no ha leído frases como las siguientes?

- *En una aplicación real no te olvides de incluir la validación de los datos y la gestión de los errores*.
- *Todo lo referente a la seguridad se deja como ejercicio a desarrollar por el lector*.
- *Además sería necesario crear las pruebas unitarias*.

¿Cómo es posible? Estamos hablando de aplicaciones profesionales y todo lo anterior es seguramente la parte más importante de cualquier aplicación. Como lector te sientes abandonado, ya que los ejemplos no son muy útiles cuando no tienen en cuenta todo lo anterior. No puedes tomar esos ejemplos como tu punto de partida porque la seguridad, validación, gestión de errores y pruebas unitarias, entre muchos otros, son los que aseguran que tu código sea correcto.

A lo largo de este tutorial nunca te encontrarás con frases de ese tipo, ya que vamos a crear pruebas unitarias, vamos a gestionar correctamente los errores, vamos a incluir validación de datos y por supuesto vamos a crear una aplicación muy segura. Todo esto es así porque Symfony no sólo consiste en código PHP, sino que también consiste en utilizar las mejores prácticas para crear aplicaciones profesionales para el mundo empresarial. Además, podemos dedicarnos a incluir todas esas cosas porque Symfony ya dispone de todas las herramientas necesarias para incluir cada una de ellas sin necesidad de escribir mucho código.

La validación, la gestión de errores, las pruebas y la seguridad están completamente integrados en Symfony, por lo que su explicación será muy sencilla. Esta es una más de las razones por las que se debería utilizar un framework para desarrollar proyectos del mundo real.

Todo el código que incluye este tutorial es código que se puede utilizar directamente en aplicaciones reales, por lo que te animamos a que copies y pegues trozos de código o que directamente copies partes enteras de la aplicación.

1.4. El proyecto

La aplicación que vamos a construir podía haber sido otro gestor de blogs, pero queríamos emplear Symfony para crear un proyecto realmente útil. Nuestro objetivo es demostrar que se pueden desarrollar aplicaciones profesionales con estilo y poco esfuerzo.

Vamos a mantener en secreto durante un día más el objetivo del proyecto, ya que tenemos que hacer muchas cosas durante este primer día. De todas formas, no es difícil adivinar el propósito del proyecto porque ya conoces su nombre: **Jobeet**.

1.5. ¿Que haremos hoy?

Como 24 horas es mucho tiempo para desarrollar una aplicación con Symfony, no vamos a escribir nada de código PHP durante este primer día. Aunque no escribamos ni una sola línea de código, hoy comprenderás las ventajas de utilizar un framework como Symfony simplemente al iniciar el desarrollo del proyecto.

Nuestro objetivo durante este día consiste en configurar el entorno de desarrollo y mostrar una página de la aplicación en el navegador. Para ello es necesario instalar Symfony, crear una aplicación y configurar un servidor web.

1.6. Prerrequisitos

En primer lugar, es imprescindible que cuentes con un entorno de desarrollo web que funcione correctamente y esté formado por un servidor web (Apache por ejemplo), un gestor de bases de datos (MySQL, PostgreSQL o SQLite por ejemplo) y PHP versión 5.2.4 o superior.

Como vamos a utilizar mucho la línea de comandos, te aconsejamos que utilices un sistema operativo tipo Unix. No obstante, todo lo que vamos a ver también funciona perfectamente en Windows, por lo que puedes ejecutar los comandos en la consolas `cmd`.

> **Nota**
>
> Los comandos de las consolas tipo Unix te pueden venir muy bien en un entorno Windows. Si quieres hacer uso de comandos como `tar`, `gzip` o `grep` en Windows, puedes instalar Cygwin (http://cygwin.com/) . Como la documentación oficial es muy escasa, te aconsejamos que utilices alguna buena guía de instalación de Cygwin (http://www.soe.ucsc.edu/~you/notes/cygwin-install.html) . Si eres de los valientes, también puedes probar los *Windows Services for Unix* (http://technet.microsoft.com/en-gb/interopmigration/bb380242.aspx) de Microsoft.

Como este tutorial sólo se centra en el framework Symfony, suponemos que tienes unos sólidos conocimientos de PHP 5 y de la programación orientada a objetos.

1.7. Instalación de Symfony

En primer lugar, crea un directorio donde vamos a guardar todos los archivos relacionados con el proyecto Jobeet:

```
$ mkdir -p /home/sfprojects/jobeet
$ cd /home/sfprojects/jobeet
```

En Windows utiliza los siguientes comandos:

```
c:\> mkdir c:\development\sfprojects\jobeet
c:\> cd c:\development\sfprojects\jobeet
```

Nota

Recomendamos a los usuarios de Windows que ejecuten Symfony y creen su proyecto en una ruta que no tenga espacios en blanco. Por tanto, evita directorios como `Documents and Settings` y `Mis Documentos`.

Crea un directorio para guardar los archivos de las librerías del framework Symfony:

```
$ mkdir -p lib/vendor
```

El sitio web oficial de Symfony dispone de una página de instalación (http://www.symfony-project.org/installation) que muestra todas las versiones de Symfony disponibles y compara sus características. Como este tutorial se ha escrito para Symfony 1.2, accede a la página de instalación de Symfony 1.2 (http://www.symfony-project.org/installation/1_2) .

Dentro de la sección *"Source Download"* encontrarás el archivo comprimido en formato `.tgz` o en formato `.zip`. Descarga el archivo, guárdalo en el directorio `lib/vendor` recién creado y descomprímelo:

```
$ cd lib/vendor
$ tar zxpf symfony-1.2.2.tgz
$ mv symfony-1.2.2 symfony
$ rm symfony-1.2.2.tgz
```

Si utilizas Windows puedes descomprimir el archivo ZIP directamente desde el explorador de archivos. Después de cambiar el nombre del directorio a `symfony`, deberías tener el siguiente directorio `c:\development\sfprojects\jobeet\lib\vendor\symfony`.

Como la configuración de PHP varía mucho de una distribución a otra, es necesario comprobar que la configuración actual de PHP cumple con los requisitos mínimos exigidos por Symfony. Para realizar esta comprobación puedes utilizar un script específico que incluye Symfony:

```
$ cd ../..
$ php lib/vendor/symfony/data/bin/check_configuration.php
```

Si se produce cualquier error, el script anterior muestra mensajes de ayuda para solucionarlos. También es recomendable que ejecutes el script de comprobación desde

un navegador, ya que la configuración de PHP puede ser diferente. Copia el script en el directorio raíz del servidor y accede a ese archivo desde el navegador. No te olvides de borrar el archivo que acabas de copiar después de realizar la comprobación:

```
$ rm web/check_configuration.php
```

```
********************************
*                              *
*  symfony requirements check  *
*                              *
********************************

php.ini used by PHP: /apache2/php/etc/php.ini


** Mandatory requirements **

  OK        requires PHP >= 5.2.4
  OK        php.ini: requires zend.zel_compatibility_mode set to off

** Optional checks **

  OK        PDO is installed
  OK        PDO has some drivers installed: sqlite2, sqlite, mysql
  OK        can use token_get_all()
  OK        can use mb_strlen()
  OK        can use iconv()
  OK        can use utf8_decode()
  OK        has a PHP accelerator
  OK        php.ini: magic_quotes_gpc set to off
  OK        php.ini: register_globals set to off
  OK        php.ini: session.auto_start set to off
```

Figura 1.1. Comprobando la configuración

Si el script anterior no muestra ningún mensaje de error, comprueba que has instalado Symfony correctamente utilizando la línea de comandos para mostrar la versión de Symfony que se ha instalado (en el siguiente comando la letra `V` se escribe en mayúscula):

```
$ cd ../..
$ php lib/vendor/symfony/data/bin/symfony -V
```

En Windows:

```
c:\> cd ..\..
c:\> php lib\vendor\symfony\data\bin\symfony -V
```

Si sientes curiosidad por los comandos que incluye esta utilidad de la línea de comandos, puedes ejecutarla sin opciones (simplemente escribiendo `symfony`) para que muestre todos los comandos disponibles:

```
$ php lib/vendor/symfony/data/bin/symfony
```

En Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony
```

La línea de comandos es imprescindible para los programadores, ya que proporciona muchas utilidades que mejoran la productividad al realizar tareas tan comunes como limpiar la cache, generar código de forma automática, etc.

1.8. Preparar el proyecto

En Symfony, las **aplicaciones** que comparten el mismo modelo de datos se agrupan en **proyectos**. El proyecto Jobeet dispone de dos aplicaciones diferentes: un frontend y un backend.

1.8.1. Crear el proyecto

Dentro del directorio `jobeet`, ejecuta la tarea `generate:project` para crear la estructura de directorios del proyecto:

```
$ php lib/vendor/symfony/data/bin/symfony generate:project jobeet
```

En Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony generate:project jobeet
```

La tarea `generate:project` genera la estructura de directorios y archivos por defecto necesarios para un proyecto Symfony:

Directorio	Descripción
`apps/`	Se encuentran los archivos y directorios de las aplicaciones
`cache/`	Los archivos que el framework guarda en la cache
`config/`	Los archivos de configuración del proyecto
`lib/`	Las librerías y clases del proyecto
`log/`	Los archivos de log del framework
`plugins/`	Los plugins instalados
`test/`	Los archivos de las pruebas unitarias y funcionales
`web/`	El directorio web raíz

Nota

¿Por qué Symfony genera tantos archivos? Una de las principales ventajas de utilizar un framework completo es que puedes estandarizar tus desarrollos. Gracias a la estructura de archivos y directorios por defecto de Symfony, cualquier programador con ciertos conocimientos de Symfony es capaz de continuar el desarrollo de cualquier proyecto Symfony. En cuestión de minutos será capaz de profundizar en el código, solucionar errores y añadir nuevas características.

La tarea `generate:project` también genera un atajo para el comando `symfony` dentro del directorio raíz del proyecto Jobeet para reducir la longitud de los comandos que tienes que escribir al ejecutar una tarea de Symfony.

Por tanto, a partir de este momento ya no vamos a utilizar la ruta completa hasta el comando `symfony`, sino que se utilizará directamente el atajo `symfony`.

1.8.2. Crear la aplicación

Ahora ya puedes crear la aplicación `frontend` ejecutando la tarea `generate:app`:

```
$ php symfony generate:app --escaping-strategy=on --csrf-secret=UniqueSecret
frontend
```

Sugerencia

Como el archivo `symfony` es ejecutable, los usuarios de Unix pueden utilizar `./symfony` en vez de `php symfony`. Si utilizas Windows, copia el archivo `symfony.bat` en tu proyecto y utiliza el comando `symfony` en vez de `php symfony`:

```
c:\> copy lib\vendor\symfony\data\bin\symfony.bat .
```

En función del nombre de la aplicación indicado como argumento, la tarea `generate:app` crea en el directorio `apps/frontend` la estructura de directorios por defecto que necesita la aplicación:

Directorio	Descripción
`config/`	Los archivos de configuración de la aplicación
`lib/`	Las librerías y clases de la aplicación
`modules/`	El código de la aplicación (MVC)
`templates/`	Los archivos de las plantillas globales

Sugerencia

Todos los comandos de `symfony` se deben ejecutar en el directorio raíz del proyecto salvo que se indique lo contrario de forma explícita.

Cuando se ejecuta la tarea `generate:app`, se han incluido dos opciones relacionadas con la seguridad:

- `--escaping-strategy`: activa el mecanismo de escape para evitar ataques de tipo XSS (*Cross Site Scripting*).
- `--csrf-secret`: activa los tokens de sesión en los formularios para evitar ataques de tipo CSRF (*Cross Site Request Forgery*).

Utilizando estos dos argumentos opcionales en la tarea `generate:app`, hemos añadido la seguridad necesaria para contrarrestar las dos vulnerabilidades más extendidas en la web. En efecto, Symfony se encarga de proteger automáticamente nuestra aplicación frente a estos tipos de ataque.

Nota

> Si desconoces los ataques de tipo XSS (http://es.wikipedia.org/wiki/XSS) o CSRF (http://en.wikipedia.org/wiki/CSRF) , puede ser interesante que dediques un tiempo a estudiar el funcionamiento de estas vulnerabilidades.

1.8.3. La ruta de Symfony

Para obtener la versión de Symfony que utiliza tu proyecto, puedes utilizar el siguiente comando:

```
$ php symfony -V
```

La opción `-V` también muestra la ruta completa hasta el directorio de instalación de Symfony, que se encuentra en el archivo de configuración `config/ProjectConfiguration.class.php`:

```
// config/ProjectConfiguration.class.php
require_once '/Users/fabien/work/symfony/dev/1.2/lib/autoload/
sfCoreAutoload.class.php';
```

Para que el proyecto sea más portable, es recomendable cambiar la ruta absoluta por una ruta relativa:

```
// config/ProjectConfiguration.class.php
require_once dirname(__FILE__).'/../lib/vendor/symfony/lib/autoload/
sfCoreAutoload.class.php';
```

De esta forma, ahora puedes colocar el directorio del proyecto Jobeet en cualquier otro directorio del servidor y todo seguirá funcionando correctamente.

1.9. Los entornos

Si echas un vistazo al directorio `web/`, verás dos archivos PHP llamados `index.php` y `frontend_dev.php`. Estos archivos se conocen con el nombre de **controladores frontales**, ya que todas las peticiones de la aplicación se realizan a través de ellos. Pero, ¿por qué tenemos dos controladores frontales si sólo tenemos una aplicación?

Los dos archivos apuntan a la misma aplicación pero se utilizan en diferentes entornos. Cuando se desarrolla una aplicación, salvo que la desarrolles directamente sobre el servidor de producción, necesitas varios entornos:

- El **entorno de desarrollo**: este es el entorno que utilizan los programadores web cuando modifican la aplicación para añadir nuevas características y corregir errores.
- El **entorno de pruebas**: este entorno se utiliza para ejecutar automáticamente las pruebas unitarias.
- El **entorno intermedio** (o *entorno "staging"*): este entorno lo utiliza el cliente para probar la aplicación e informar sobre los errores que ha encontrado o las características que le faltan a la aplicación.

- El **entorno de producción**: este es el entorno en el que se ejecuta la aplicación que utilizan los usuarios finales.

¿Qué es lo que diferencia a cada entorno? En el entorno de desarrollo es necesario por ejemplo que la aplicación guarde en el log todos los detalles de cada aplicación para simplificar la depuración, pero la cache tiene que estar deshabilitada para que cualquier cambio realizado se tenga en cuenta de forma instantánea. Por tanto, el entorno de desarrollo se debe optimizar para el programador. El ejemplo más claro es cuando se produce una excepción. Para que el programador detecte lo antes posible la causa del error, Symfony muestra directamente en el navegador toda la información disponible sobre la petición actual:

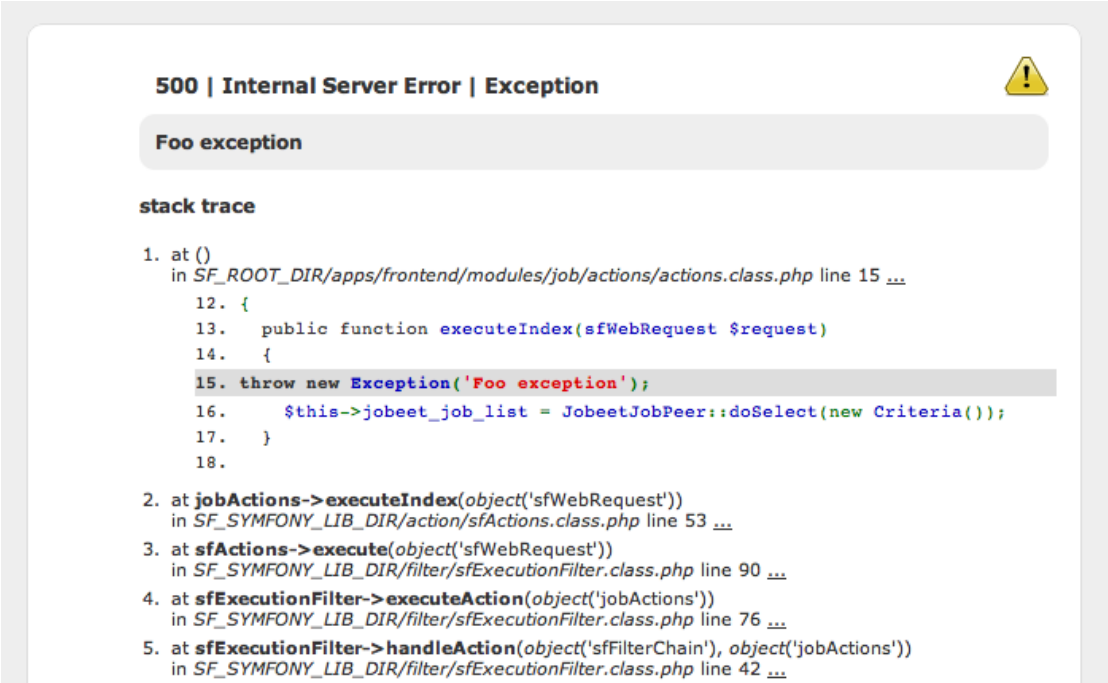

Figura 1.2. Una excepción en el entorno de desarrollo

Por otra parte, en el entorno de producción la cache debe estar habilitada y por supuesto se deben mostrar mensajes de error propios en vez de la información relacionada con la excepción producida. Por tanto, el entorno de producción debe estar optimizado para obtener el máximo rendimiento y para conseguir la mejor experiencia de usuario.

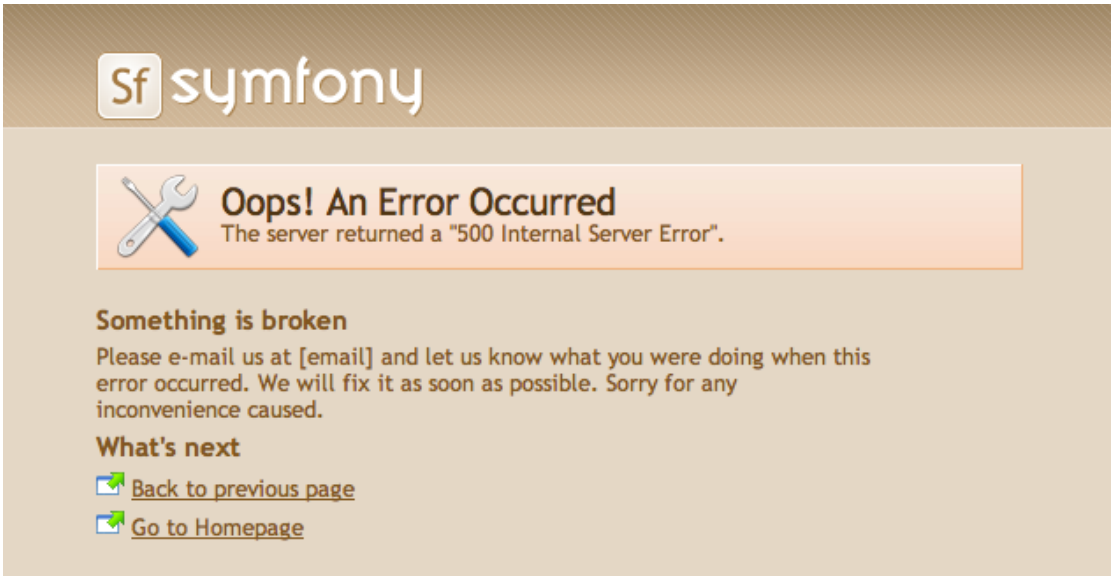

Figura 1.3. Una excepción en el entorno de producción

Un entorno en Symfony no es más que un conjunto específico de opciones de configuración. El framework Symfony incluye por defecto tres entornos llamados `dev`, `test` y `prod`. Durante el tutorial del día 22 aprenderás a crear nuevos entornos, como por ejemplo el entorno `staging`.

Si abres los archivos de los controladores frontales, verás que su contenido es el mismo salvo la opción que indica el entorno utilizado:

```
// Archivo web/index.php
<?php

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

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

Nota

Definir un nuevo entorno en Symfony es tan sencillo como crear un nuevo controlador frontal. Más adelante se muestra cómo modificar las opciones de un entorno.

1.10. Configurar mal el servidor web

En la sección anterior se creó un directorio que contiene todos los archivos y directorios del proyecto Jobeet. Si has creado ese directorio bajo el directorio raíz del servidor web, ya puedes acceder a tu proyecto mediante un servidor web.

Obviamente, como no es necesario realizar ninguna modificación, es un método muy rápido de tener un proyecto Symfony listo para ser utilizado. Si ahora intentas acceder por ejemplo al archivo `config/databases.yml` desde tu navegador entenderás las consecuencias tan negativas que tiene no dedicar unos minutos a configurar

correctamente el servidor web. Si el usuario que accede a tu sitio web sabe que está desarrollado con Symfony, tendrá acceso a muchos archivos con información sensible.

Nunca jamás instales tus proyectos de esta forma en un servidor de producción y lee la siguiente sección para aprender a configurar correctamente el servidor web.

1.11. Configurar correctamente el servidor web

Una buena práctica web consiste en colocar en el directorio raíz del servidor web solamente los archivos que necesitan los navegadores, como las hojas de estilos, los archivos JavaScript y las imágenes. Nuestra recomendación es que guardes todos estos archivos en el subdirectorio `web/` del proyecto Symfony.

Si echas un vistazo a este directorio, verás algunos subdirectorios creados para cada tipo de archivo (`css/` y `images/`) y los archivos de los dos controladores frontales. Estos dos controladores frontales son los únicos archivos PHP que deben encontrarse bajo el directorio raíz del servidor web. El resto de archivos PHP se pueden ocultar a los navegadores, lo que es una buena idea desde el punto de vista de la seguridad.

1.11.1. Configuración del servidor web

A continuación debes modificar la configuración de Apache para hacer accesible el proyecto a cualquier usuario del mundo.

Localiza el archivo de configuración `httpd.conf` y añade lo siguiente justo al final del archivo:

```
# Asegúrate de que sólo tienes esta línea una vez en todo el archivo de
configuración
NameVirtualHost 127.0.0.1:8080

# Esta es la configuración para Jobeet
Listen 127.0.0.1:8080

<VirtualHost 127.0.0.1:8080>
  DocumentRoot "/home/sfprojects/jobeet/web"
  DirectoryIndex index.php
  <Directory "/home/sfprojects/jobeet/web">
    AllowOverride All
    Allow from All
  </Directory>

  Alias /sf /home/sfprojects/jobeet/lib/vendor/symfony/data/web/sf
  <Directory "/home/sfprojects/jobeet/lib/vendor/symfony/data/web/sf">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

Nota

El alias `/sf` se necesita para las imágenes y archivos JavaScript que utilizan las páginas por defecto de Symfony y la barra de depuración web.

En Windows reemplaza la línea `Alias` por algo como lo siguiente:

```
Alias /sf "c:\development\sfprojects\jobeet\lib\vendor\symfony\data\web\sf"
```

Además, la ruta `/home/sfprojects/jobeet/web` se debe sustituir por algo como lo siguiente `c:\development\sfprojects\jobeet\web`

La configuración anterior hace que Apache espere las peticiones en el puerto `8080` de tu máquina, por lo que el sitio web de Jobeet se puede acceder en la siguiente URL:

```
http://localhost:8080/
```

Puedes sustituir `8080` por cualquier otro número que prefieras, pero se recomienda utilizar un número mayor que `1024` para que no tengas que utilizar permisos de administrador.

Utilizar un dominio propio para Jobeet

Si eres el administrador de tu máquina, es mucho mejor crear *virtual hosts* en vez de utilizar un nuevo puerto cada vez que creas un proyecto. En vez de elegir un puerto y añadir una directiva `Listen`, escoge un nombre de dominio y añade una directiva `ServerName`:

```
# Esta es la configuración para Jobeet
<VirtualHost 127.0.0.1:80>
  ServerName jobeet.localhost
  <!-- aquí incluye la misma configuración que antes -->
</VirtualHost>
```

El nombre de dominio `jobeet.localhost` que utiliza la configuración de Apache lo tienes que registrar de forma local. Si utilizas un sistema operativo tipo Linux, debes añadirlo en el archivo `/etc/hosts`. Si utilizas Windows XP, este archivo se encuentra en el directorio `C:\WINDOWS\system32\drivers\etc\`. En cualquier caso, añade la siguiente línea:

```
127.0.0.1         jobeet.localhost
```

1.11.2. Probar la nueva configuración

Reinicia el servidor web Apache y comprueba que puedes acceder a la aplicación abriendo un navegador y accediendo a la URL http://localhost:8080/index.php/ o http://jobeet.localhost/index.php/ dependiendo de la configuración de Apache que elegiste en la sección anterior.

Figura 1.4. Página de bienvenida de Symfony

Nota

Si tienes el módulo `mod_write` correctamente instalado en Apache, puedes eliminar la parte `index.php/` de todas las URL. El motivo es que Symfony crea el archivo `web/.htaccess` que ya incluye las reglas necesarias para reescribir las URL.

También puedes probar a acceder a la aplicación en el entorno de desarrollo. Para ello, accede a la siguiente URL:

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

La principal diferencia es que ahora se muestra la barra de depuración web en la esquina superior derecha, incluyendo unos pequeños iconos si has configurado correctamente el alias para la ruta `sf/`.

Figura 1.5. La barra de depuración web del entorno de desarrollo

La barra de depuración web se muestra en todas las páginas del entorno de desarrollo y al pinchar en cada pestaña se tiene acceso a mucha información: la configuración de la aplicación, los mensajes de log de la petición actual, las sentencias SQL ejecutadas en la base de datos, información sobre la memoria consumida y el tiempo total de ejecución de la petición.

Nota

La configuración es un poco diferente si quieres ejecutar Symfony en el servidor web IIS de Windows, por lo que deberías leer el tutorial sobre cómo configurar IIS para Symfony (http://www.symfony-project.com/cookbook/1_0/web_server_iis) .

1.12. Subversion

Una buena práctica cuando se desarrollan aplicaciones web consiste en emplear un sistema de control de versiones del código fuente. Este tipo de herramientas permiten:

- Trabajar con más confianza
- Volver a una versión anterior en caso de que un cambio rompa la aplicación
- Permitir a dos o más personas trabajar simultáneamente sobre un mismo proyecto de forma eficiente
- Disponer de acceso directo a todas las versiones de la aplicación

En esta sección se describe cómo utilizar Subversion (http://subversion.tigris.org/) con Symfony. Si utilizas cualquier otra herramienta para el versionado del código fuente, seguramente no será complicado adaptar las siguientes explicaciones para Subversion.

Para seguir el resto de la sección es imprescindible contar con un servidor de Subversion correctamente instalado y configurado y que pueda ser accedido mediante HTTP.

Sugerencia

Si no dispones de un servidor de Subversion, puedes solicitar uno gratuitamente en Google Code (http://code.google.com/hosting/) . También puedes buscar *"free subversion repository"* en Google para encontrar muchas otras opciones disponibles.

En primer lugar, crea un repositorio para el proyecto `jobeet` en tu servidor de repositorios:

```
$ svnadmin create /ruta/hasta/el/repositorio/jobeet
```

Después, crea la estructura básica de directorios en tu ordenador:

```
$ svn mkdir -m "Creación de la estructura de directorios inicial"
http://svn.ejemplo.com/jobeet/trunk http://svn.ejemplo.com/jobeet/tags
http://svn.ejemplo.com/jobeet/branches
```

A continuación, realiza el *checkout* del directorio `trunk/` vacío:

```
$ cd /home/sfprojects/jobeet
$ svn co http://svn.example.com/jobeet/trunk/ .
```

Después, borra el contenido de los directorios `cache/` y `log/` ya que no tiene sentido añadirlos al repositorio:

```
$ rm -rf cache/* log/*
```

Asegúrate de establecer los permisos adecuados en los directorios `cache/` y `log/` para que el servidor web pueda escribir en ellos:

```
$ chmod 777 cache log
```

Seguidamente, importa todos los archivos y directorios al repositorio:

```
$ svn add *
```

Como no vamos a importar los archivos de los directorios `cache/` y `log/`, debes añadirlos a la lista de archivos ignorados:

```
$ svn propedit svn:ignore cache
```

Después de ejecutar el comando anterior se abre el editor de archivos de texto configurado por defecto. Como queremos ignorar todos los contenidos de este directorio, escribe simplemente un asterisco:

```
*
```

Guarda el archivo y cierra el editor de textos para concluir el proceso.

Repite los pasos anteriores para el directorio `log/`:

```
$ svn propedit svn:ignore log
```

Vuelve a escribir un asterisco, guarda los cambios y cierra el editor:

```
*
```

Por último, sube estos cambios al repositorio:

```
$ svn import -m "Primera importación" . /ruta/hasta/el/repositorio/jobeet/trunk
```

Sugerencia

Si utilizas Windows, puedes emplear una aplicación genial llamada TortoiseSVN (http://tortoisesvn.tigris.org/) como herramienta para gestionar el repositorio de Subversion.

1.13. Nos vemos mañana

Se ha acabado el tiempo por hoy. Aunque todavía no hemos hablado de Symfony, hemos creado un buen entorno de desarrollo y hemos hablado de algunas de las mejores prácticas de desarrollo web, por lo que estamos listos para empezar a programar.

Mañana desvelaremos cuál es el propósito de la aplicación y hablaremos de los requisitos de la aplicación que vamos a desarrollar a lo largo del tutorial.

Nota

Si quieres acceder al código fuente de este o de cualquier otro tutorial, el código está disponible en el repositorio Subversion oficial de Jobeet (`http://svn.jobeet.org/propel`).

Para descargar el código del primer día, utiliza la etiqueta `release_day_01`:

```
$ svn co http://svn.jobeet.org/propel/tags/release_day_01/ jobeet/
```

Capítulo 2. El proyecto

Aunque todavía no hemos escrito ni una sola línea de código PHP, ayer configuramos el entorno de desarrollo, creamos un proyecto de Symfony vacío y nos aseguramos de empezar teniendo en cuenta algunas buenas prácticas relacionadas con la seguridad. De momento, lo único que puedes ver en la pantalla de tu navegador es la página de bienvenida por defecto de Symfony:

Figura 2.1. Página de bienvenida de Symfony

Ahora ha llegado el momento de introducirse en el maravilloso mundo de Symfony y aprender hasta el último detalle de este framework. Nuestro objetivo de hoy consiste en describir los requerimientos del proyecto mediante una serie de escenarios.

2.1. La idea del proyecto

Todo el mundo habla estos días de la crisis económica y de la subida del paro. Afortunadamente la mayoría de programadores de Symfony no se encuentran en esa situación y ese es uno de los principales motivos por los que te decidiste a aprender Symfony. Por otra parte, encontrar buenos programadores Symfony es bastante complicado.

¿Dónde puedes encontrar programadores Symfony? Y si eres programador ¿dónde puedes anunciar tus servicios o tus habilidades con el framework?

Para todo lo anterior necesitas un buen sitio web de búsqueda de empleo. ¿Estás pensando en Infojobs o Monster? Ni lo sueñes. Lo que necesitas es un sitio dedicado exclusivamente a los empleos relacionados con Symfony. Un sitio en el que puedas encontrar los mejores programadores, los auténticos expertos. Un sitio en el que sea fácil, rápido y divertido buscar un trabajo o publicar una oferta.

No hace falta que busques más porque Jobeet es lo que estabas esperando. Jobeet es una aplicación de software libre que permite crear sitios de búsqueda de empleo. Aunque Jobeet sólo hace una cosa, la hace muy bien. Jobeet es sencillo de utilizar, personalizar, extender e integrar con tu sitio web. Incluye de serie el soporte para varios idiomas e incorpora las últimas tecnologías Web 2.0 para mejorar la experiencia de usuario. También incluye canales RSS y una API que permite la interacción con otros servicios y aplicaciones.

¿Pero no existen muchos sitios web parecidos a Jobeet? Es cierto que como usuario ya has visto muchos sitios de búsqueda de empleo similares a Jobeet, pero te retamos a que encuentres una sola aplicación de este tipo que sea software libre y que tenga tantas características como las que vamos a incluir.

Como toda la aplicación la tenemos que construir en menos de 24 horas, más vale que empecemos cuanto antes a desarrollarla.

Nota

Si estás buscando de verdad un trabajo relacionado con Symfony o quieres contratar a algún programador que sepa Symfony, puedes visitar el sitio web symfonians (http://symfonians.net/)

2.2. Los escenarios del proyecto

Antes de meternos de lleno con el código, vamos a describir un poco más las características del proyecto. Las siguientes secciones utilizan diferentes escenarios y bocetos gráficos para describir todas las características que se quieren incluir en la primera versión o iteración del proyecto.

El sitio web de Jobeet dispone de cuatro tipos de usuarios:

- **administrador** (*admin*): es el dueño del sitio y tiene todo el poder
- **usuario** (*user*): visita el sitio web para ver ofertas de trabajo
- **publicador** (*poster*): visita el sitio web para publicar ofertas de trabajo
- **afiliado** (*affiliate*): publica en su propio sitio web algunas de las ofertas de trabajo

El proyecto se compone de dos aplicaciones: `frontend` (escenarios F1 a F7), donde los usuarios interactuan con el sitio web, y el `backend` (escenarios B1 a B3), donde los administradores gestionan el sitio web.

La aplicación `backend` dispone de acceso restringido y requiere ciertas credenciales para acceder.

2.2.1. Escenario F1: El usuario accede a la portada y ve las últimas ofertas de trabajo activas

Cuando el usuario accede a la portada de Jobeet, ve la lista de ofertas de trabajo activas. Las ofertas se agrupan por categoría y se ordenan por fecha de publicación (primero se muestran los trabajos más recientes). Para cada oferta se muestra la población, el puesto y la empresa.

Para cada categoría sólo se muestran las primeras diez ofertas y el resto se pueden visualizar pulsando sobre el enlace disponible (ver escenario F2).

En la portada el usuario también puede refinar el listado de ofertas (escenario F3) o publicar una nueva oferta (escenario F5).

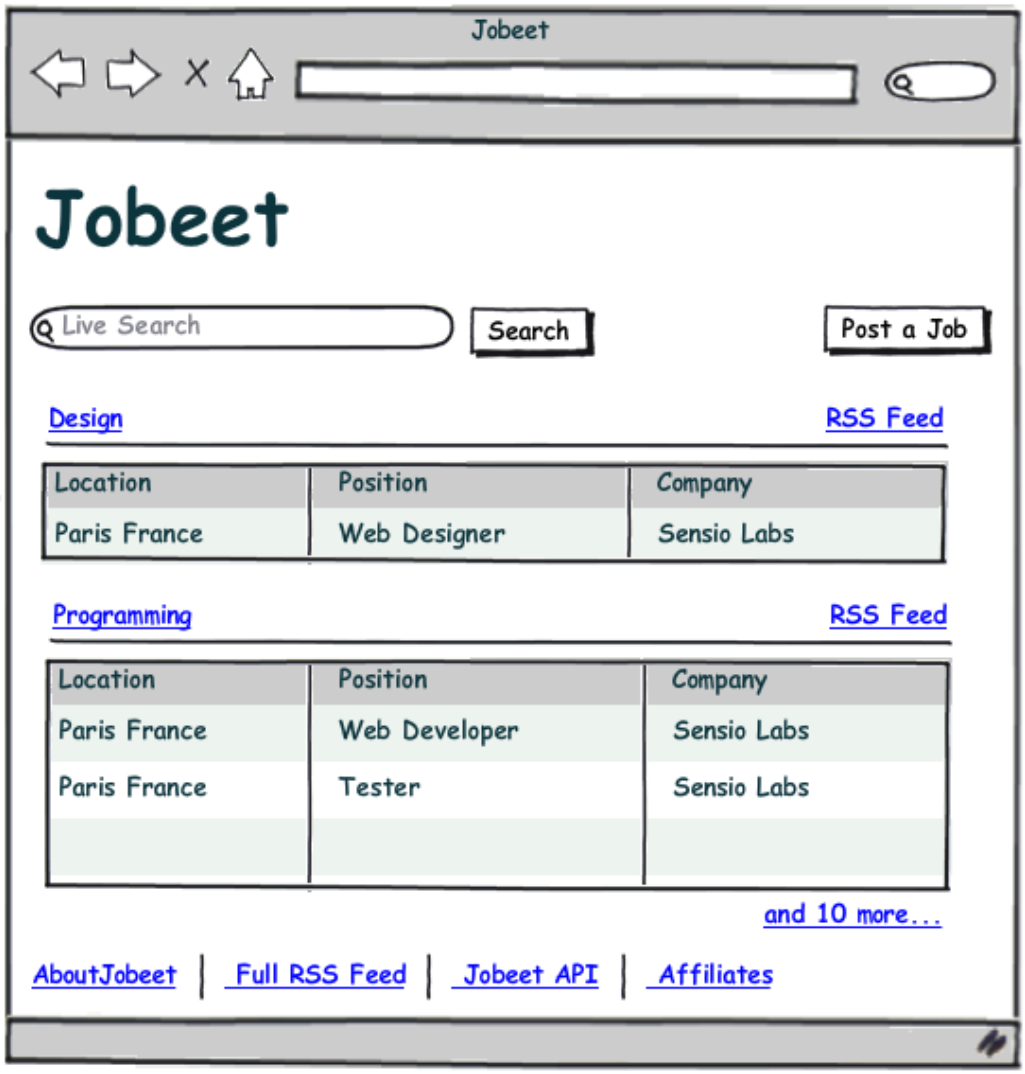

Figura 2.2. Boceto de la portada del sitio

2.2.2. Escenario F2: El usuario puede visualizar todas las ofertas de trabajo de una categoría

Cuando el usuario pulsa sobre el nombre de una categoría o sobre el enlace para ver más trabajos, se muestra el listado completo de todas las ofertas de trabajo ordenadas por fecha. Este listado incluye una paginación con 20 ofertas por página.

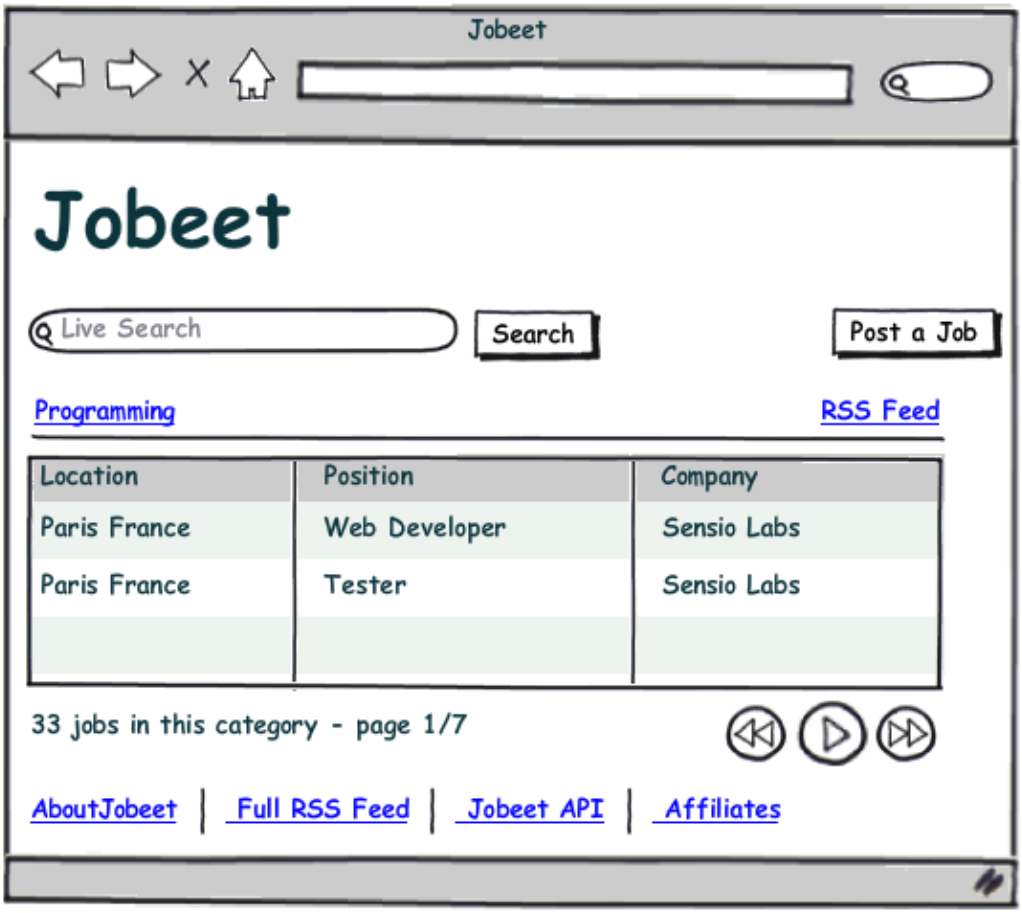

Figura 2.3. La página de la categoría

2.2.3. Escenario F3: El usuario refina el listado mediante palabras clave

El usuario puede utilizar palabras clave para refinar la búsqueda. Estas palabras clave se buscan en los campos de la población, el puesto, la categoría y la empresa.

2.2.4. Escenario F4: El usuario pincha sobre una oferta de trabajo para ver más información

El usuario puede pinchar sobre una oferta de trabajo del listado para ver toda su información.

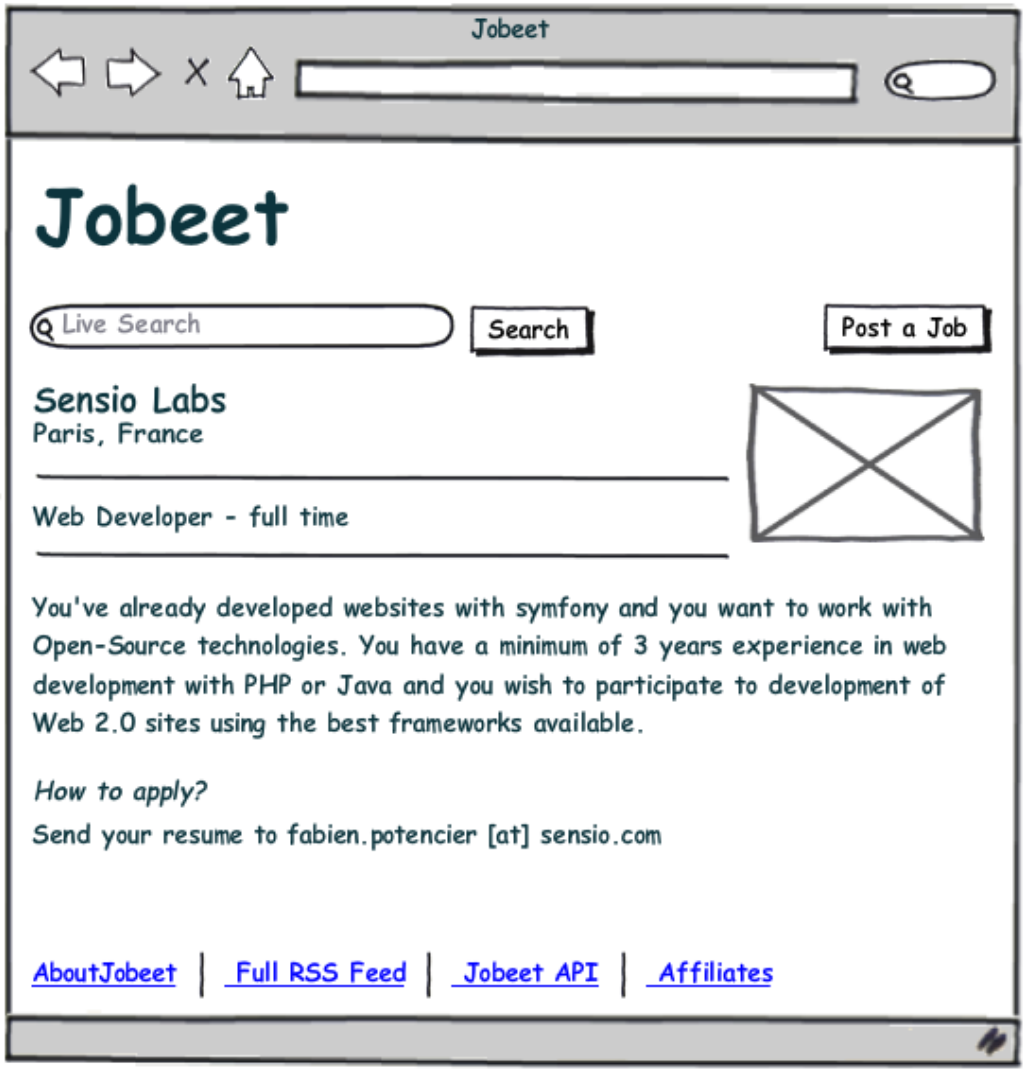

Figura 2.4. La página de detalle de una oferta de trabajo

2.2.5. Escenario F5: El usuario publica una nueva oferta de trabajo

Los usuarios también puede publicar ofertas de trabajo, que incluyen la siguiente información:

- Empresa (*company*)
- Tipo de trabajo (*type*) que puede ser: jornada completa (*full-time*), jornada parcial (*part-time*) o freelance.
- Logo, que es opcional
- URL, que es opcional
- Puesto (*position*)
- Población (*location*)
- Categoría (*category*): seleccionada entre una lista de posibles valores

- Descripción (*job description*): los emails y URL que contenga se convierten automáticamente en enlaces
- Cómo solicitar el trabajo (*how to apply*): los emails y URL que contenga se convierten automáticamente en enlaces
- Pública (*public*): si la oferta se puede publicar en otros sitios web afiliados
- Email: del usuario que publica la oferta

Para publicar una oferta de trabajo no es obligatorio registrarse en el sitio web. El proceso es muy sencillo porque sólo se compone de dos pasos: primero el usuario rellena el formulario con toda la información necesaria para describir la oferta de trabajo y a continuación, valida la información mediante la previsualización de la página de la oferta.

Aunque los usuarios no se registran, las ofertas de trabajo se pueden modificar posteriormente gracias a una URL específica protegida con un *token* que se proporciona al usuario al crear la oferta de trabajo.

Cada oferta tiene un período de validez de 30 días (configurable por el administrador, como se detalla en el escenario B2). Los usuarios pueden reactivar y extender la validez de la oferta por otros 30 días siempre y cuando falten menos de cinco días para que la oferta expire.

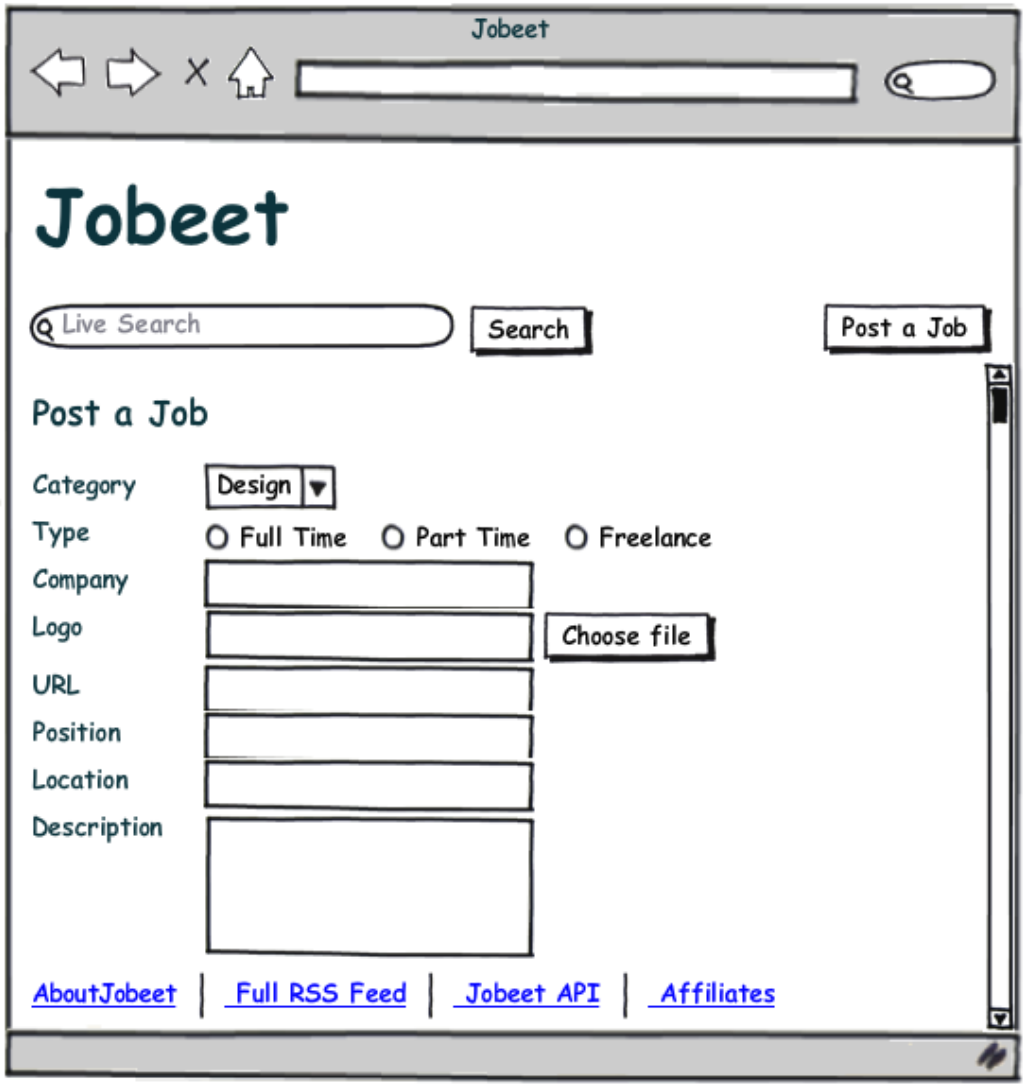

Figura 2.5. La página para insertar una nueva oferta de trabajo

2.2.6. Escenario F6: El usuario quiere convertirse en un afiliado

Los usuarios que quieren convertirse en afiliados deben solicitarlo y deben obtener una autorización para utilizar la API de Jobeet. Para realizar la solicitud es necesario proporcionar la siguiente información:

- Nombre (*name*)
- Email
- URL del sitio web (*website URL*)

Los administradores activan las cuentas de usuario de los afiliados (escenario B3). Una vez activada la cuenta, el afiliado recibe por email un *token* para utilizar la API.

Cuando realizan su solicitud, los afiliados pueden indicar que sólo quieren obtener las ofertas de trabajo relacionadas con una serie de categorías específicas.

2.2.7. Escenario F7: Un usuario afiliado obtiene la lista de ofertas de trabajo activas

Los afiliados pueden utilizar el *token* proporcionado para obtener la lista de ofertas de trabajo activas mediante la API del sitio web. El listado se puede devolver en los formatos XML, JSON o YAML.

El listado contiene la información pública disponible para cada oferta de trabajo. Los afiliados también pueden limitar el número de ofertas de trabajo del listado y pueden especificar una categoría para refinar la búsqueda.

2.2.8. Escenario B1: El administrador configura el sitio web

El administrador puede modificar las categorías disponibles en el sitio web.

2.2.9. Escenario B2: El administrador gestiona las ofertas de trabajo

El administrador puede modificar y borrar cualquier oferta de trabajo publicada.

2.2.10. Escenario B3: El administrador gestiona los afiliados

El administrador puede crear y modificar afiliados. Además de ser el responsable de activar a cada afiliado, también puede deshabilitar a cualquier afiliado activo.

Cuando el administrador activa a un nuevo afiliado, el sistema crea un *token* único para que lo utilice ese afiliado.

2.3. Nos vemos mañana

En la mayoría de proyectos web nunca se empieza a programar desde el primer día. En primer lugar es necesario conocer los requerimientos del sistema y realizar bocetos de cada característica importante. Este es precisamente el trabajo que hemos realizado hoy.

Capítulo 3. El modelo de datos

Para todos los que estáis ansiosos por abrir vuestro editor favorito y empezar a escribir código PHP hoy es vuestro día de suerte, ya que durante la lección de hoy vamos a empezar a programar. Hoy vamos a definir el modelo de datos de Jobeet, vamos a utilizar un ORM para interactuar con la base de datos y vamos a crear el primer módulo de la aplicación. Lo mejor es que como Symfony se encarga de la mayor parte del trabajo, vamos a crear un módulo web completamente funcional sin tener que escribir mucho código PHP.

3.1. El modelo relacional

Los escenarios que se presentaron en la lección de ayer describen los objetos principales que componen el proyecto: ofertas de trabajo (*jobs*), afiliados (*affiliates*) y categorías (*categories*). A continuación se muestra el correspondiente diagrama de entidad-relación:

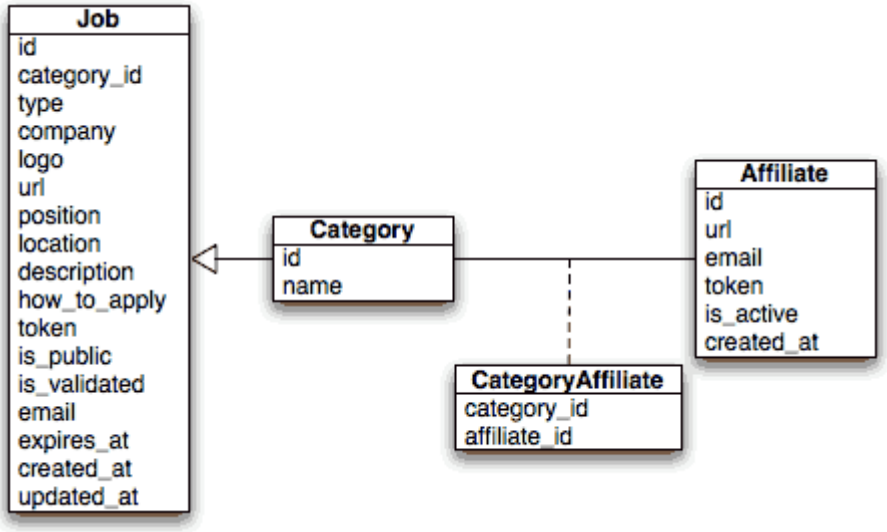

Figura 3.1. Diagrama de entidad-relación

Además de todas las columnas de información descritas en los escenarios, hemos añadido en algunas tablas un campo llamado `created_at`. Symfony trata de forma especial a todos los campos llamados `created_at`, ya que guarda en ellos automáticamente la fecha y hora del momento en el que se inserta el registro en la base de datos. Lo mismo sucede con los campos `updated_at`, cuyo valor se actualiza automáticamente cada vez que se actualiza un registro de la base de datos.

3.2. El esquema

Los datos de las ofertas de trabajo, afiliados y categorías se guardan en una base de datos relacional. Por otra parte, como Symfony es un framework orientado a objetos, nuestro objetivo es trabajar con objetos siempre que sea posible. Así por ejemplo,

preferimos utilizar objetos a tener que escribir sentencias SQL para obtener los registros de la base de datos.

Para trabajar con objetos en una base de datos relacional, es necesario realizar un *mapeo* o conversión entre la información de la base de datos y los objetos PHP. Este *mapeo* se realiza con unas herramientas llamadas ORM (http://es.wikipedia.org/wiki/Mapeo_objeto-relacional) y Symfony incluye por defecto dos de las más utilizadas: Propel (http://propel.phpdb.org/) y Doctrine (http://www.doctrine-project.org/) . En este tutorial vamos a utilizar Propel.

A partir de la descripción de cada tabla y de las relaciones entre tablas, el ORM crea las clases PHP necesarias para trabajar con objetos. Existen dos formas de crear la descripción del esquema de datos: mediante la introspección de una base de datos existente o creando el esquema manualmente.

Nota

Existen aplicaciones para crear bases de datos gráficamente (por ejemplo Dbdesigner de Fabforce (http://www.fabforce.net/dbdesigner4/)) y para generar archivos de tipo `schema.xml` (por ejemplo DB Designer 4 TO Propel Schema Converter (http://blog.tooleshed.com/docs/dbd2propel/transform.php)).

Como todavía no tenemos ninguna base de datos y como queremos que Jobeet funcione con todos los tipos de gestores de bases de datos, vamos a crear el archivo del esquema a mano. Para ello, abre el archivo `config/schema.yml` y añade lo siguiente tal y como está escrito:

```
# config/schema.yml
propel:
  jobeet_category:
    id:           ~
    name:         { type: varchar(255), required: true, index: unique }

  jobeet_job:
    id:           ~
    category_id:  { type: integer, foreignTable: jobeet_category,
foreignReference: id, required: true }
    type:         { type: varchar(255) }
    company:      { type: varchar(255), required: true }
    logo:         { type: varchar(255) }
    url:          { type: varchar(255) }
    position:     { type: varchar(255), required: true }
    location:     { type: varchar(255), required: true }
    description:  { type: longvarchar, required: true }
    how_to_apply: { type: longvarchar, required: true }
    token:        { type: varchar(255), required: true, index: unique }
    is_public:    { type: boolean, required: true, default: 1 }
    is_activated: { type: boolean, required: true, default: 0 }
    email:        { type: varchar(255), required: true }
    expires_at:   { type: timestamp, required: true }
    created_at:   ~
    updated_at:   ~
```

```
  jobeet_affiliate:
    id:           ~
    url:          { type: varchar(255), required: true }
    email:        { type: varchar(255), required: true, index: unique }
    token:        { type: varchar(255), required: true }
    is_active:    { type: boolean, required: true, default: 0 }
    created_at:   ~

  jobeet_category_affiliate:
    category_id:  { type: integer, foreignTable: jobeet_category,
foreignReference: id, required: true, primaryKey: true, onDelete: cascade }
    affiliate_id: { type: integer, foreignTable: jobeet_affiliate,
foreignReference: id, required: true, primaryKey: true, onDelete: cascade }
```

Sugerencia

Si eres de los que prefieres crear la base de datos directamente con sentencias SQL, puedes generar el archivo de configuración `schema.yml` a partir de una base de datos existente mediante la tarea `propel:build-schema`

```
$ php symfony propel:build-schema
```

El esquema de datos no es más que la traducción del diagrama de entidad-relación al formato YAML.

El formato YAML

Según la definición del sitio web oficial de YAML (http://www.yaml.org/) , *"YAML es un formato para serializar datos que es fácil de leer por las personas y es compatible con todos los lenguajes de programación"*.

Dicho de otra forma, YAML es un lenguaje muy sencillo que permite describir datos: cadenas de texto, número enteros, fechas, arrays *simples* y arrays asociativos.

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 arrays asociativos se separan con dos puntos. YAML también dispone de una notación abreviada para describir la misma estructura con menos líneas: los arrays simples se definen con `[]` y los arrays asociativos se definen con `{}`.

Si todavía no conoces YAML, deberías aprender sus características básicas antes de continuar, ya que Symfony utiliza YAML en la mayoría de sus archivos de configuración.

Lo más importante que debes tener en cuenta al modificar un archivo YAML es que la tabulación siempre se realiza con espacios en blanco y nunca con el tabulador.

El archivo `schema.yml` describe todas las tablas y columnas de la base de datos. Cada columna se describe con la siguiente información:

- `type`: el tipo de columna, que puede ser `boolean`, `tinyint`, `smallint`, `integer`, `bigint`, `double`, `float`, `real`, `decimal`, `char`, `varchar(size)`, `longvarchar`, `date`, `time`, `timestamp`, `blob` y `clob`.
- `required`: si vale `true`, la columna es obligatoria.

- `index`: si vale `true`, se crea un índice para la columna; si vale `unique`, se crea un índice único.
- `primaryKey`: indica que esta columna es clave primaria de la tabla.
- `foreignTable`, `foreignReference`: indica que esta columna es clave externa de otra tabla.

En las columnas cuyo valor es simplemente `~`, que en realidad es como se indica el valor `null` en YAML (`id`, `created_at` y `updated_at`), Symfony adivina cuál es la mejor configuración para esa columna (los campos llamados `id` se consideran claves primarias y los campos llamados `created_at` y `updated_at` se consideran de tipo `timestamp`).

Nota

El atributo `onDelete` define el comportamiento de las claves primarias ante las sentencias `ON DELETE`. Propel admite los valores `CASCADE`, `SETNULL` y `RESTRICT`. Cuando se borra por ejemplo el registro de una oferta de trabajo (*job*) todos los registros relacionados de la tabla `jobeet_category_affiliate` se borran automáticamente mediante la base de datos o mediante Propel si el sistema gestor de base de datos no es capaz de hacerlo.

3.3. La base de datos

El framework Symfony es compatible con todas las bases de datos soportadas por PDO (http://www.php.net/PDO) , la capa de abstracción de bases de datos incluida en PHP: MySQL, PostgreSQL, SQLite, Oracle, MSSQL, etc.

En este tutorial se utiliza MySQL, por lo que puedes ejecutar el siguiente comando para crear la base de datos:

```
$ mysqladmin -uroot -pConTraSenA create jobeet
```

Nota

Si quieres, puedes utilizar cualquier otro gestor de bases de datos que no sea MySQL. Como vamos a trabajar con un ORM que se encarga de generar automáticamente las sentencias SQL, es muy sencillo adaptar el código a otro tipo de base de datos.

A continuación se le indica a Symfony que vamos a utilizar esta base de datos para el proyecto Jobeet:

```
$ php symfony configure:database "mysql:host=localhost;dbname=jobeet" root
ConTraSenA
```

La tarea `configure:database` admite hasta tres argumentos: el DSN de PDO (http://www.php.net/manual/es/pdo.drivers.php) , el nombre de usuario y la contraseña para acceder a la base de datos. Si en el servidor de desarrollo no utilizas ninguna contraseña para acceder a la base de datos, puedes omitir el tercer argumento.

Nota

La tarea `configure:database` guarda la configuración de la base de datos en el archivo `config/databases.yml`. Si prefieres editar los archivos de configuración a mano, puedes hacerlo y no utilizar esta tarea.

3.4. El ORM

Gracias a la descripción de las tablas y columnas de la base de datos en el archivo `schema.yml`, podemos hacer uso de algunas tareas incluidas en Propel para generar automáticamente las sentencias SQL necesarias para crear todas las tablas de la base de datos:

```
$ php symfony propel:build-sql
```

La tarea `propel:build-sql` genera en el directorio `data/sql/` las sentencias SQL optimizadas para el sistema gestor de bases de datos que estamos utilizando:

```
# fragmento del archivo data/sql/lib.model.schema.sql
CREATE TABLE ''jobeet_category''
(
    `id` INTEGER  NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255)  NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `jobeet_category_U_1` (`name`)
)Type=InnoDB;
```

Para crear la estructura de tablas en la base de datos, ejecuta la tarea `propel:insert-sql`:

```
$ php symfony propel:insert-sql
```

Como la tarea anterior borra todas las tablas existentes antes de volver a crearlas, se muestra un mensaje de confirmación que debes aceptar. Si añades la opción `--no-confirmation` cuando ejecutas la tarea, no se muestra ningún mensaje de confirmación, lo que es útil cuando se incluye esta tarea en un script automático:

```
$ php symfony propel:insert-sql --no-confirmation
```

Sugerencia

Como sucede con cualquier otra herramienta para la línea de comandos, las tareas de Symfony admiten argumentos y opciones. Cada tarea incluye una explicación completa de su uso que se puede mostrar mediante la tarea `help`:

```
$ php symfony help propel:insert-sql
```

Las explicaciones muestran todos los argumentos y opciones de la tarea, los valores iniciales de cada uno de ellos y también algunos ejemplos de uso.

El ORM también se encarga de generar automáticamente las clases PHP que relacionan las tablas de la base de datos con los objetos de la aplicación:

```
$ php symfony propel:build-model
```

La tarea `propel:build-model` genera en el directorio `lib/model/` todos los archivos PHP que se utilizan para interactuar con la base de datos. Si echas un vistazo a los archivos generados automáticamente, verás que Propel crea cuatro clases por cada tabla de la base de datos. Si por ejemplo se considera la tabla `jobeet_job`:

- `JobeetJob`: los objetos de esta clase representan un registro de la tabla `jobeet_job`. Inicialmente esta clase está completamente vacía.
- `BaseJobeetJob`: la clase de la que hereda `JobeetJob`. Al contrario que la clase anterior, cada vez que ejecutas la tarea `propel:build-model`, esta clase se borra y se vuelve a generar. Por tanto, si quieres personalizar las clases del modelo, lo debes hacer en la clase `JobeetJob`.
- `JobeetJobPeer`: se trata de una clase que define los métodos estáticos utilizados para obtener colecciones de objetos de tipo `JobeetJob`. Inicialmente esta clase está completamente vacía.
- `BaseJobeetJobPeer`: la clase de la que hereda `JobeetJobPeer`. Como sucede con la clase `BaseJobeetJob`, cada vez que ejecutas la tarea `propel:build-model`, esta clase se borra y se vuelve a generar. Por tanto, si quieres personalizar las clases del modelo, lo debes hacer en la clase `JobeetJobPeer`.

Una vez creadas las clases PHP, los valores almacenados en las columnas de un registro de la base de datos se pueden obtener y/o manipular gracias a los métodos `get*()` y `set*()` disponibles:

```
$job = new JobeetJob();
$job->setPosition('Web developer');
$job->save();

echo $job->getPosition();

$job->delete();
```

También es posible definir claves externas relacionando objetos entre sí:

```
$category = new JobeetCategory();
$category->setName('Programming');

$job = new JobeetJob();
$job->setCategory($category);
```

Por último, existe una tarea llamada `propel:build-all` que es un atajo de todas las tareas que hemos utilizado hasta este momento y algunas más. Así que ejecuta esta tarea para que genere de forma consecutiva las sentencias SQL, la base de datos, las clases del modelo, los formularios y los validadores:

```
$ php symfony propel:build-all --no-confirmation
```

Los validadores se muestran al final de esta lección y los formularios se explican detalladamente en la lección del día 10.

Como explicaremos más adelante, Symfony dispone de un mecanismo que carga automáticamente las clases PHP, lo que significa que nunca tendrás que utilizar una sentencia `require()` en tu código. La carga automática de clases es otra de las ayudas que Symfony proporciona a los programadores, aunque tiene una pega: cada vez que añades una clase nueva al proyecto es necesario borrar la cache que utiliza Symfony. Como la tarea `propel:build-model` acaba de crear muchas clases nuevas, no olvides borrar la cache mediante el comando:

```
$ php symfony cache:clear
```

Sugerencia

El nombre de las tareas de Symfony se compone de una primera parte llamada *namespace* y de una segunda parte que es el propio nombre de la tarea. Cada una de las partes se puede abreviar tanto como se quiera siempre que no se produzca una ambigüedad con el nombre del resto de tareas. Por tanto, los siguientes comandos son equivalentes a `cache:clear`:

```
$ php symfony cache:cl
$ php symfony ca:c
```

Además, como la tarea `cache:clear` es la más utilizada de Symfony con mucha diferencia, dispone de un atajo todavía más corto:

```
$ php symfony cc
```

3.5. Los datos iniciales

Aunque ya hemos creado la base de datos, todas sus tablas están vacías. En cualquier aplicación web siempre existen tres tipos de datos:

- **Datos iniciales**: son los datos que necesita la aplicación para funcionar. Jobeet por ejemplo necesita el nombre de algunas categorías y también es necesario al menos un usuario de tipo `admin` para poder acceder a la aplicación `backend`.
- **Datos de prueba**: son los datos necesarios para probar la aplicación. Los buenos programadores crean pruebas unitarias para asegurar que la aplicación se comporta tal y como se describe en los escenarios. La mejor forma de probar la aplicación consiste en realizar pruebas unitarias automáticas. Cada vez que se ejecutan las pruebas unitarias es necesario disponer de datos de prueba en la base de datos.
- **Datos de usuarios**: son los datos reales creados por los usuarios que utilizan la aplicación.

Cada vez que Symfony genera las tablas de la base de datos, se elimina toda la información existente. Para insertar de nuevo los datos iniciales podríamos utilizar un script de PHP o podríamos ejecutar directamente unas sentencias SQL con el comando `mysql`. No obstante, como se trata de una necesidad bastante habitual, Symfony ofrece una alternativa mucho mejor: crear archivos en formato YAML, guardarlos en el directorio `data/fixtures/` y utilizar la tarea `propel:data-load` para cargarlos automáticamente en la base de datos:

En primer lugar, crea los siguientes archivos de datos en formato YAML:

```
# data/fixtures/010_categories.yml
JobeetCategory:
  design:        { name: Design }
  programming:   { name: Programming }
  manager:       { name: Manager }
  administrator: { name: Administrator }
# data/fixtures/020_jobs.yml
JobeetJob:
  job_sensio_labs:
    category_id:  programming
    type:         full-time
    company:      Sensio Labs
    logo:         sensio-labs.gif
    url:          http://www.sensiolabs.com/
    position:     Web Developer
    location:     Paris, France
    description:  |
      You have already developed websites with symfony and you want
      to work with Open-Source technologies. You have a minimum of
      3 years experience in web development with PHP or Java and
      you wish to participate to development of Web 2.0 sites using
      the best frameworks available.
    how_to_apply: |
      Send your resume to fabien.potencier [at] sensio.com
    is_public:    true
    is_activated: true
    token:        job_sensio_labs
    email:        job@example.com
    expires_at:   2010-10-10

  job_extreme_sensio:
    category_id:  design
    type:         part-time
    company:      Extreme Sensio
    logo:         extreme-sensio.gif
    url:          http://www.extreme-sensio.com/
    position:     Web Designer
    location:     Paris, France
    description:  |
      Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
      enim ad minim veniam, quis nostrud exercitation ullamco laboris
      nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
      in reprehenderit in.

      Voluptate velit esse cillum dolore eu fugiat nulla pariatur.
      Excepteur sint occaecat cupidatat non proident, sunt in culpa
      qui officia deserunt mollit anim id est laborum.
    how_to_apply: |
      Send your resume to fabien.potencier [at] sensio.com
    is_public:    true
    is_activated: true
    token:        job_extreme_sensio
```

```
    email:        job@example.com
    expires_at:   2010-10-10
```

Nota

El archivo de datos de las ofertas de trabajo hace referencia a dos imágenes. Puedes descargarlas desde el sitio web de Symfony y colocarlas en el directorio `uploads/jobs/`: http://www.symfony-project.org/get/jobeet/sensio-labs.gif y http://www.symfony-project.org/get/jobeet/extreme-sensio.gif

Un archivo de datos (*fixtures file*) es un archivo escrito en formato YAML que define los objetos del modelo y los etiqueta con un nombre único (en el ejemplo anterior hemos creado dos ofertas de trabajo etiquetadas `job_sensio_labs` y `job_extreme_sensio`). Este nombre es imprescindible para relacionar objetos entre sí sin tener que definir claves primarias (que normalmente son valores que se auto-incrementan y por tanto, no se pueden establecer). En los archivos anteriores, la categoría de la oferta de trabajo `job_sensio_labs` es `programming`, que es el nombre único que le hemos dado a la categoría `Programming`.

Sugerencia

En los archivos YAML, cuando una cadena de texto contiene saltos de línea (como por ejemplo la columna `description` del archivo de datos de las ofertas de trabajo) puedes utilizar el símbolo | para indicar que la cadena de texto ocupa varias líneas.

Aunque los archivos de datos pueden contener objetos de uno o varios modelos diferentes, en los archivos de datos de Jobeet hemos decidido crear un archivo para cada modelo.

Sugerencia

Si te has fijado atentamente, habrás visto que los nombres de los archivos de datos incluyen un prefijo numérico. Aunque puedes utilizar los nombres que quieras, prefijar cada archivo con un número es una de las formas más sencillas de controlar el orden en el que se cargan los archivos de datos. Además, es una buena idea no utilizar números consecutivos por si más adelante tenemos que crear nuevos archivos de datos que se tienen que cargar entre medio de dos archivos ya existentes.

En los archivos de datos no es obligatorio establecer el valor de todas las columnas. Si no se indica el valor de una columna, Symfony le asigna el valor por defecto establecido en el esquema de la base de datos. Además, como Symfony utiliza Propel para cargar los datos, funcionan todas las características avanzadas (como establecer automáticamente el valor de las columnas `created_at` y `updated_at`) y todos los comportamientos que hayas definido en las clases del modelo.

Una vez creados los archivos de datos, cargarlos en la base de datos es tan sencillo como ejecutar la tarea `propel:data-load`:

```
$ php symfony propel:data-load
```

Sugerencia

La tarea `propel:build-all-load` es equivalente a ejecutar la tarea `propel:build-all` seguida de la tarea `propel:data-load`

3.6. Probando la aplicación en el navegador

Hasta el momento hemos utilizado mucho la línea de comandos, pero eso no es nada emocionante, sobre todo para un proyecto web. No obstante, gracias a la línea de comandos ya tenemos todo lo que necesitamos para crear páginas web que interactúen con la base de datos.

A continuación se va a crear un listado de las ofertas de trabajo, se va a modificar una oferta existente y se va a borrar otra oferta de trabajo. Como se explicó en la lección del primer día, los proyectos Symfony se componen de aplicaciones. A su vez, cada aplicación está dividida en **módulos**. Un módulo es un conjunto autosuficiente de código PHP que representa una característica de la aplicación (como por ejemplo, el módulo de la API) o un conjunto de operaciones que el usuario puede realizar sobre un objeto del modelo (como por ejemplo el módulo de las ofertas de trabajo).

Symfony es capaz de generar automáticamente un módulo que permite realizar las operaciones básicas sobre los datos de un objeto del modelo:

```
$ php symfony propel:generate-module --with-show --non-verbose-templates
frontend job JobeetJob
```

La tarea `propel:generate-module` anterior genera un módulo llamado `job` en la aplicación `frontend` y basado en el modelo `JobeetJob`. Después de ejecutar la tarea `propel:generate-module`, se han creado varios archivos y directorios dentro del directorio `apps/frontend/modules/job/`:

Directorio	Descripción
`actions/`	Las acciones del módulo
`templates/`	Las plantillas del módulo

El archivo `actions/actions.class.php` define todas las acciones disponibles en el módulo `job`:

Nombre de la acción	Descripción
`index`	Muestra un listado con los registros de la base de datos
`show`	Muestra los campos y valores de un registro específico
`new`	Muestra un formulario para insertar un nuevo registro en la base de datos
`create`	Inserta un nuevo registro en la base de datos
`edit`	Muestra un formulario para modificar un registro existente en la base de datos

`update`	Actualiza los datos de un registro a partir de la información enviada por el usuario
`delete`	Elimina un registro de la base de datos

Ahora ya puedes probar el módulo `job` accediendo a la siguiente URL en tu navegador:

```
http://jobeet.localhost/frontend_dev.php/job
```

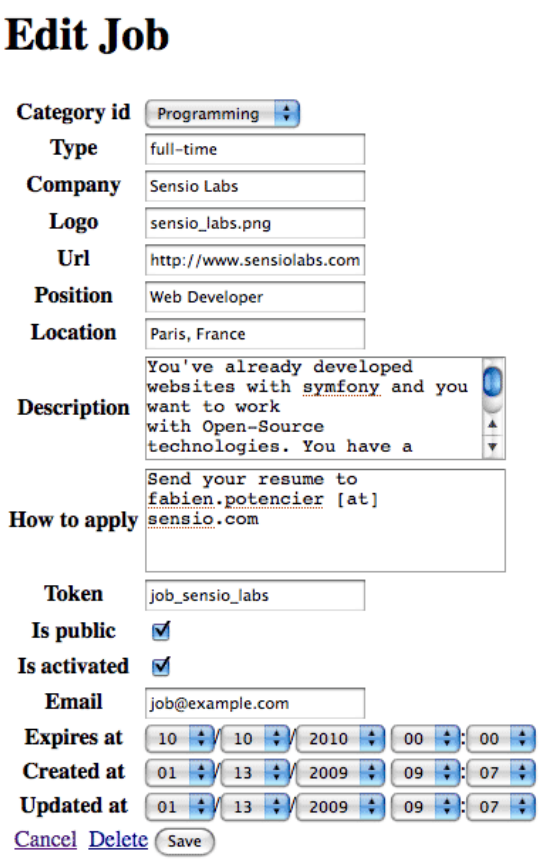

Figura 3.2. Módulo job

Si intentas modificar los datos de una oferta de trabajo, verás que Symfony muestra una excepción, ya que no se ha indicado cuál es la representación en forma de texto de los objetos de tipo categoría. La representación textual de un objeto PHP se establece con el método mágico `__toString()`. Añade el siguiente código en la clase `JobeetCategory` del modelo para establecer su representación textual:

```
// lib/model/JobeetCategory.php
class JobeetCategory extends BaseJobeetCategory
{
  public function __toString()
  {
    return $this->getName();
  }
}
```

Ahora, cuando Symfony necesite mostrar la representación en forma de texto de una categoría, se invoca el método `__toString()`, que devuelve directamente el nombre de la categoría. Como seguramente vamos a necesitar la representación textual de todas las

clases del modelo, es una buena idea definir ahora el método `__toString()` en el resto de las clases del modelo:

```
// lib/model/JobeetJob.php
class JobeetJob extends BaseJobeetJob
{
  public function __toString()
  {
    return sprintf('%s at %s (%s)', $this->getPosition(), $this->getCompany(),
$this->getLocation());
  }
}
// lib/model/JobeetAffiliate.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function __toString()
  {
    return $this->getUrl();
  }
}
```

Ahora ya puedes modificar cualquier dato de las ofertas de trabajo. Prueba a dejar un campo en blanco o intenta introducir una fecha incorrecta. En efecto, Symfony ha generado automáticamente unas reglas de validación básicas a partir de la información del esquema de datos.

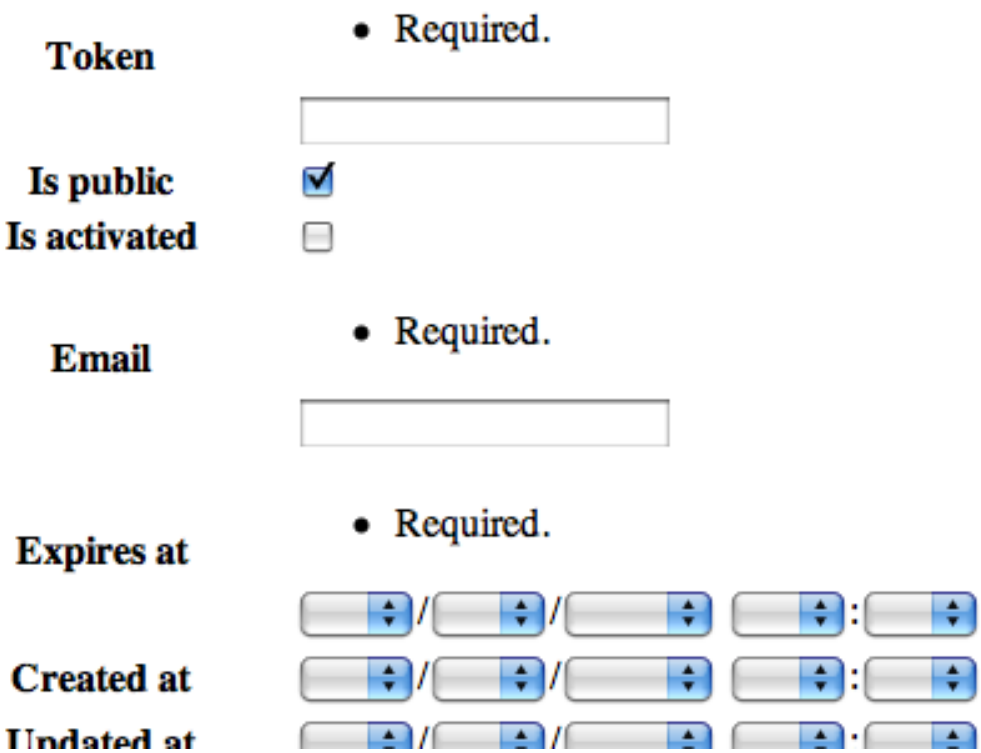

Figura 3.3. Validación de datos

3.7. Nos vemos mañana

Y esto es todo por hoy. Tal y como te advertimos en la introducción de esta lección, hoy apenas hemos escrito un poco de código PHP, pero ya disponemos de un módulo web completo para el modelo de datos de las ofertas de trabajo. Ahora ya sólo nos falta personalizar y ajustar el módulo generado automáticamente. Además, recuerda que cuanto menos código PHP escribas, menos probabilidades tienes de introducir errores en la aplicación.

Si todavía te ves con ganas, puedes investigar el código que Symfony ha generado automáticamente para el módulo y para el modelo y tratar de entender cómo funciona. Si no lo haces, no te preocupes porque en la lección de mañana vamos a explicar uno de los paradigmas más utilizados en los frameworks para aplicaciones web, el patrón de diseño MVC (http://es.wikipedia.org/wiki/Modelo_Vista_Controlador) .

Como es habitual, el código de esta lección se ha publicado en el repositorio de Subversion de Jobeet y ha sido etiquetado como `release_day_03`. Para obtener su código sólo tienes que ejecutar el siguiente comando:

```
$ svn co http://svn.jobeet.org/propel/tags/release_day_03/ jobeet/
```

Capítulo 4. El controlador y la vista

Ayer vimos cómo Symfony simplifica el trabajo con las bases de datos mediante una capa de abstracción que elimina las diferencias entre bases de datos y mediante la traducción de la información relacional de la base de datos en clases orientadas a objetos. También trabajamos con Propel para describir el esquema de la base de datos, crear las tablas y llenarlas con algunos datos iniciales.

En la lección de hoy vamos a personalizar el módulo `job` que creamos ayer. Este módulo `job` básico ya dispone de todo el código necesario para Jobeet:

- Una página para listar todas las ofertas de trabajo
- Una página para crear una nueva oferta
- Una página para actualizar una oferta de trabajo existente
- Una página para borrar una oferta de trabajo

Aunque el código ya está listo para ser utilizado, vamos a refactorizar las plantillas para que se parezcan más a los bocetos gráficos que diseñamos para Jobeet.

4.1. La arquitectura MVC

Si has desarrollado sitios web con PHP sin utilizar ningún framework, seguramente sigues el razonamiento de crear un archivo PHP por cada página HTML del sitio. Además, todos esos archivos PHP contienen seguramente la misma estructura: inicialización y configuración global, lógica de negocio relacionada con la página solicitada, obtención de registros de la base de datos y por último, el código PHP que se emplea para generar la página.

También es posible que utilices un sistema de plantillas para separar el código PHP y las etiquetas HTML. Puede que también utilices una capa de abstracción de base de datos para separar la lógica de negocio y la interacción con el modelo de datos. A pesar de estas mejoras, la mayoría de las veces te encuentras con una gran cantidad de código que es muy difícil de mantener. Programar la aplicación de esa manera quizás te costó muy poco tiempo, pero modificarla y añadirle nuevas características se convierte en una pesadilla, sobre todo porque nadie más que tu sabe cómo está construida y cómo funciona.

Para cada problema siempre hay buenas soluciones y para la programación web, la solución más utilizada actualmente para organizar el código es el patrón de diseño MVC (http://es.wikipedia.org/wiki/Modelo_Vista_Controlador) . En pocas palabras, el patrón de diseño MVC organiza el código en base a su función. De hecho, este patrón separa el código en tres capas:

- La capa del **modelo** define la lógica de negocio (la base de datos pertenece a esta capa). Como ya sabes, Symfony guarda todas las clases y archivos relacionados con el modelo en el directorio `lib/model/`.
- La **vista** es lo que utilizan los usuarios para interactuar con la aplicación (los gestores de plantillas pertenecen a esta capa). En Symfony la capa de la vista está formada principalmente por plantillas en PHP. Estas plantillas se guardan en varios directorios llamados `templates/` repartidos por todo el proyecto, tal y como veremos hoy mismo.
- El **controlador** es un bloque de código que realiza llamadas al modelo para obtener los datos y se los pasa a la vista para que los muestre al usuario. Cuando instalamos Symfony el primer día, explicamos que todas las peticiones se canalizan a través de los controladores frontales (`index.php` y `frontend_dev.php`). Estos controladores frontales realmente delegan todo el trabajo en las **acciones**. Como vimos ayer, las agrupaciones lógicas de acciones se denominan **módulos**.

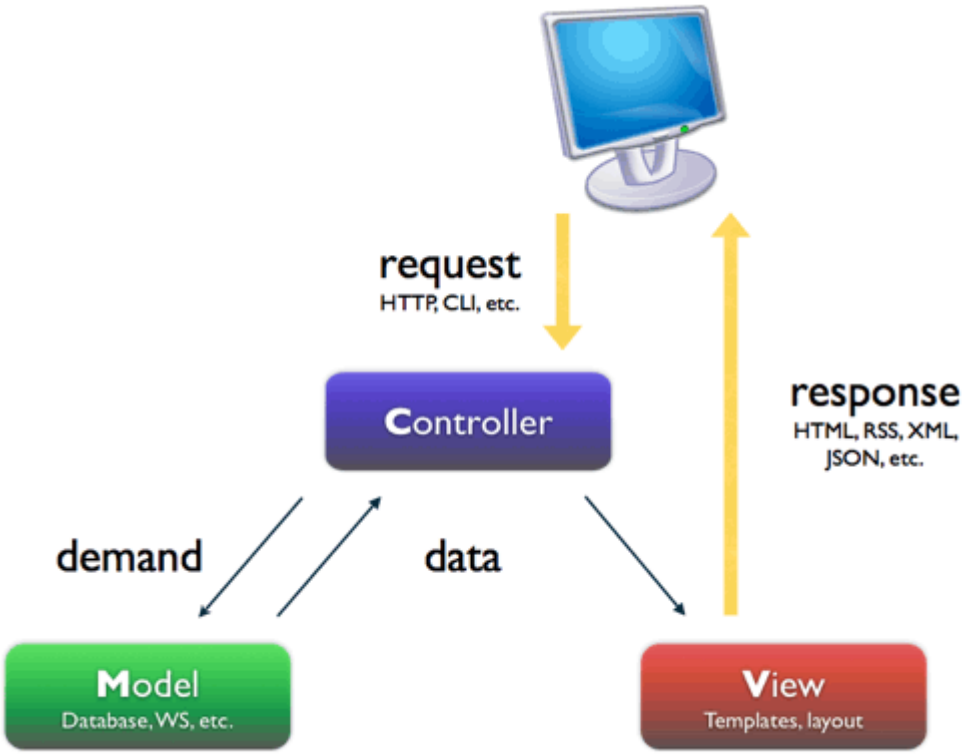

Figura 4.1. Arquitectura MVC

Hoy vamos a utilizar los bocetos gráficos que definimos el segundo día para personalizar y hacer más dinámicas la portada y las páginas que muestran cada oferta de trabajo. Al mismo tiempo, vamos a modificar muchas cosas en muchos archivos diferentes para explicar la estructura de directorios de Symfony y su forma de separar el código en capas.

4.2. El layout

Si te fijas atentamente en los bocetos gráficos, verás que algunas partes se repiten en todas las páginas. Como ya sabes, duplicar el código nunca es buena idea, ya sea código PHP o etiquetas HTML. Por tanto, tenemos que encontrar alguna forma de evitar la repetición de estos elementos comunes de las páginas.

Una forma sencilla de resolver este problema consiste en definir una cabecera y un pie que se añaden en cada plantilla:

Figura 4.2. Cabecera y pie de página

El problema es que los archivos de la cabecera y del pie no contienen código HTML válido, por lo que debemos buscar una alternativa. En vez de perder el tiempo tratando de reinventar la rueda, vamos a utilizar otro patrón de diseño para resolver este problema: el patrón de diseño decorator (http://es.wikipedia.org/wiki/Decorator_(patrón_de_diseño)) .

El patrón *decorator* resuelve el problema de otra forma diferente: el contenido se muestra con una plantilla que después se decora con una plantilla global que en Symfony se llama **layout**:

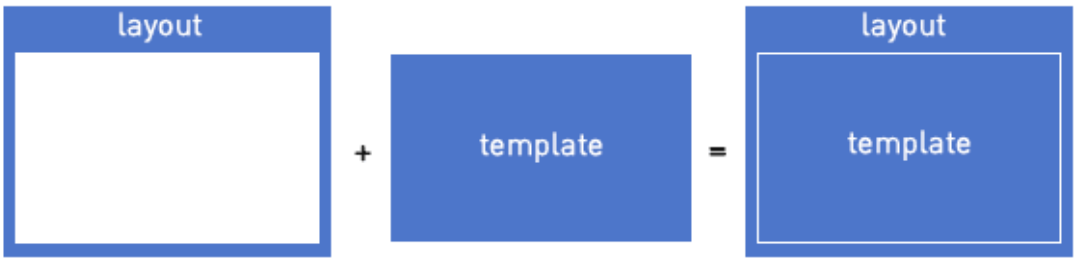

Figura 4.3. Funcionamiento del layout

El layout por defecto de todas las aplicaciones es un archivo llamado `layout.php` que se encuentra en el directorio `apps/frontend/templates/`. En este directorio se guardan todas las plantillas globales de una aplicación.

Para crear un layout apropiado para la aplicación Jobeet, reemplaza el contenido del layout por defecto de Symfony por este otro código:

```
<!-- 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>
```

```
  <title>Jobeet - Your best job board</title>
  <link rel="shortcut icon" href="/favicon.ico" />
  <?php include_javascripts() ?>
  <?php include_stylesheets() ?>
</head>
<body>
  <div id="container">
    <div id="header">
      <div class="content">
        <h1><a href="/job">
          <img src="/images/logo.jpg" alt="Jobeet Job Board" />
        </a></h1>

        <div id="sub_header">
          <div class="post">
            <h2>Ask for people</h2>
            <div>
              <a href="/job/new">Post a Job</a>
            </div>
          </div>

          <div class="search">
            <h2>Ask for a job</h2>
            <form action="" method="get">
              <input type="text" name="keywords" id="search_keywords" />
              <input type="submit" value="search" />
              <div class="help">
                Enter some keywords (city, country, position, ...)
              </div>
            </form>
          </div>
        </div>
      </div>
    </div>

    <div id="content">
      <?php if ($sf_user->hasFlash('notice')): ?>
        <div class="flash_notice">
          <?php echo $sf_user->getFlash('notice') ?>
        </div>
      <?php endif; ?>

      <?php if ($sf_user->hasFlash('error')): ?>
        <div class="flash_error">
          <?php echo $sf_user->getFlash('error') ?>
        </div>
      <?php endif; ?>

      <div class="content">
        <?php echo $sf_content ?>
      </div>
    </div>

    <div id="footer">
      <div class="content">
```

```
          <span class="symfony">
            <img src="/images/jobeet-mini.png" />
            powered by <a href="http://www.symfony-project.org/">
            <img src="/images/symfony.gif" alt="symfony framework" />
            </a>
          </span>
          <ul>
            <li><a href="">About Jobeet</a></li>
            <li class="feed"><a href="">Full feed</a></li>
            <li><a href="">Jobeet API</a></li>
            <li class="last"><a href="">Affiliates</a></li>
          </ul>
        </div>
      </div>
    </div>
  </body>
</html>
```

Las plantillas de Symfony se crean con archivos PHP normales. Por eso en el layout anterior existen llamadas a funciones PHP y referencias a variables PHP. De todas las variables, la más interesante se llama `$sf_content`, ya que la crea el propio framework y contiene el código HTML generado por la acción.

Si vuelves a acceder al módulo `job` desde un navegador (`http://jobeet.localhost/frontend_dev.php/job`), verás que ahora todas las acciones están decoradas por un layout.

4.3. Las hojas de estilo, imágenes y archivos JavaScript

Como este tutorial no trata sobre el diseño web, hemos preparado todos los archivos que utilizan las páginas de Jobeet: descarga el archivo ZIP con todas las imágenes (http://www.symfony-project.org/get/jobeet/images.zip) y descomprímelo en el directorio `web/images/`, descarga el archivo ZIP con todas las hojas de estilos CSS (http://www.symfony-project.org/get/jobeet/css.zip) y descomprímelo en el directorio `web/css/`.

Nota

En el layout también hemos incluido un favicon. Si quieres, puedes descargar el favicon de Jobeet (http://www.symfony-project.org/images/jobeet/favicon.ico) y guardarlo en el directorio `web/`.

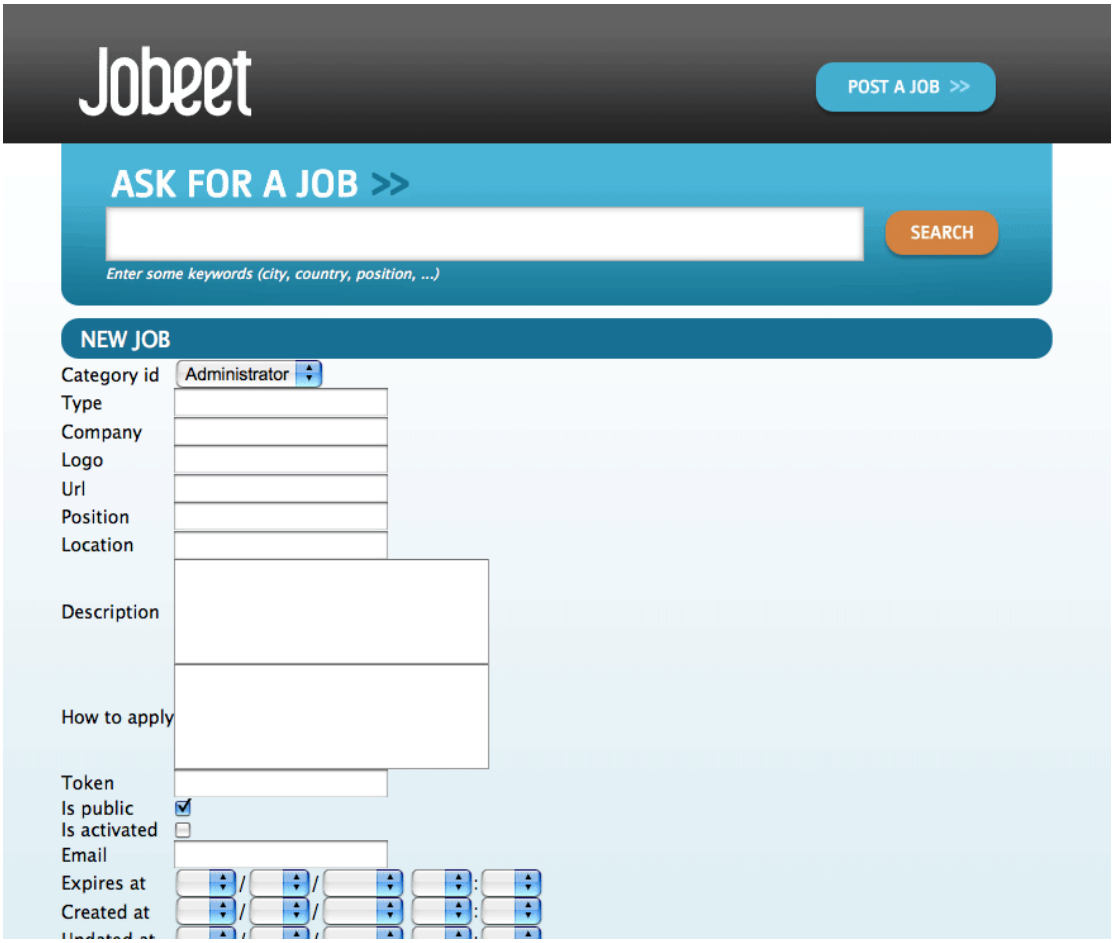

Figura 4.4. El módulo job con el layout y las imágenes y hojas de estilos

Sugerencia

La tarea `generate:project` crea por defecto tres directorios para guardar los archivos relacionados con la web: `web/images/` para las imágenes, `web/css/` para las hojas de estilos y `web/js/` para los archivos de JavaScript. Se trata de otra de las convenciones que sigue Symfony, pero si lo deseas, puedes guardar tus archivos en cualquier otro directorio dentro del directorio `web/`.

Si has investigado el código HTML de las páginas del módulo `job`, habrás visto que aunque el archivo `main.css` no se incluye en el layout, está presente en todas las páginas. ¿Cómo es posible que se incluya un archivo CSS que no se encuentra en el layout?

La respuesta es que la hoja de estilos se ha incluido mediante la llamada a la función `include_stylesheets()` que se realiza dentro de la sección `<head>` del layout. La función `include_stylesheets()` se conoce con el nombre de **helper**. Un helper es una función de Symfony a la que se le pueden pasar parámetros y que devuelve código HTML. Los helpers se utilizan casi siempre para mejorar la productividad en el desarrollo, ya que suelen generar fragmentos de código que se utilizan habitualmente en las plantillas. El helper `include_stylesheets()` genera las etiquetas `<link>` necesarias para enlazar las hojas de estilo. Pero, ¿cómo sabe el helper los archivos CSS que tiene que incluir?

La capa de la vista se puede configurar mediante el archivo de configuración `view.yml` de la aplicación. A continuación se muestra el archivo que genera por defecto la tarea `generate:app`:

```
# 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.css]

  javascripts:    []

  has_layout:     on
  layout:         layout
```

El archivo `view.yml` se emplea para configurar las opciones por defecto (`default`) de todas las plantillas de la aplicación. La opción `stylesheets` por ejemplo define un array que contiene el nombre de las hojas de estilo que se incluyen en cada página de la aplicación (esta información es la que utiliza el helper `include_stylesheets()` para incluir los archivos CSS en las páginas).

Nota

En el archivo `view.yml` por defecto, la referencia de la hoja de estilos es `main.css` y no `/css/main.css`. En realidad, las dos referencias anteriores son equivalentes, ya que Symfony añade automáticamente el prefijo `/css` a las rutas relativas.

Si se indican varios archivos, Symfony los incluye en el mismo orden en el que se han indicado:

```
stylesheets:    [main.css, jobs.css, job.css]
```

También es posible añadir el atributo `media` para cada archivo y también se puede omitir el sufijo `.css`:

```
stylesheets:    [main.css, jobs.css, job.css, print: { media: print }]
```

La configuración anterior se convierte en el siguiente código HTML:

```
<link rel="stylesheet" type="text/css" media="screen" href="/css/main.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/jobs.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/job.css" />
<link rel="stylesheet" type="text/css" media="print" href="/css/print.css" />
```

Sugerencia

El archivo de configuración `view.yml` también establece el layout por defecto que utilizan las páginas de la aplicación. Inicialmente su nombre es `layout`, por lo que Symfony decora todas las páginas con el archivo `layout.php`. También es posible deshabilitar la decoración de las páginas indicando un valor `false` en la opción `has_layout`.

Aunque la configuración actual funciona correctamente, el archivo `jobs.css` sólo es necesario en la portada del sitio y el archivo `job.css` sólo debe incluirse en la página que muestra cada oferta de trabajo. Cada módulo de la aplicación puede definir su propio archivo de configuración `view.yml`, por lo que modifica el archivo `view.yml` de la aplicación para que sólo incluya el archivo `main.css`:

```
# apps/frontend/config/view.yml
stylesheets:    [main.css]
```

Para modificar la parte de la vista del módulo `job`, crea un nuevo archivo `view.yml` en el directorio `apps/frontend/modules/job/config/` y añade el siguiente contenido:

```
# apps/frontend/modules/job/config/view.yml
indexSuccess:
  stylesheets: [jobs.css]

showSuccess:
  stylesheets: [job.css]
```

Como se verá más adelante, `indexSuccess` y `showSuccess` son los nombres de las plantillas asociadas con las acciones `index` y `show`. El archivo `view.yml` del módulo utiliza estos nombres para crear las secciones que modifican el aspecto de cada acción. En cada sección se pueden establecer las mismas opciones que se encuentran en la sección `default` del archivo `view.yml` de la aplicación. Cuando no se define el valor de alguna opción en el archivo `view.yml` del módulo, Symfony lo toma directamente del archivo `view.yml` de la aplicación. Si quieres establecer una misma opción para todas las acciones del módulo, debes hacerlo bajo una sección especial llamada `all`.

Cómo funcionan los archivos de configuración en Symfony

En la mayoría de archivos de configuración de Symfony, se puede establecer la misma opción en diferentes niveles:

- La configuración por defecto, que se encuentra en los propios archivos del framework
- La configuración global del proyecto, que se encuentra en `config/`
- La configuración local de la aplicación, que se encuentra en `apps/[nombre_de_aplicacion]/config/`
- La configuración local del módulo, que se encuentra en `apps/[nombre_de_aplicacion]/modules/[nombre_de_modulo]/config/`

Cuando se ejecuta la aplicación, el sistema de configuración de Symfony junta todos los valores de todas las opciones de todos los archivos de configuración y los guarda en la cache para mejorar el rendimiento.

Como regla general, cualquier opción que se puede configurar en un archivo de configuración también se puede configurar mediante código PHP. En el ejemplo anterior, en vez de crear un archivo `view.yml` para el módulo `job`, se podría utilizar el helper `use_stylesheet()` para incluir una hoja de estilos directamente desde la plantilla:

```
<?php use_stylesheet('main.css') ?>
```

Este helper también se puede utilizar en el layout para incluir una hoja de estilos específica en todas las páginas de la aplicación.

Elegir un método u otro para configurar la parte de la vista es una cuestión de preferencias personales. Realizar la configuración con un archivo `view.yml` permite definir opciones para todas las acciones del módulo, algo que no es posible desde una plantilla, pero la configuración es bastante estática. Por otra parte, realizar la configuración con el helper `use_stylesheet()` es más flexible y además permite disponer en el mismo lugar del código HTML y de la definición de los archivos CSS. Jobeet va a hacer uso del helper `use_stylesheet()`, por lo que puedes borrar el archivo `view.yml` que acabamos de crear y puedes actualizar las plantillas con las llamadas al helper `use_stylesheet()`:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php use_stylesheet('job.css') ?>
```

Nota

De la misma forma, la configuración de los archivos JavaScript se realiza mediante la opción `javascripts` del archivo de configuración `view.yml` o mediante llamadas al helper `use_javascript()` desde una plantilla.

4.4. La portada del módulo de las ofertas de trabajo

Como se explicó en la lección anterior, la portada del módulo `job` se genera en una acción llamada `index`. La acción es la parte del controlador de esta página y la plantilla asociada (llamada `indexSuccess.php`) es la parte de la vista:

```
apps/
  frontend/
    modules/
      job/
        actions/
          actions.class.php
        templates/
          indexSuccess.php
```

4.4.1. La acción

Las acciones se definen como métodos de una clase. Para la portada que estamos creando, la clase se llama `jobActions` (siempre es el nombre del módulo seguido por la palabra `Actions`) y el método se llama `executeIndex()` (siempre es la palabra `execute` seguida del nombre de la acción). Lo único que hace esta acción es obtener la información de todas las ofertas de trabajo de la base de datos:

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_job_list = JobeetJobPeer::doSelect(new Criteria());
  }

  // ...
}
```

Entrando en el detalle del código anterior, se puede observar que el método `executeIndex()` (que es el controlador) realiza llamadas a los métodos de la clase `JobeetJobPeer` del modelo para obtener la lista de todas las ofertas de trabajo (gracias a `new Criteria()`). Este método devuelve un array de objetos de tipo `JobeetJob`, que se asigna a la propiedad `jobeet_job_list` del objeto.

Todas las propiedades de este objeto se pasan automáticamente a la plantilla, que es la parte de la vista. Para pasar datos del controlador a la vista, lo único que tienes que hacer es crear una propiedad en el objeto mediante `$this->nombreDeLaPropiedad`:

```
public function executeFooBar(sfWebRequest $request)
{
  $this->foo = 'bar';
  $this->bar = array('bar', 'baz');
}
```

El código anterior permite que en la plantilla existan dos variables llamadas `$foo` y `$bar` que contienen los valores establecidos en la acción.

4.4.2. La plantilla

Symfony utiliza por defecto una convención para deducir el nombre de la plantilla asociada a cada acción y que consiste en el nombre de la acción seguido de la palabra `Success`. Por tanto, la plantilla llamada `indexSuccess.php` es la que genera todo el código HTML de la tabla que muestra el listado de ofertas de trabajo. A continuación se muestra el código completo de la plantilla:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<h1>Job List</h1>

<table>
```

```
  <thead>
    <tr>
      <th>Id</th>
      <th>Category</th>
      <th>Type</th>
<!-- more columns here -->
      <th>Created at</th>
      <th>Updated at</th>
    </tr>
  </thead>
  <tbody>
    <?php foreach ($jobeet_job_list as $jobeet_job): ?>
    <tr>
      <td>
        <a href="<?php echo url_for('job/show?id='.$jobeet_job->getId()) ?>">
          <?php echo $jobeet_job->getId() ?>
        </a>
      </td>
      <td><?php echo $jobeet_job->getCategoryId() ?></td>
      <td><?php echo $jobeet_job->getType() ?></td>
<!-- more columns here -->
      <td><?php echo $jobeet_job->getCreatedAt() ?></td>
      <td><?php echo $jobeet_job->getUpdatedAt() ?></td>
    </tr>
    <?php endforeach; ?>
  </tbody>
</table>

<a href="<?php echo url_for('job/new') ?>">New</a>
```

En el código de la plantilla anterior, se emplea una sentencia `foreach` para recorrer la lista de objetos de tipo `Job` (almacenados en la variable `$jobeet_job_list`) y para cada oferta de trabajo, se muestra el valor de todas sus columnas. Para acceder al valor de cada columna, puedes utilizar un método generado automáticamente y que se construye uniendo la palabra `get` junto con el nombre de la columna en formato *camelCase*. El formato *camelCase* consiste en eliminar los guiones bajos del nombre original de la columna y escribir en mayúsculas la primera letra de cada palabra. De esta forma, la columna `created_at` tiene un método asociado llamado `getCreatedAt()`.

El código anterior muestra el valor de todas las columnas de los objetos, pero en la aplicación real sólo queremos mostrar algunas de las columnas disponibles:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<div id="jobs">
  <table class="jobs">
    <?php foreach ($jobeet_job_list as $i => $job): ?>
      <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
        <td class="location"><?php echo $job->getLocation() ?></td>
        <td class="position">
          <a href="<?php echo url_for('job/show?id='.$job->getId()) ?>">
            <?php echo $job->getPosition() ?>
```

```
          </a>
        </td>
        <td class="company"><?php echo $job->getCompany() ?></td>
      </tr>
    <?php endforeach; ?>
  </table>
</div>
```

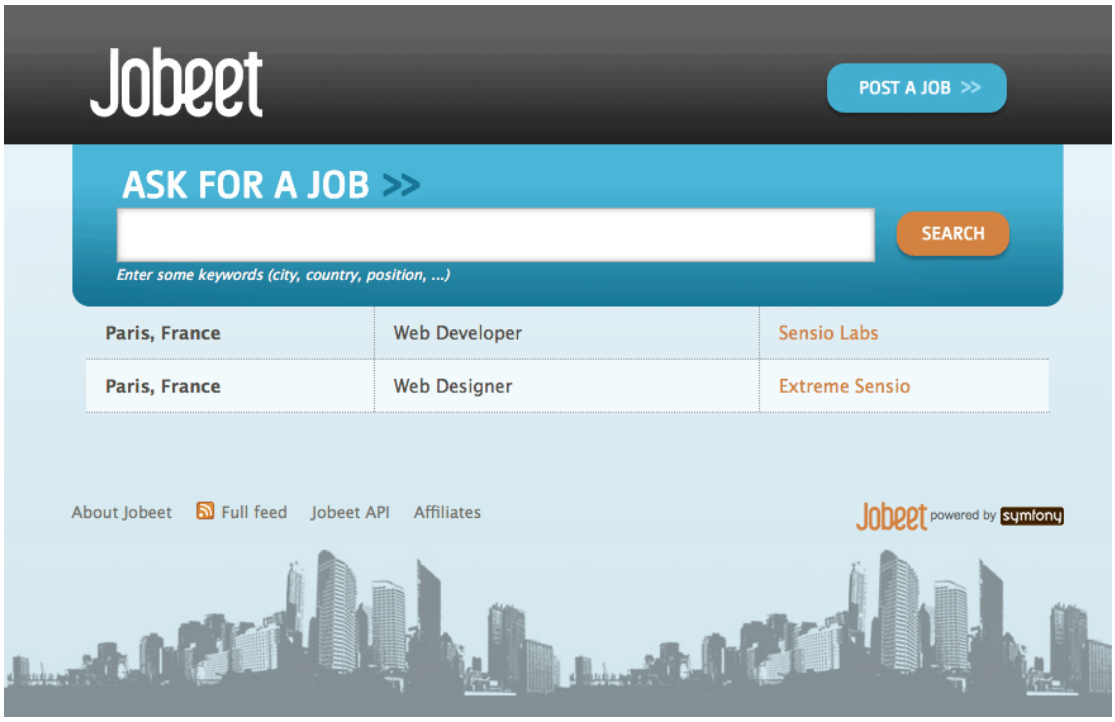

Figura 4.5. La página principal

La función `url_for()` utilizada en la plantilla anterior es un helper muy útil de Symfony que explicaremos en la lección de mañana.

4.5. La plantilla de la página de una oferta de trabajo

A continuación se va a modificar la plantilla de la página que muestra los detalles de una oferta de trabajo. Abre el archivo `showSuccess.php` y reemplaza todo su contenido por el siguiente código PHP:

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php use_stylesheet('job.css') ?>
<?php use_helper('Text') ?>

<div id="job">
  <h1><?php echo $job->getCompany() ?></h1>
  <h2><?php echo $job->getLocation() ?></h2>
  <h3>
    <?php echo $job->getPosition() ?>
    <small> - <?php echo $job->getType() ?></small>
  </h3>

  <?php if ($job->getLogo()): ?>
    <div class="logo">
```

```
      <a href="<?php echo $job->getUrl() ?>">
        <img src="/uploads/jobs/<?php echo $job->getLogo() ?>"
          alt="<?php echo $job->getCompany() ?> logo" />
      </a>
    </div>
  <?php endif; ?>

  <div class="description">
    <?php echo simple_format_text($job->getDescription()) ?>
  </div>

  <h4>How to apply?</h4>

  <p class="how_to_apply"><?php echo $job->getHowToApply() ?></p>

  <div class="meta">
    <small>posted on <?php echo $job->getCreatedAt('m/d/Y') ?></small>
  </div>

  <div style="padding: 20px 0">
    <a href="<?php echo url_for('job/edit?id='.$job->getId()) ?>">Edit</a>
  </div>
</div>
```

Para mostrar los detalles de la oferta de trabajo, la plantilla hace uso de una variable llamada `$job` que se debe pasar desde la acción. Como en la acción `show` esta variable se llama `$jobeet_job`, es necesario modificar su nombre (ten en cuenta que en la acción esta variable aparece dos veces):

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeShow(sfWebRequest $request)
{
  $this->job = JobeetJobPeer::retrieveByPk($request->getParameter('id'));
  $this->forward404Unless($this->job);
}
```

Algunos métodos accesores de Propel también admiten argumentos. Como se ha definido una columna llamada `created_at` de tipo timestamp, el método `getCreatedAt()` permite establecer como su primer argumento el formato en el que se quiere obtener la fecha:

```
$job->getCreatedAt('m/d/Y');
```

Nota

Para mostrar la descripción de la oferta de trabajo en formato HTML, se utiliza el helper `simple_format_text()`, ya que entre otras cosas, reemplaza los saltos de línea por etiquetas `<br/>`. Como este helper pertenece al grupo de helpers llamado `Text` y Symfony no lo carga por defecto, tenemos que cargarlo a mano mediante el helper `use_helper()`.

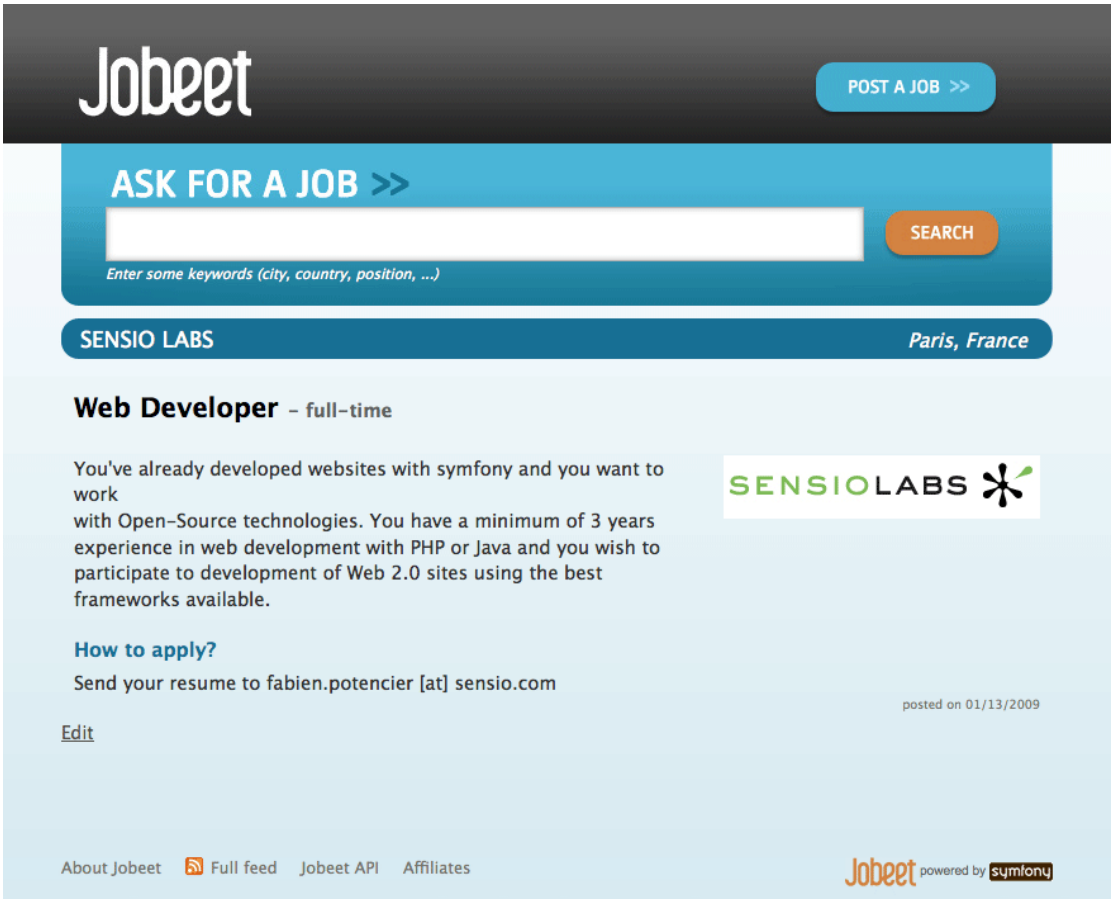

Figura 4.6. La página de una oferta de trabajo

4.6. Slots

Por el momento, el título de toda las páginas de la aplicación es el mismo y se define en la etiqueta `<title>` del layout:

```
<title>Jobeet - Your best job board</title>
```

Aunque se trata de un título correcto, en algunas páginas como la de detalle de una oferta de trabajo es mucho más útil mostrar información como el nombre de la empresa y el puesto de trabajo. En Symfony, cuando una zona del layout depende de la plantilla, tienes que utilizar *slots*:

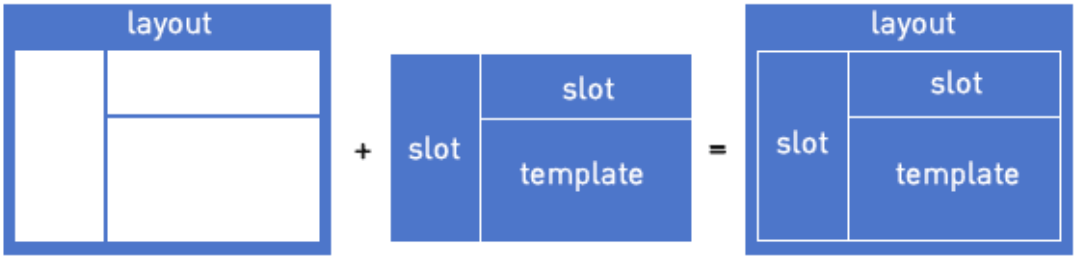

Figura 4.7. Funcionamiento básico de los slots

Añade el siguiente código al layout para que el título de la página sea dinámico:

```
// apps/frontend/templates/layout.php
<title><?php include_slot('title') ?></title>
```

Los slots se definen con un nombre único (en este caso, `title`) y se muestran con el helper `include_slot()`. Después de incluir el slot en el layout, ahora tienes que utilizar el helper `slot()` en la plantilla para establecer su valor:

```
// apps/frontend/modules/job/templates/showSuccess.php
<?php slot(
  'title',
  sprintf('%s is looking for a %s', $job->getCompany(), $job->getPosition()))
?>
```

Si el valor del slot es difícil de generar, el helper `slot()` se puede utilizar en forma de bloque de código:

```
// apps/frontend/modules/job/templates/showSuccess.php
<?php slot('title') ?>
  <?php echo sprintf('%s is looking for a %s', $job->getCompany(),
$job->getPosition()) ?>
<?php end_slot(); ?>
```

Algunas páginas, como por ejemplo la portada, muestran un título genérico. Para no tener que definir una y otra vez el mismo título en varias plantillas, se puede establecer en el layout un valor por defecto para el slot `title`:

```
// apps/frontend/templates/layout.php
<title>
  <?php if (!include_slot('title')): ?>
    Jobeet - Your best job board
  <?php endif; ?>
</title>
```

El helper `include_slot()` devuelve el valor `true` si se ha definido algún valor para el slot. Por tanto, cuando se establece el valor del slot, el código anterior lo muestra normalmente. En otro caso, se muestra un título genérico.

Sugerencia

Hasta ahora se han visto varios helpers que empiezan por `include_`. Estos helpers muestran directamente el código HTML y la mayoría disponen de una versión alternativa que empieza por `get_` y que sólo devuelve el contenido, sin mostrarlo.

```
<?php include_slot('title') ?>
<?php echo get_slot('title') ?>

<?php include_stylesheets() ?>
<?php echo get_stylesheets() ?>
```

4.7. La acción de la página de una oferta de trabajo

La página que muestra el detalle de cada oferta de trabajo se genera en la acción `show`, definida en el método `executeShow()` del módulo `job`:

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = JobeetJobPeer::retrieveByPk($request->getParameter('id'));
    $this->forward404Unless($this->job);
  }

  // ...
}
```

Al igual que sucedía en la acción `index`, se emplea la clase `JobeetJobPeer` para obtener los datos de una oferta de trabajo. La principal diferencia es que en esta ocasión se emplea el método `retrieveByPk()`. El parámetro que se debe pasar a este método es el identificador único de la oferta, es decir, su clave primaria. En la siguiente sección se explica por qué la sentencia `$request->getParameter('id')` devuelve la clave primaria de la oferta de trabajo.

Sugerencia

Las clases del modelo generadas automáticamente contienen muchos métodos útiles para interactuar con los objetos del proyecto. Te recomendamos que dediques un tiempo a investigar el código que se encuentra en el directorio `lib/om/` para descubrir todas las utilidades de estas clases.

Cuando la oferta de trabajo solicitada no existe en la base de datos, se redirige al usuario a una página de tipo `404` gracias al método `forward404Unless()`. El primer argumento del método es un valor booleano. Si este valor no es `true`, se detiene la ejecución de la acción actual. No es necesario devolver ningún valor porque se lanza una excepción de tipo `sfError404Exception`.

En cuanto a las excepciones, la página que se muestra es diferente en función de si la aplicación se ejecuta en el entorno de producción o en el de desarrollo:

Figura 4.8. Error 404 en el entorno de desarrollo

Figura 4.9. Error 404 en el entorno de producción

Nota

Antes de que subas el sitio web de Jobeet al servidor de producción, vamos a explicar cómo personalizar la página del error 404.

El conjunto de métodos forward

La llamada a `forward404Unless` es equivalente a :

```
$this->forward404If(!$this->job);
```

Que a su vez es equivalente a:

```
if (!$this->job)
{
  $this->forward404();
}
```

Además, el método `forward404()` no es más que un atajo de:

```
$this->forward('default', '404');
```

El método `forward()` reenvía la ejecución a otra acción de la misma aplicación. En el ejemplo anterior se reenvía a la acción `404` del módulo `default`. Este módulo lo incluye Symfony por defecto y contiene las acciones necesarias para mostrar la página del error 404, la página que indica que son necesarias credenciales de seguridad y la página que muestra un formulario de login.

4.8. La petición y la respuesta

Cuando accedes a la página `/job` o `/job/show/id/1` en tu navegador, estás interactuando con el servidor web. El navegador envía una **petición** y el servidor web devuelve una **respuesta**.

Como ya se ha visto en el código de los ejemplos anteriores, Symfony encapsula la petición en un objeto de tipo `sfWebRequest` (como se puede ver por ejemplo en la declaración del método `executeShow()`). Como Symfony es un framework orientado a objetos, la respuesta también es un objeto, en este caso de tipo `sfWebResponse`. Si

quieres acceder al objeto de la respuesta desde la acción, puedes llamar al método `$this->getResponse()`.

Estos dos objetos incluyen muchos métodos útiles para acceder a la información desde funciones y variables globales de PHP.

> **Nota**
>
> ¿Cuál es el motivo por el que Symfony añade una capa de abstracción sobre algunas funcionalidades de PHP? En primer lugar, los métodos de Symfony son mucho más poderosos que los métodos equivalentes de PHP. En segundo lugar, porque cuando pruebas una aplicación es mucho más fácil simular una petición o una respuesta mediante un objeto, en vez de utilizar variables globales o funciones de PHP como `header()`, que ocultan gran parte de su funcionamiento interno.

4.8.1. La petición

La clase `sfWebRequest` encapsula los arrays globales `$_SERVER`, `$_COOKIE`, `$_GET`, `$_POST` y `$_FILES`:

Nombre del método	Equivalente de PHP
`getMethod()`	`$_SERVER['REQUEST_METHOD']`
`getUri()`	`$_SERVER['REQUEST_URI']`
`getReferer()`	`$_SERVER['HTTP_REFERER']`
`getHost()`	`$_SERVER['HTTP_HOST']`
`getLanguages()`	`$_SERVER['HTTP_ACCEPT_LANGUAGE']`
`getCharsets()`	`$_SERVER['HTTP_ACCEPT_CHARSET']`
`isXmlHttpRequest()`	`$_SERVER['X_REQUESTED_WITH'] == 'XMLHttpRequest'`
`getHttpHeader()`	`$_SERVER`
`getCookie()`	`$_COOKIE`
`isSecure()`	`$_SERVER['HTTPS']`
`getFiles()`	`$_FILES`
`getGetParameter()`	`$_GET`
`getPostParameter()`	`$_POST`
`getUrlParameter()`	`$_SERVER['PATH_INFO']`
`getRemoteAddress()`	`$_SERVER['REMOTE_ADDR']`

En el código de los ejemplos anteriores también se ha empleado el método `getParameter()`, que permite acceder a los parámetros de la petición. El valor que devuelve este método se obtiene de las variables globales `$_GET` y `$_POST` o de la variable `PATH_INFO`.

Si quieres asegurarte de que un parámetro de la petición viene específicamente de una de esas variables, puedes utilizar respectivamente los métodos `getGetParameter()`, `getPostParameter()` y `getUrlParameter()`.

Nota

Si quieres restringir una acción a un método específico, por ejemplo para asegurar que un formulario se ha enviado con el método `POST`, puedes utilizar el método `isMethod()` de la siguiente manera: `$this->forwardUnless($request->isMethod('POST'));`.

4.8.2. La respuesta

La clase `sfWebResponse` encapsula los métodos `header()` y `setrawcookie()` de PHP:

Nombre del método	Equivalente de PHP
`setCookie()`	`setrawcookie()`
`setStatusCode()`	`header()`
`setHttpHeader()`	`header()`
`setContentType()`	`header()`
`addVaryHttpHeader()`	`header()`
`addCacheControlHttpHeader()`	`header()`

Obviamente, la clase `sfWebResponse` también incluye un método para establecer el contenido de la respuesta (`setContent()`) y otro para enviarla al navegador (`send()`).

En las secciones anteriores se ha mostrado cómo incluir hojas de estilos y archivos JavaScript tanto en el archivo `view.yml` como en las plantillas. En realidad, las dos técnicas utilizan los métodos `addStylesheet()` y `addJavascript()` del objeto de la respuesta.

Sugerencia

Las clases sfAction (http://www.symfony-project.org/api/1_2/sfAction), sfRequest (http://www.symfony-project.org/api/1_2/sfRequest) y sfResponse (http://www.symfony-project.org/api/1_2/sfResponse) incluyen muchos otros métodos útiles. Puedes consultar la documentación de la API de Symfony 1.2 (http://www.symfony-project.org/api/1_2/) para aprenderlo todo sobre las clases internas de Symfony.

4.9. Nos vemos mañana

Hoy hemos hablado sobre algunos de los patrones de diseño que utiliza Symfony. Seguramente ahora comprendes mejor la estructura de directorios de Symfony. También hemos trabajado con las plantillas mediante el layout y las plantillas de los módulos. Además, hemos hecho las plantillas más dinámicas gracias a los slots y las acciones.

En la lección de mañana explicaremos el helper `url_for()` que hemos utilizado hoy y también nos adentraremos en el sistema de enrutamiento.

Capítulo 5. El sistema de enrutamiento

Si has seguido la lección de ayer, ahora estarás más familiarizado con el patrón de diseño MVC y lo verás como una forma muy natural de programar aplicaciones web. Si continúas programando siguiendo este patrón, dentro de poco ya no querrás volver a programar como lo hacías antes. Ayer también modificamos las páginas de la aplicación Jobeet y de paso, aprendimos conceptos importantes de Symfony como el layout, los helpers y los slots.

En la lección de hoy nos vamos a adentrar en el maravilloso mundo del sistema de enrutamiento de Symfony.

5.1. URLs

Si pinchas el enlace de cualquier oferta de trabajo de la portada de Jobeet, la URL de la página de detalle será algo como `/job/show/id/1`. Seguramente, si tienes experiencia programando sitios web con PHP, estás más acostumbrado a URL parecidas a `/job.php?id=1`. ¿Cómo funcionan las URL en Symfony? ¿Cómo sabe Symfony qué acción se tiene que ejecutar en base a esa URL? ¿Por qué se obtiene el `id` de la oferta de trabajo mediante `$request->getParameter('id')`? Hoy vamos a contestar a todas estas preguntas.

En primer lugar vamos a hablar de las URL y vamos a explicar exactamente en qué consisten. En el ámbito de la web, una URL es el identificador único de un recurso web. Cuando accedes a una URL, en realidad estás solicitando al navegador que obtenga el recurso identificado por esa URL.

Como la URL es la forma en la que el usuario interactúa con el sitio web, debe incluir toda la información necesaria para localizar el recurso al que hace referencia. Sin embargo, las URL tradicionales no describen el recurso, sino que directamente muestran la estructura interna de la aplicación. Al usuario no le importa si tu sitio está programado con PHP o si las ofertas de trabajo tienen un identificador en la base de datos.

Mostrar la estructura interna de la aplicación también es una mala idea desde el punto de vista de la seguridad. ¿Qué sucede si un usuario intenta adivinar la URL de recursos para los que no tiene permiso de acceso? Obviamente el programador habrá restringido su acceso, pero siempre es mejor ocultar este tipo de información delicada.

Las URL son tan importantes dentro de Symfony que tienen todo un sub-framework dedicado a trabajar con las URL: el sistema de **enrutamiento**. Este sub-framework gestiona las URI internas y las URL externas. Cuando la aplicación recibe una petición, el sistema de enrutamiento procesa la URL y la convierte en una URI interna.

En las lecciones anteriores ya se ha visto la URI interna de la página de detalle de una oferta de trabajo en la plantilla `showSuccess.php`:

```
'job/show?id='.$job->getId()
```

El helper `url_for()` se encarga de convertir esta URI interna en una URL correcta:

```
/job/show/id/1
```

Las URI internas se componen de varias partes:

- `job` es el nombre del módulo.
- `show` es el nombre de la acción
- El resto es la *query string*, que define los parámetros que se pasan a la acción

Por tanto, el patrón genérico de las URI internas es:

```
nombre_de_modulo/nombre_de_accion?clave1=valor1&clave2=valor2&...
```

Como el sistema de enrutamiento de Symfony es bidireccional, puedes modificar las URL sin modificar el funcionamiento interno de la aplicación. Esta es una de las ventajas principales del patrón de diseño del controlador frontal.

5.2. Configurando el enrutamiento

La conversión entre URI internas y URL externas se define en el archivo de configuración `routing.yml`:

```
# apps/frontend/config/routing.yml
homepage:
  url:   /
  param: { module: default, action: index }

default_index:
  url:   /:module
  param: { action: index }

default:
  url:   /:module/:action/*
```

El archivo `routing.yml` describe las rutas de la aplicación. Cada ruta está formada por un nombre (`homepage`), un patrón (`/:module/:action/*`) y unos parámetros (dentro de la opción `param`).

Cuando la aplicación recibe una petición, el sistema de enrutamiento trata de encontrar el patrón que coincide con la URL solicitada. El orden en el que se añaden las rutas al archivo `routing.yml` es muy importante, ya que siempre se utiliza la primera ruta cuyo patrón cumple las condiciones de la URL y siempre se empieza a buscar desde la primera hasta la última ruta. A continuación vamos a utilizar algunos ejemplos para comprender mejor su funcionamiento.

Cuando accedes a la portada de Jobeet, la URL es `/job`, por lo que la primera ruta cuyo patrón coincide con la URL es `default_index`. En los patrones, cuando una palabra empieza por dos puntos (`:`) se considera que es una variable, por lo que el patrón `/:module` significa: cualquier URL que sea una barra `/` seguida de cualquier contenido. En este ejemplo, la variable `module` tendrá como valor la palabra `job`. Después, este valor se puede obtener en la acción mediante `$request->getParameter('module')`. La ruta `default_index` también define un valor por defecto para la variable llamada `action`. Por tanto, cuando una URL cumple con el patrón de esta ruta, a la petición se le añade un parámetro llamado `action` que vale `index`.

Si ahora accedes a la página `/job/show/id/1`, Symfony detecta que el patrón que se cumple es el de la última ruta `/:module/:action/*`. En los patrones, un asterisco (`*`) es equivalente a una sucesión de pares clave/valor separados por barras (`/`). Por tanto, la URL `/job/show/id/1` se intepreta de la siguiente forma:

Parámetro de la petición	**Valor**
`module`	`job`
`action`	`show`
`id`	`1`

Nota

Las variables llamadas `module` y `action` son especiales, ya que las emplea Symfony para determinar la acción que se ejecuta.

La URL `/job/show/id/1` se puede crear en una plantilla mediante la siguiente llamada al helper `url_for()`:

```
url_for('job/show?id='.$job->getId())
```

Si lo prefieres, puedes utilizar directamente el nombre de la ruta prefijándolo con el carácter `@`:

```
url_for('@default?module=job&action=show&id='.$job->getId())
```

Aunque las dos formas son equivalentes, la segunda es mucho más rápida porque Symfony no tiene que procesar todas las rutas para encontrar la ruta cuyo patrón cumple con la URL. Además, la segunda forma es mucho más flexible, ya que no depende del nombre de los módulos y de las acciones.

5.3. Personalizando el enrutamiento

Por el momento, cuando accedes a la URL `/`, se muestra la página de bienvenida por defecto de Symfony. El motivo es que esa URL cumple con el patrón de la ruta `homepage`. No obstante, parece lógico modificar esa URL para que apunte a la página principal de Jobeet. Para ello, sustituye el valor de la variable `module` por `job` en la ruta `homepage`:

```
# apps/frontend/config/routing.yml
homepage:
  url:   /
  param: { module: job, action: index }
```

Ahora también podemos modificar el enlace del logotipo de Jobeet en el layout para que apunte a la ruta `homepage`:

```
<!-- apps/frontend/templates/layout.php -->
<h1>
  <a href="<?php echo url_for('@homepage') ?>">
    <img src="/images/jobeet.gif" alt="Jobeet Job Board" />
  </a>
</h1>
```

Como lo anterior ha sido muy fácil, vamos a ver un ejemplo más complejo, que consiste en modificar las URL de las páginas de detalle de las ofertas de trabajo por algo más útil, como por ejemplo:

```
/job/sensio-labs/paris-france/1/web-developer
```

Sin conocer nada de Jobeet y sin ni siquiera ver la página, a partir de la URL ya sabes que una empresa llamada Sensio Labs está buscando programadores web para trabajar en París, Francia.

Nota

Las URL *limpias* son muy importantes porque proporcionan información al usuario. Además son muy útiles para poder copiarlas y pegarlas en un email y para optimizar tu sitio web para los buscadores.

A continuación se muestra un posible patrón que cumple las condiciones de esa URL:

```
/job/:company/:location/:id/:position
```

Modifica el archivo `routing.yml` y añade una nueva ruta llamada `job_show_user` al principio del archivo:

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  param: { module: job, action: show }
```

Si ahora vuelves a acceder a la portada de Jobeet, verás que los enlaces no se han cambiado. El motivo es que para generar una ruta, tienes que pasar todas las variables necesarias. Por tanto, modifica la llamada al helper `url_for()` en la plantilla `indexSuccess.php`:

```
url_for('job/
show?id='.$job->getId().'&company='.$job->getCompany().'&location='.$job->getLocation().'&position=
```

Las URI internas también se pueden expresar utilizando la notación de los arrays:

```
url_for(array(
  'module'   => 'job',
  'action'   => 'show',
```

```
  'id'       => $job->getId(),
  'company'  => $job->getCompany(),
  'location' => $job->getLocation(),
  'position' => $job->getPosition(),
))
```

5.4. Requisitos

Durante el tutorial del primer día explicamos la necesidad de la validación de datos y la gestión de errores. El sistema de enrutamiento incluye su propio mecanismo de validación. En la opción `requirements` de cada ruta se puede indicar una expresión regular con las condiciones que debe cumplir el patrón:

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  param: { module: job, action: show }
  requirements:
    id:  \d+
```

La opción `requirements` anterior obliga a que el valor de la variable `id` sea un número. Si la URL que se pasa no cumple esta condición, no se produce una coincidencia con el patrón de la ruta y Symfony sigue buscando coincidencias en el resto de rutas.

5.5. La clase sfRoute

Las rutas definidas en el archivo `routing.yml` se convierten internamente en objetos de la clase sfRoute (http://www.symfony-project.org/api/1_2/sfRoute) . Si quieres utilizar otra clase, puedes indicarlo en la opción `class` de la definición de la ruta.

Si conoces el protocolo HTTP, sabrás que define diferentes métodos para realizar las peticiones: `GET`, `POST`, `HEAD`, `DELETE` y `PUT`. Los tres primeros métodos los soportan todos los navegadores, pero los últimos dos métodos no están soportados.

Si quieres restringir una ruta para que sólo se tenga en cuenta para unos métodos HTTP específicos, puedes modificar la clase de la ruta por sfRequestRoute (http://www.symfony-project.org/api/1_2/sfRequestRoute) y añadir la restricción en la variable virtual `sf_method`:

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  class: sfRequestRoute
  param: { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

Nota

Restringir una ruta a unos métodos HTTP específicos no es exactamente lo mismo que utilizar `sfWebRequest::isMethod()` en las acciones. El motivo es que, cuando el método HTTP no es el que se requiere, el sistema de enrutamiento sigue buscando entre las siguientes rutas.

5.6. La clase para las rutas basadas en objetos

La URI interna de la página de una oferta de trabajo es muy larga y bastante aburrida de escribir (`url_for('job/show?id='.$job->getId().'&company='.$job->getCompany().'&location='.$job->getLocation().'&po`
Como se ha comentado en la sección anterior, es posible modificar la clase que utiliza cada ruta. En el caso de la ruta llamada `job`, se va a emplear la clase sfPropelRoute (http://www.symfony-project.org/api/1_2/sfPropelRoute) , ya que es una clase optimizada para las rutas que representan objetos Propel o colecciones de objetos Propel:

```
job_show_user:
  url:     /job/:company/:location/:id/:position
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

La opción `options` establece el comportamiento de la ruta. La opción `model` define la clase del modelo de Propel relacionada con la ruta (en este caso, `JobeetJob`) y la opción `type` indica que esta ruta está relacionada con un solo objeto. Si la ruta representara una colección de objetos, se debería utilizar el valor `list` en esta opción `type`.

Como la ruta `job_show_user` ahora está relacionada con `JobeetJob`, se puede simplificar la llamanda al helper `url_for()` de la siguiente manera:

```
url_for(array('sf_route' => 'job_show_user', 'sf_subject' => $job))
```

Incluso se puede simplificar todavía más:

```
url_for('job_show_user', $job)
```

> **Nota**
>
> La primera forma es útil cuando tienes que pasar más argumentos aparte del objeto.

Todo esto es posible porque todas las variables de la ruta tienen un método para acceder a su valor dentro de la clase `JobeetJob`. La variable `company` por ejemplo se sustituye por el valor devuelto por el método `getCompany()`.

Si observas el aspecto de las URL generadas, verás que todavía no son exactamente como queríamos:

```
http://jobeet.localhost/frontend_dev.php/job/Sensio+Labs/Paris%2C+France/1/
Web+Developer
```

El siguiente paso consiste en preparar los valores de cada columna para que se muestren correctamente en la URL, proceso que se conoce con el nombre de *slugify*, por

lo que debemos sustituir todos los caracteres que no sean ASCII por un guión medio -. Para ello, abre el archivo `JobeetJob` y añade los siguientes métodos en la clase:

```
// lib/model/JobeetJob.php
public function getCompanySlug()
{
  return Jobeet::slugify($this->getCompany());
}

public function getPositionSlug()
{
  return Jobeet::slugify($this->getPosition());
}

public function getLocationSlug()
{
  return Jobeet::slugify($this->getLocation());
}
```

A continuación, crea un archivo llamado `lib/Jobeet.class.php` y añade el método `slugify` a la nueva clase:

```
// lib/Jobeet.class.php
class Jobeet
{
  static public function slugify($text)
  {
    // replace all non letters or digits by -
    $text = preg_replace('/\W+/', '-', $text);

    // trim and lowercase
    $text = strtolower(trim($text, '-'));

    return $text;
  }
}
```

Los cambios anteriores han creado tres métodos accesores *virtuales*: `getCompanySlug()`, `getPositionSlug()` y `getLocationSlug()`. Los tres métodos devuelven el valor original de la columna de datos después de aplicarle el método `slugify()`. Por tanto, ahora la ruta `job_show_user` también puede hacer uso de estos métodos accesores para reemplazar los valores originales de cada columna por sus valores virtuales:

```
job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

Como acabamos de añadir una nueva clase, antes de refrescar la portada de Jobeet es necesario que borres la cache de Symfony:

```
$ php symfony cc
```

Si vuelves a acceder a la portada de Jobeet, verás que las URL ahora sí que son tal y como las queríamos:

```
http://jobeet.localhost/frontend_dev.php/job/sensio-labs/paris-france/1/
web-developer
```

Todo lo anterior es sólo parte de lo que son capaces las rutas de Symfony. Las rutas pueden generar una URL en función de un objeto, pero también pueden obtener el objeto relacionado con una URL. El objeto relacionado se puede obtener mediante el método `getObject()` del objeto de la ruta. Cuando procesa una petición, el sistema de enrutamiento guarda el objeto relacionado con la ruta para que lo utilices en las acciones. Por tanto, modifica el método `executeShow()` para obtener el objeto `Jobeet` mediante el objeto de la ruta:

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    $this->forward404Unless($this->job);
  }

  // ...
}
```

Si tratas de obtener la oferta de trabajo relacionada con un `id` desconocido, verás una página de error 404, pero esta vez el mensaje ha cambiado:

Figura 5.1. Mensaje de error 404 cuando se utiliza sfPropelRoute

El motivo es que la excepción del error 404 se ha lanzado automáticamente desde el método `getRoute()`. Por tanto, puedes simplificar todavía más el método `executeShow`:

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();
  }
```

```
  // ...
}
```

Sugerencia

Si no quieres que la ruta muestre un error de tipo 404, establece la opción `allow_empty` a `true` en la definición de esa ruta.

Nota

El objeto relacionado con la ruta no se carga de forma automática. Este objeto sólo se obtiene de la base de datos cuando se invoca el método `getRoute()`.

5.7. Enrutamiento en acciones y plantillas

En las plantillas, el helper `url_for()` convierte una URI interna en una URL externa. Otros helpers de Symfony también utilizan una URI interna como argumento, como por ejemplo el helper `link_to()`, que genera una etiqueta `<a>`:

```
<?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
```

El helper anterior genera el siguiente código:

```
<a href="/job/sensio-labs/paris-france/1/web-developer">Web Developer</a>
```

Tanto `url_for()` como `link_to()` también pueden generar URL absolutas si se les pasa el valor `true` como último parámetro:

```
url_for('job_show_user', $job, true);

link_to($job->getPosition(), 'job_show_user', $job, true);
```

Si quieres generar una URL desde una acción, puedes utilizar el método `generateUrl()`:

```
$this->redirect($this->generateUrl('job_show_user', $job));
```

El conjunto de métodos redirect

En el tutorial de ayer explicamos el conjunto de métodos `forward`. Estos métodos reenvían la petición actual a otra acción sin necesidad de pasar por el navegador.

Los métodos `redirect` redireccionan al usuario a otra URL. Al igual que sucede con los métodos `forward`, puedes utilizar el método `redirect()`, o los atajos `redirectIf()` y `redirectUnless()`.

5.8. La clase para las colecciones de rutas

En las secciones anteriores se ha personalizado la ruta de la acción `show` del módulo `job`, pero las URL del resto de métodos (`index`, `new`, `edit`, `create`, `update` y `delete`) siguen utilizando la ruta `default`:

```
default:
  url: /:module/:action/*
```

La ruta `default` es muy útil para empezar a programar sin preocuparse de tener que definir muchas rutas. Pero como esta ruta es totalmente genérica y está preparada para aceptar cualquier cosa, no se puede configurar para nuestras necesidades específicas.

Como todas las acciones del módulo `job` están relacionadas con la clase `JobeetJob` del modelo, se puede definir una ruta de tipo `sfPropelRoute` para cada una de la misma forma que hemos hecho en la acción `show`. No obstante, como el módulo `job` incluye las siete acciones típicas que se realizan sobre los datos del modelo, también podemos utilizar la clase sfPropelRouteCollection (http://www.symfony-project.org/api/1_2/sfPropelRouteCollection) . Por tanto, modifica el archivo `routing.yml` de forma que tenga el siguiente contenido:

```
# apps/frontend/config/routing.yml
job:
  class:   sfPropelRouteCollection
  options: { model: JobeetJob }

job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]

# default rules
homepage:
  url:   /
  param: { module: job, action: index }

default_index:
  url:   /:module
  param: { action: index }

default:
  url:   /:module/:action/*
```

La ruta `job` anterior en realidad es un atajo para que se generen automáticamente las siguientes siete rutas de tipo `sfPropelRoute`:

```
job:
  url:     /job.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: list }
  param:   { module: job, action: index, sf_format: html }
  requirements: { sf_method: GET }

job_new:
  url:     /job/new.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: new, sf_format: html }
```

```
  requirements: { sf_method: get }

job_create:
  url:     /job.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: create, sf_format: html }
  requirements: { sf_method: post }

job_edit:
  url:     /job/:id/edit.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: edit, sf_format: html }
  requirements: { sf_method: get }

job_update:
  url:     /job/:id.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: update, sf_format: html }
  requirements: { sf_method: put }

job_delete:
  url:     /job/:id.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: delete, sf_format: html }
  requirements: { sf_method: delete }

job_show:
  url:     /job/:id.:sf_format
  class:   sfPropelRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show, sf_format: html }
  requirements: { sf_method: get }
```

Nota

Algunas rutas generadas por `sfPropelRouteCollection` tienen exactamente la misma URL. El sistema de enrutamiento es capaz de diferenciarlas porque todas tienen diferentes métodos en la opción `requirements`.

Las rutas `job_delete` y `job_update` utilizan métodos de HTTP que todavía no están soportados en los navegadores (`DELETE` y `PUT` respectivamente). Por tanto, Symfony no tiene más remedio que simular estos métodos utilizando un truco. Si abres la plantilla `_form.php` verás un ejemplo de cómo se hace:

```
// apps/frontend/modules/job/templates/_form.php
<form action="..." ...>
<?php if (!$form->getObject()->isNew()): ?>
  <input type="hidden" name="sf_method" value="PUT" />
<?php endif; ?>
```

```
<?php echo link_to(
  'Delete',
  'job/delete?id='.$form->getObject()->getId(),
  array('method' => 'delete', 'confirm' => 'Are you sure?')
) ?>
```

Los helpers de Symfony pueden simular cualquier método HTTP mediante un parámetro especial llamado `sf_method`.

Nota

Además de `sf_method`, Symfony dispone de otros parámetros especiales cuyo nombre siempre empieza por `sf_`. Las rutas generadas automáticamente en el código anterior tienen otro parámetro especial llamado `sf_format`, que se explicará más adelante.

5.9. Depurando las rutas

Cuando se utilizan colecciones de rutas, suele ser útil listar todas las rutas generadas. La tarea `app:routes` muestra todas las rutas de la aplicación especificada:

```
$ php symfony app:routes frontend
```

Si quieres acceder a toda la información disponible sobre una ruta, indica su nombre como segundo argumento:

```
$ php symfony app:routes frontend job_edit
```

5.10. Rutas por defecto

Una buena práctica al desarrollar aplicaciones web consiste en definir explícitamente las rutas para todas las posibles URL de la aplicación. Como la ruta `job` define todas las rutas necesarias para la aplicación Jobeet, puedes eliminar o comentar las rutas que incluye por defecto el archivo de configuración `routing.yml`:

```
# apps/frontend/config/routing.yml
#default_index:
#  url:   /:module
#  param: { action: index }
#
#default:
#  url:   /:module/:action/*
```

Después de realizar el cambio anterior, la aplicación Jobeet debe seguir funcionando igual que antes.

5.11. Nos vemos mañana

En esta lección hemos explicado muchas cosas nuevas. Además de haber aprendido a utilizar el sub-framework de enrutamiento de Symfony, hemos visto cómo evitar que las URL muestren el funcionamiento interno de la aplicación.

En el tutorial de mañana no vamos a introducir nuevos conceptos, pero vamos a explicar en detalle muchas de las cosas que hemos visto hasta el momento.

Capítulo 6. Profundizando en el modelo

Ayer fue un gran día, ya que aprendimos cómo crear URL limpias y cómo utilizar el framework Symfony para automatizar varias tareas.

Hoy nos vamos a centrar en mejorar el sitio web de Jobeet realizando modificaciones en todas sus características. Al mismo tiempo vamos a profundizar en todos los conceptos que hemos estudiado durante los primeros cinco días del tutorial.

6.1. El objeto Criteria de Propel

Uno de los requisitos presentados durante el segundo día decía que *"cuando el usuario accede a la portada de Jobeet, ve la lista de ofertas de trabajo activas"*.

Sin embargo, ahora mismo se muestran todas las ofertas de trabajo, estén activas o no:

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_job_list = JobeetJobPeer::doSelect(new Criteria());
  }

  // ...
}
```

Una oferta de trabajo activa es aquella que se publicó hace menos de 30 días. El método `doSelect()` toma como argumento un objeto de tipo `Criteria` que describe la consulta que se va a realizar a la base de datos. El código del ejemplo anterior utiliza un objeto `Criteria` vacío, lo que significa que se obtienen todos los registros de la base de datos.

Si queremos obtener sólo las ofertas de trabajo activas, tenemos que reemplazar el código anterior por lo siguiente:

```
public function executeIndex(sfWebRequest $request)
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::CREATED_AT, time() - 86400 * 30,
Criteria::GREATER_THAN);

  $this->jobeet_job_list = JobeetJobPeer::doSelect($criteria);
}
```

El método `Criteria::add()` añade una condición `WHERE` a la sentencia SQL generada. De esta forma podemos limitar el objeto `Criteria` para que sólo seleccione las ofertas de trabajo que se han publicado en los últimos 30 días. El método `add()` permite el uso de

muchos operadores para realizar comparaciones, siendo los más utilizados los que se muestran a continuación:

- `Criteria::EQUAL`
- `Criteria::NOT_EQUAL`
- `Criteria::GREATER_THAN`, `Criteria::GREATER_EQUAL`
- `Criteria::LESS_THAN`, `Criteria::LESS_EQUAL`
- `Criteria::LIKE`, `Criteria::NOT_LIKE`
- `Criteria::CUSTOM`
- `Criteria::IN`, `Criteria::NOT_IN`
- `Criteria::ISNULL`, `Criteria::ISNOTNULL`
- `Criteria::CURRENT_DATE`, `Criteria::CURRENT_TIME`, `Criteria::CURRENT_TIMESTAMP`

6.2. Depurando las sentencias SQL generadas por Propel

Como en las aplicaciones Symfony no escribes las sentencias SQL a mano, Propel tiene en cuenta las diferencias entre los gestores de bases de datos para generar sentencias SQL optimizadas para la base de datos que elegiste durante el tutorial del día 3. Aun así, en ocasiones es necesario ver las sentencias SQL generadas por Propel, por ejemplo para descubrir por qué no funciona una consulta determinada.

En el entorno `dev`, Symfony guarda todas estas sentencias (y mucha otra información) en los archivos de log que se encuentran en el directorio `log/`. Por cada combinación de aplicación y entorno se crea un archivo de log. Por tanto, el archivo en el que tenemos que buscar se llama `frontend_dev.log`:

```
# log/frontend_dev.log
Dec 6 15:47:12 symfony [debug] {sfPropelLogger} exec: SET NAMES 'utf8'
Dec 6 15:47:12 symfony [debug] {sfPropelLogger} prepare: SELECT jobeet_job.ID,
jobeet_job.CATEGORY_ID, jobeet_job.TYPE, jobeet_job.COMPANY, jobeet_job.LOGO,
jobeet_job.URL, jobeet_job.POSITION, jobeet_job.LOCATION,
jobeet_job.DESCRIPTION, jobeet_job.HOW_TO_APPLY, jobeet_job.TOKEN,
jobeet_job.IS_PUBLIC, jobeet_job.CREATED_AT, jobeet_job.UPDATED_AT FROM
''jobeet_job'' WHERE jobeet_job.CREATED_AT>:p1
Dec 6 15:47:12 symfony [debug] {sfPropelLogger} Binding '2008-11-06 15:47:12'
at position :p1 w/ PDO type PDO::PARAM_STR
```

A partir de los mensajes anteriores es inmediato comprobar que Propel ha incluido una condición de tipo `WHERE` para la columna `created_at` (`WHERE jobeet_job.CREATED_AT > :p1`).

Nota

> La cadena de texto `:p1` indica que Propel genera *sentencias preparadas* o *"prepared statements"*. El valor por el que se sustituye `:p1` (en este caso, `2008-11-06 15:47:12`) se pasa durante la ejecución de la sentencia y se le aplica el mecanismo de escape de la base de datos. Utilizar *sentencias preparadas* reduce drásticamente la posibilidad de sufrir ataques de tipo SQL injection (http://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL) .

Aunque toda la información está disponible en los archivos de log, es un poco aburrido alternar entre el navegador, el entorno de desarrollo y los archivos de log cada vez que se quiere probar un cambio. Afortunadamente, gracias a la barra de depuración web de Symfony, toda la información necesaria está disponible directamente dentro del navegador:

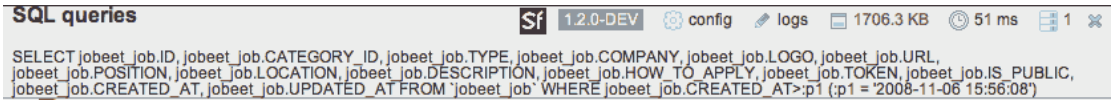

Figura 6.1. Sentencias SQL en la barra de depuración web

6.3. Serializando objetos

Aunque el código anterior funciona correctamente, no es suficiente para cumplir con el requerimiento que establecimos durante el segundo día: *"los usuarios pueden reactivar y extender la validez de la oferta por otros 30 días..."*.

El problema del código anterior es que utiliza el valor de `created_at`, que es la columna que guarda la fecha de creación del objeto. El valor de esta columna no se debería modificar, por lo que no se puede cumplir con el requerimiento establecido.

Si haces memoria, recordarás que el esquema de la base de datos dispone de una columna llamada `expires_at`. Por el momento, esta columna no guarda ningún valor porque no la hemos utilizado en los archivos de datos (*fixtures*). Cuando se crea una nueva oferta de trabajo, el valor de esta columna debe establecerse a un valor equivalente a 30 días después de la fecha actual.

Para modificar un objeto de Propel antes de que se guarde en la base de datos, debes redefinir el método `save()` de la clase del modelo:

```
// lib/model/JobeetJob.php
class JobeetJob extends BaseJobeetJob
{
  public function save(PropelPDO $con = null)
  {
    if ($this->isNew() && !$this->getExpiresAt())
    {
      $now = $this->getCreatedAt() ? $this->getCreatedAt('U') : time();
      $this->setExpiresAt($now + 86400 * 30);
    }

    return parent::save($con);
  }
```

```
  // ...
}
```

El método `isNew()` devuelve `true` cuando el objeto no se ha guardado todavía en la base de datos y `false` en cualquier otro caso.

Ahora ya se puede modificar la acción para que haga uso de la columna `expires_at` en vez de `created_at` al obtener las ofertas de trabajo activas:

```
public function executeIndex(sfWebRequest $request)
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);

  $this->jobeet_job_list = JobeetJobPeer::doSelect($criteria);
}
```

El objeto `Criteria` se restringe para que sólo seleccione las ofertas de trabajo cuya fecha de expiración todavía no se ha cumplido, es decir, las ofertas de trabajo para las que su valor `expires_at` es una fecha futura.

6.4. Profundizando en los archivos de datos

Si vuelves a cargar la página principal de Jobeet no notarás ninguna diferencia, ya que las ofertas de trabajo que se encuentran en la base de datos se insertaron hace pocos días. Por ello, se va a modificar el archivo de datos para añadir una oferta de trabajo expirada:

```
# data/fixtures/020_jobs.yml
JobeetJob:
  # other jobs

  expired_job:
    category_id:  programming
    company:      Sensio Labs
    position:     Web Developer
    location:     Paris, France
    description:  Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    how_to_apply: Send your resume to lorem.ipsum [at] dolor.sit
    is_public:    true
    is_activated: true
    created_at:   2005-12-01
    token:        job_expired
    email:        job@example.com
```

Nota

Debes tener mucho cuidado cuando copias y pegas código en un archivo de datos para no romper la tabulación del archivo. La clave `expired_job` sólo debe contener dos espacios en blanco por delante.

Aunque Propel establece automáticamente el valor de las columnas llamadas `created_at`, se puede redefinir su valor en los archivos de datos, tal y como hemos

hecho en el archivo anterior. Vuelve a insertar los datos de prueba en la base de datos con el siguiente comando y refresca la página principal de Jobeet para comprobar que no se muestra la oferta de trabajo expirada:

```
$ php symfony propel:data-load
```

Si quieres también puedes ejecutar la siguiente consulta para asegurarte de que el método `save()` establece automáticamente el valor de la columna `expires_at` en función del valor de `created_at`:

```
SELECT `position`, `created_at`, `expires_at` FROM `jobeet_job`;
```

6.5. Personalizando la configuración

En el método `JobeetJob::save()` anterior se ha establecido directamente el número de días necesarios para que expire una oferta de trabajo. Seguramente es una buena idea hacer que el número de días sea configurable. El framework Symfony incluye un archivo de configuración llamado `app.yml` que se emplea para establecer las opciones de la aplicación. Este archivo en formato YAML puede contener cualquier información que se necesite para la aplicación:

```
# apps/frontend/config/app.yml
all:
  active_days: 30
```

Desde la aplicación, las opciones del archivo `app.yml` se pueden obtener mediante la clase `sfConfig`:

```
sfConfig::get('app_active_days')
```

El nombre de la opción se ha prefijado con `app_` porque la clase `sfConfig` también permite obtener las opciones de configuración de Symfony, tal y como veremos más adelante.

Después de añadir la opción de configuración, podemos modificar el método `save()` para tenerla en cuenta:

```
public function save(PropelPDO $con = null)
{
  if ($this->isNew() && !$this->getExpiresAt())
  {
    $now = $this->getCreatedAt() ? $this->getCreatedAt('U') : time();
    $this->setExpiresAt($now + 86400 * sfConfig::get('app_active_days'));
  }

  return parent::save($con);
}
```

El archivo de configuración `app.yml` es una buena manera de centralizar todas las opciones de configuración de la aplicación.

6.6. Refactorizando

Una vez más, aunque el código anterior funciona bien, no es correcto del todo. ¿Sabes por qué?

El código que contiene el objeto `Criteria` no debe incluirse en la acción (es decir, en la capa del controlador), ya que pertenece a la capa del modelo. En la arquitectura MVC, el modelo define toda la lógica de negocio y el controlador simplemente realiza llamadas al modelo para obtener los datos. Como se trata de un código que devuelve un listado de ofertas de trabajo, vamos a crear un método llamado `getActiveJobs()` en la clase `JobeetJobPeer`:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public function getActiveJobs()
  {
    $criteria = new Criteria();
    $criteria->add(self::EXPIRES_AT, time(), Criteria::GREATER_THAN);

    return self::doSelect($criteria);
  }
}
```

Ahora el código de la acción puede utilizar este nuevo método para obtener todas las ofertas de trabajo activas.

```
public function executeIndex(sfWebRequest $request)
{
  $this->jobeet_job_list = JobeetJobPeer::getActiveJobs();
}
```

A continuación se indican las ventajas de esta refactorización respecto del código anterior:

- La lógica que se encarga de obtener las ofertas de trabajo se encuentra en el modelo, el sitio al que pertenece.
- El código del controlador ahora es mucho más fácil de leer.
- El método `getActiveJobs()` se puede reutilizar siempre que se necesite, por ejemplo en otra acción.
- Ahora se pueden realizar pruebas unitarias para el código del modelo.

Otra pequeña mejora consiste en ordenar las ofertas de trabajo según el valor de la columna `expires_at`:

```
static public function getActiveJobs()
{
  $criteria = new Criteria();
  $criteria->add(self::EXPIRES_AT, time(), Criteria::GREATER_THAN);
  $criteria->addDescendingOrderByColumn(self::EXPIRES_AT);
```

```
  return self::doSelect($criteria);
}
```

El método `addDescendingOrderByColumn()` añade una condición de tipo `ORDER BY` descendente a la sentencia SQL generada. Si quieres ordenar los registros de forma ascendente, también existe un método llamado `addAscendingOrderByColumn()`.

6.7. Mostrando las categorías en la portada

Otro de los requerimientos que establecimos durante el segundo día era: *"las ofertas se agrupan por categoría y se ordenan por fecha de publicación (primero se muestran los trabajos más recientes)"*.

Hasta ahora no hemos tenido en cuenta la categoría de cada oferta de trabajo, aunque los requerimientos de la aplicación indican que la portada muestra las ofertas de trabajo agrupadas por categoría. En primer lugar debemos obtener todas las categorías que tienen al menos una oferta de trabajo activa.

Abre la clase `JobeetCategoryPeer` y añade el siguiente método llamado `getWithJobs()`:

```
// lib/model/JobeetCategoryPeer.php
class JobeetCategoryPeer extends BaseJobeetCategoryPeer
{
  static public function getWithJobs()
  {
    $criteria = new Criteria();
    $criteria->addJoin(self::ID, JobeetJobPeer::CATEGORY_ID);
    $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);
    $criteria->setDistinct();

    return self::doSelect($criteria);
  }
}
```

El método `Criteria::addJoin()` añade una condición de tipo `JOIN` en la sentencia SQL generada. Por defecto la condición `JOIN` se añade a la condición `WHERE`. Si quieres modificar el tipo de `JOIN`, utiliza uno de los siguientes valores como tercer argumento:`Criteria::LEFT_JOIN`, `Criteria::RIGHT_JOIN` y `Criteria::INNER_JOIN`.

Ahora actualiza la acción `index` para que utilice el nuevo método:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeIndex(sfWebRequest $request)
{
  $this->categories = JobeetCategoryPeer::getWithJobs();
}
```

En la plantilla asociada a la acción ahora tenemos que iterar por todas las categorías para mostrar sus ofertas de trabajo activas:

```
// apps/frontend/modules/job/templates/indexSuccess.php
<?php use_stylesheet('jobs.css') ?>
```

```
<div id="jobs">
  <?php foreach ($categories as $category): ?>
    <div class="category_<?php echo Jobeet::slugify($category->getName()) ?>">
      <div class="category">
        <div class="feed">
          <a href="">Feed</a>
        </div>
        <h1><?php echo $category ?></h1>
      </div>

      <table class="jobs">
        <?php foreach ($category->getActiveJobs() as $i => $job): ?>
          <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
            <td class="location">
              <?php echo $job->getLocation() ?>
            </td>
            <td class="position">
              <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
            </td>
            <td class="company">
              <?php echo $job->getCompany() ?>
            </td>
          </tr>
        <?php endforeach; ?>
      </table>
    </div>
  <?php endforeach; ?>
</div>
```

Nota

La plantilla anterior utiliza `echo $category` para mostrar el nombre de la categoría. ¿Te parece extraño? Teniendo en cuenta que `$category` es un objeto, ¿cómo es posible que `echo` muestre mágicamente el nombre de la categoría? La respuesta se encuentra en el tutorial del día 3, donde definimos métodos mágicos `__toString()` en todas las clases del modelo.

Para que la plantilla anterior funcione correctamente, debemos añadir el método `getActiveJobs()` en la clase `JobeetCategory`:

```
// lib/model/JobeetCategory.php
public function getActiveJobs()
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::CATEGORY_ID, $this->getId());

  return JobeetJobPeer::getActiveJobs($criteria);
}
```

En la llamada al método `add()`, hemos omitido el tecer argumento porque `Criteria::EQUAL` es el valor por defecto.

El método `JobeetCategory::getActiveJobs()` utiliza a su vez el método `JobeetJobPeer::getActiveJobs()` para obtener las ofertas de trabajo activas para la categoría indicada.

Cuando se invoca el método `JobeetJobPeer::getActiveJobs()`, queremos hacer la condición más restrictiva pasándole una categoría. En lugar de pasar el objeto de la categoría actual, hemos decidido pasarle un objeto de tipo `Criteria`, ya que es la mejor forma de encapsular una condición genérica.

Por tanto, el método `getActiveJobs()` tiene que que tenerlo en cuenta y debe fusionar el objeto `Criteria` que se le pasa y su propio `Criteria`. Como `Criteria` es un objeto, el código resultante es muy sencillo:

```
// lib/model/JobeetJobPeer.php
static public function getActiveJobs(Criteria $criteria = null)
{
  if (is_null($criteria))
  {
    $criteria = new Criteria();
  }

  $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);
  $criteria->addDescendingOrderByColumn(self::EXPIRES_AT);

  return self::doSelect($criteria);
}
```

6.8. Limitando los resultados

Un último requerimiento del listado de ofertas de trabajo de la portada es el siguiente: *"para cada categoría sólo se muestran las primeras diez ofertas y el resto se pueden visualizar pulsando sobre el enlace disponible"*.

Limitar el número de resultados es muy sencillo, por lo que sólo debes modificar el código del método `getActiveJobs()` de la siguiente forma:

```
// lib/model/JobeetCategory.php
public function getActiveJobs($max = 10)
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::CATEGORY_ID, $this->getId());
  $criteria->setLimit($max);

  return JobeetJobPeer::getActiveJobs($criteria);
}
```

El límite de la condición `LIMIT` se ha establecido en la propia clase del modelo, pero sería mucho mejor que ese valor fuera configurable. Por tanto, modifica la plantilla para pasar a este método el máximo número de ofertas de trabajo que se obtiene del archivo de configuración `app.yml`:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php foreach
($category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')) as $i =>
$job): ?>
```

Para que el código anterior funcione, no te olvides de añadir la opción de configuración en el archivo `app.yml`:

```
all:
  active_days:          30
  max_jobs_on_homepage: 10
```

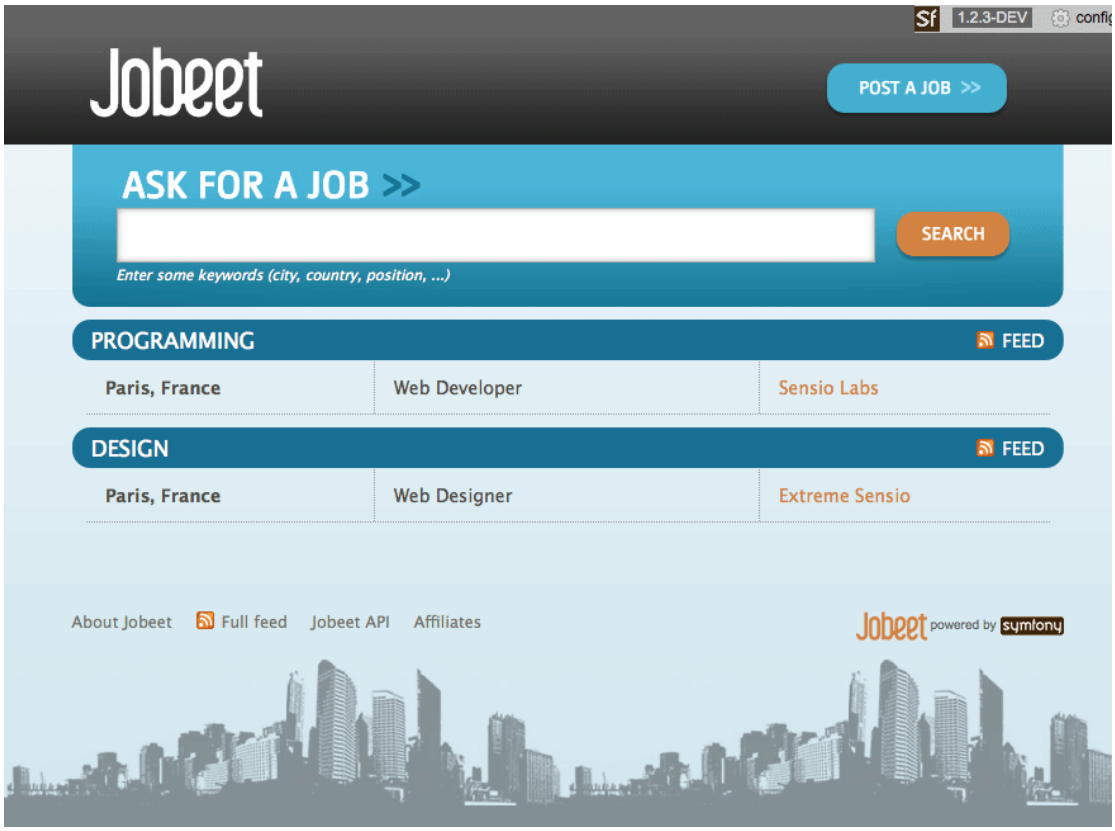

Figura 6.2. Portada organizada por categorías

6.9. Archivos de datos dinámicos

Ahora mismo, salvo que la opción `max_jobs_on_homepage` valga `1`, no vas a notar ninguna diferencia en el listado de ofertas de trabajo de la portada. Lo que necesitamos es crear muchas ofertas de trabajo de pruebas en el archivo de datos. Si crees que debes copiar y pegar 20 veces una oferta de trabajo y después cambiar algunos datos, estás equivocado. Copiar y pegar siempre es una mala solución, incluso en los archivos de datos.

Una de las ventajas de los archivos YAML de Symfony es que pueden contener código PHP que se evalúa antes de procesar el archivo. Abre el archivo de datos `020_jobs.yml` y añade el siguiente código al final del todo:

```
JobeetJob:
# Starts at the beginning of the line (no whitespace before)
<?php for ($i = 100; $i <= 130; $i++): ?>
  job_<?php echo $i ?>:
    category_id:  programming
    company:      Company <?php echo $i."\n" ?>
```

```
    position:     Web Developer
    location:     Paris, France
    description:  Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    how_to_apply: |
      Send your resume to lorem.ipsum [at] company_<?php echo $i ?>.sit
    is_public:    true
    is_activated: true
    token:        job_<?php echo $i."\n" ?>
    email:        job@example.com

<?php endfor; ?>
```

Como siempre que se trabaja con archivos YAML, debes tener mucho cuidado con la tabulación de la información. Cuando añadas código PHP a un archivo YAML, ten en cuenta estos trucos sencillos:

- Las sentencias `<?php ?>` siempre deben empezar una línea o ser parte de un valor.
- Si la sentencia `<?php ?>` finaliza la línea, se debe incluir explícitamente un carácter de nueva línea ("\n").

Ahora ya puedes volver a cargar los archivos de datos mediante la tarea `propel:data-load` para comprobar si en la categoría `Programming` de la portada solamente se muestran `10` ofertas de trabajo. En la siguiente imagen hemos cambiado el número máximo de ofertas de trabajo a `5` para que la imagen no sea demasiado grande:

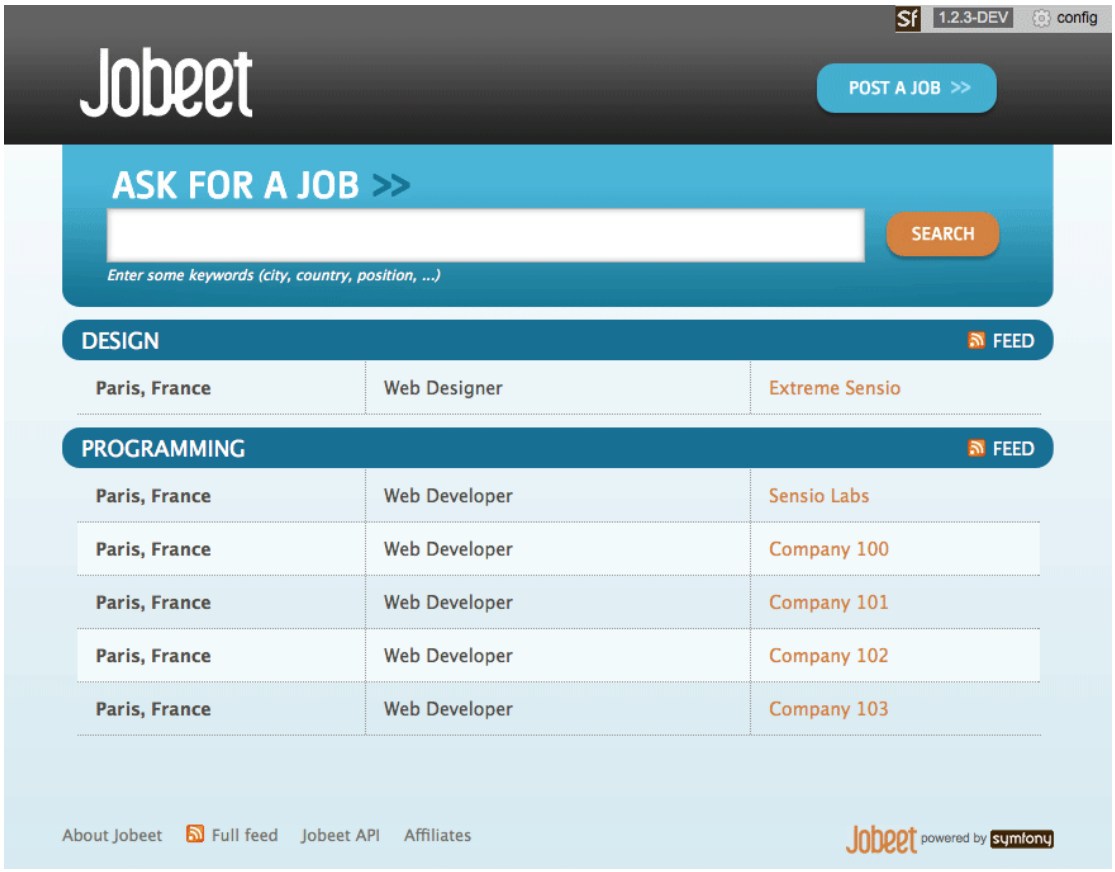

Figura 6.3. Paginación en los listados de portada

6.10. Restringiendo el acceso a la página de una oferta de trabajo

Cuando una oferta de trabajo expira, ya no debe ser posible visualizar su información, aunque se conozca su URL. Prueba a acceder a la URL de la oferta de trabajo que hemos insertado como expirada (debes reemplazar el valor de la variable `id` por el valor del `id` de tu base de datos, que puedes obtener con la consulta `SELECT `id`, `token` FROM `jobeet_job` WHERE `expires_at` < NOW()`):

```
/frontend_dev.php/job/sensio-labs/paris-france/ID/web-developer-expired
```

La aplicación no debería mostrar los detalles de la oferta de trabajo, sino que debería reenviar al usuario a una página de error 404. Pero, ¿cómo podemos hacerlo si la oferta de trabajo se obtiene automáticamente en la ruta?

Las rutas de tipo `sfPropelRoute` utilizan por defecto el método `doSelectOne()` para obtener un objeto, pero se puede utilizar otro método indicándolo en la opción `method_for_criteria` de la configuración de la ruta:

```
# apps/frontend/config/routing.yml
job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfPropelRoute
  options:
    model: JobeetJob
    type:  object
    method_for_criteria: doSelectActive
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [GET]
```

El método `doSelectActive()` recibe como argumento el objeto `Criteria` construido por la ruta:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public function doSelectActive(Criteria $criteria)
  {
    $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);

    return self::doSelectOne($criteria);
  }

  // ...
}
```

Si intentas acceder ahora a la página de una oferta de trabajo expirada, serás redirigido a una página de error 404.

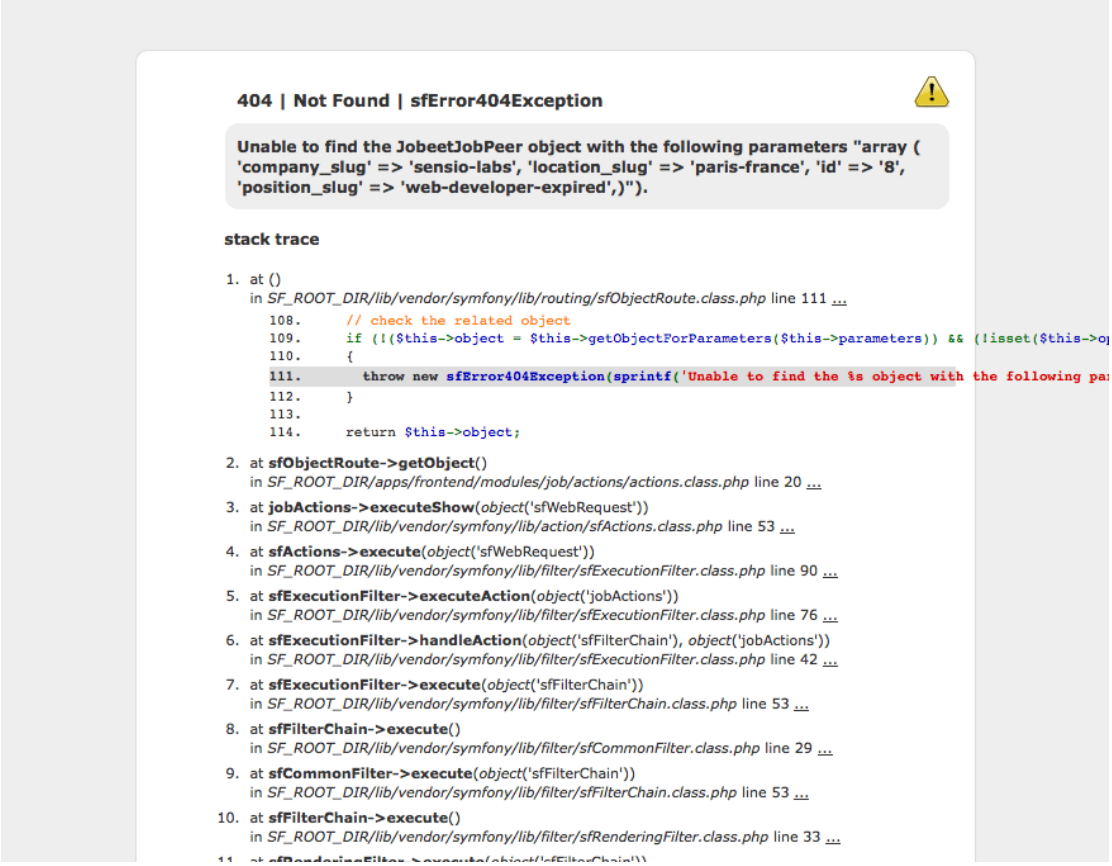

Figura 6.4. Error 404 que se muestra al acceder a la página de una oferta de trabajo expirada

6.11. Enlazando a la página de la categoría

A continuación vamos a crear la página de cada categoría y vamos a añadir en la página principal un enlace a la página de cada categoría.

No obstante, como hoy no hemos trabajado mucho y todavía nos queda tiempo, vamos a dejarlo como ejercicio para que demuestres todo lo que ya sabes. Mañana publicaremos la solución a este ejercicio.

6.12. Nos vemos mañana

No te olvides de completar este ejercicio en tu proyecto Jobeet local. Consulta todo lo que necesites la documentación de la API (http://www.symfony-project.org/api/1_2/) y el resto de documentación de Symfony 1.2 (http://www.symfony-project.org/doc/1_2/). Mañana nos vemos con nuestra solución al ejercicio planteado.

Capítulo 7. Trabajando con la página de cada categoría

Ayer profundizamos en varios aspectos de Symfony: realizar consultas con Propel, los archivos de datos o *fixtures*, el sistema de enrutamiento, la depuración y la configuración personalizada. Además, ayer te propusimos un reto en forma de ejercicio.

Esperamos que hayáis trabajado por vuestra cuenta para crear la página de cada categoría, ya que de esa forma el tutorial de hoy será mucho más provechoso para ti.

Así que vamos a explicar una posible solución al ejercicio de ayer.

7.1. La ruta de la categoría

En primer lugar, debemos crear una nueva ruta para que las páginas de las categorías tengan URL *limpias*. Añade la siguiente ruta al principio del todo del archivo `routing.yml`:

```
# apps/frontend/config/routing.yml
category:
  url:      /category/:slug
  class:    sfPropelRoute
  param:    { module: category, action: show }
  options:  { model: JobeetCategory, type: object }
```

Sugerencia

Siempre que vas a añadir una nueva característica en la aplicación, es una buena práctica pensar primero en su URL y después crear la ruta asociada. Además, esta práctica es obligatoria si has borrado las rutas por defecto de Symfony.

Las rutas pueden utilizar como parámetro cualquier columna de su objeto asociado. Las rutas también pueden emplear cualquier otro valor para el que exista un método accesor de tipo `get()` en la clase del objeto. Como `slug` no es una columna de la tabla `category`, tenemos que añadir un método accesor en `JobeetCategory` para que la ruta anterior pueda funcionar:

```
// lib/model/JobeetCategory.php
public function getSlug()
{
  return Jobeet::slugify($this->getName());
}
```

7.2. El enlace a la página de la categoría

A continuación, edita la plantilla `indexSuccess.php` del módulo `job` para añadir el enlace a la página de la categoría:

```
<!-- some HTML code -->

        <h1><?php echo link_to($category, 'category', $category) ?></h1>

<!-- some HTML code -->

      </table>

      <?php if (($count = $category->countActiveJobs() -
sfConfig::get('app_max_jobs_on_homepage')) > 0): ?>
        <div class="more_jobs">
          and <?php echo link_to($count, 'category', $category) ?>
          more...
        </div>
      <?php endif; ?>
    </div>
  <?php endforeach; ?>
</div>
```

El enlace a la página de la categoría sólo se muestra cuando existen más de 10 ofertas de trabajo en esa misma categoría. El enlace muestra el número de ofertas de trabajo adicionales que existen, sin contar las 10 que se muestran en la portada. Para que el código de la plantilla anterior funcione correctamente, debemos añadir el método `countActiveJobs()` en `JobeetCategory`:

```
// lib/model/JobeetCategory.php
public function countActiveJobs()
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::CATEGORY_ID, $this->getId());

  return JobeetJobPeer::countActiveJobs($criteria);
}
```

Además, el método `countActiveJobs()` utiliza un método `countActiveJobs()` que todavía no existe en la clase `JobeetJobPeer`. Reemplaza el contenido del archivo `JobeetJobPeer.php` por el siguiente código:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public function getActiveJobs(Criteria $criteria = null)
  {
    return self::doSelect(self::addActiveJobsCriteria($criteria));
  }

  static public function countActiveJobs(Criteria $criteria = null)
  {
    return self::doCount(self::addActiveJobsCriteria($criteria));
  }

  static public function addActiveJobsCriteria(Criteria $criteria = null)
  {
    if (is_null($criteria))
```

```
    {
      $criteria = new Criteria();
    }

    $criteria->add(self::EXPIRES_AT, time(), Criteria::GREATER_THAN);
    $criteria->addDescendingOrderByColumn(self::CREATED_AT);

    return $criteria;
  }

  static public function doSelectActive(Criteria $criteria)
  {
    return self::doSelectOne(self::addActiveJobsCriteria($criteria));
  }

}
```

Como habrás observado, hemos refactorizado todo el código de `JobeetJobPeer` para utilizar un nuevo método compartido llamado `addActiveJobsCriteria()`, de forma que el código de la clase siga los principios de DRY (Don't Repeat Yourself) (http://es.wikipedia.org/wiki/No_te_repitas) .

Sugerencia

La primera vez que reutilizas una parte de código, es suficiente con copiarla y pegarla. No obstante, si necesitas ese mismo trozo de código otra vez, es necesario que refactorices las apariciones de ese código y las conviertas en un método o función compartida.

En el método `countActiveJobs()` anterior, en vez de utilizar `doSelect()` y después contar el número de resultados, hemos utilizado directamente el método `doCount()` que es mucho más rápido.

Como acabas de comprobar, hemos tenido que modificar un montón de archivos para añadir una sola característica sencilla. No obstante, cada vez que hemos añadido código, lo hemos insertado en la capa correcta (modelo, vista, controlador) y también hemos conseguido que el código sea fácilmente reutilizable. Además, hemos aprovechado estos cambios para refactorizar parte del código existente. Todo este proceso es el flujo normal de trabajo cuando desarrollas un proyecto con Symfony.

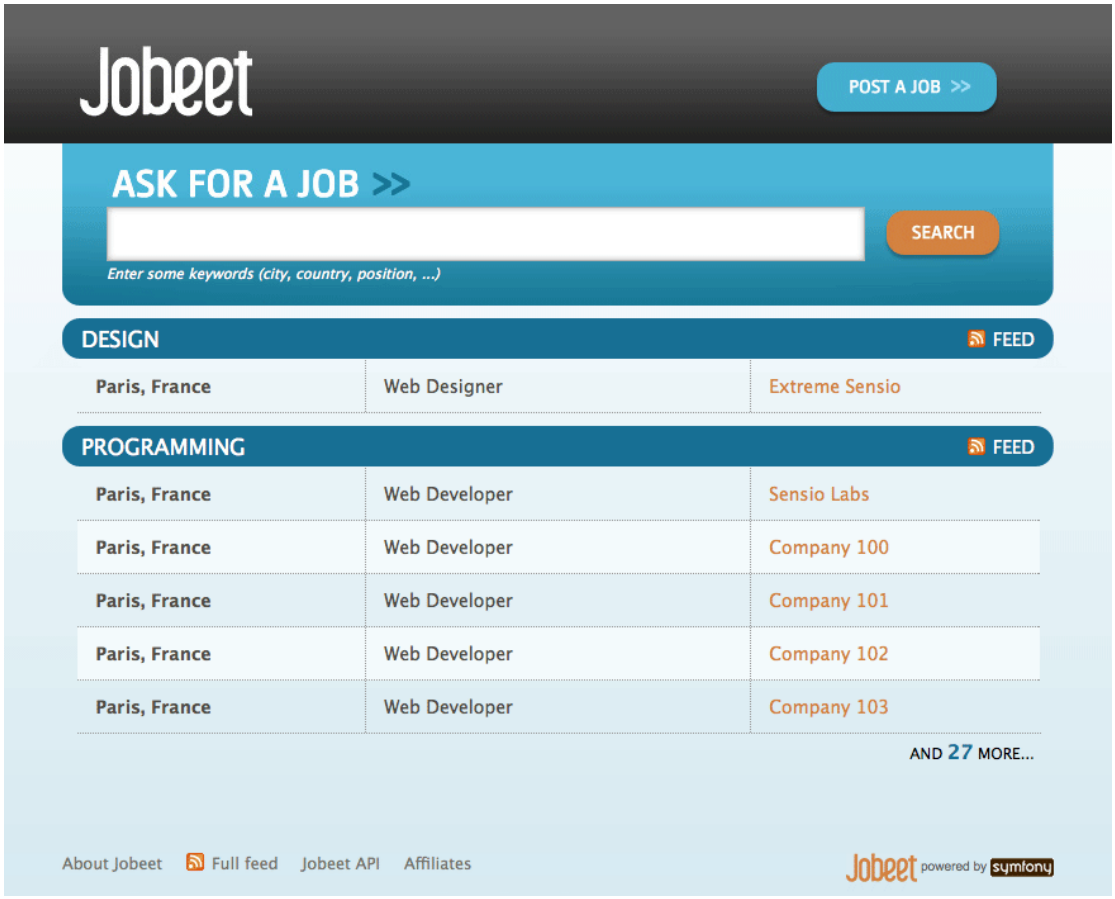

Figura 7.1. Portada de la aplicación

7.3. Creando el módulo de las categorías

El siguiente paso consiste en crear el módulo `category`:

```
$ php symfony generate:module frontend category
```

Al crear el módulo seguramente has utilizado la tarea `propel:generate-module`. Aunque no es incorrecto, como no vamos a utilizar el 90% del código que genera automáticamente esa tarea, vamos a utilizar en su lugar la tarea `generate:module`, que crea un módulo vacío.

> **Sugerencia**
>
> ¿Por qué no hemos añadido simplemente una acción llamada `category` en el módulo `job`? Podríamos haberlo hecho, pero como el principal elemento relacionado con la página de una categoría es la propia categoría, es mucho más lógico crear un módulo específico para las categorías.

Cuando se accede a la página de una categoría, la ruta llamada `category` debe obtener la categoría asociada con el valor de la variable `slug` de la petición. No obstante, como el `slug` no se guarda en la base de datos y como no se puede deducir el nombre de la categoría a partir del `slug`, es imposible obtener la categoría asociada a un `slug`.

7.4. Actualizando la base de datos

Debido a los problemas explicados en la sección anterior, debemos añadir una columna llamada `slug` en la tabla `category`:

```
# config/schema.yml
propel:
  jobeet_category:
    id:           ~
    name:         { type: varchar(255), required: true }
    slug:         { type: varchar(255), required: true, index: unique }
```

Ahora que `slug` es una columna auténtica de la tabla, puedes eliminar el método `getSlug()` de la clase `JobeetCategory`.

Cada vez que se modifica el nombre de una categoría, es necesario calcular el nuevo valor de su `slug` y guardarlo en la base de datos. Para ello, puedes redefinir el método `setName()`:

```
// lib/model/JobeetCategory.php
public function setName($name)
{
  parent::setName($name);

  $this->setSlug(Jobeet::slugify($name));
}
```

Ejecuta la tarea `propel:build-all-load` para volver a generar todas las tablas de la base de datos y para cargar los datos de prueba de los archivos de datos:

```
$ php symfony propel:build-all-load --no-confirmation
```

Ahora ya tenemos todo listo para crear el nuevo método `executeShow()`. Reemplaza el contenido del archivo de acciones del módulo `category` por el siguiente código:

```
// apps/frontend/modules/category/actions/actions.class.php
class categoryActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->category = $this->getRoute()->getObject();
  }
}
```

Nota

Como hemos eliminado el método `executeIndex()` generado automáticamente, también puedes borrar la plantilla `indexSuccess.php` asociada, que se encuentra en el archivo `apps/frontend/modules/category/templates/indexSuccess.php`.

Por último, crea la plantilla `showSuccess.php`:

```
// apps/frontend/modules/category/templates/showSuccess.php
<?php use_stylesheet('jobs.css') ?>
```

```
<?php slot('title', sprintf('Jobs in the %s category', $category->getName())) ?>

<div class="category">
  <div class="feed">
    <a href="">Feed</a>
  </div>
  <h1><?php echo $category ?></h1>
</div>

<table class="jobs">
  <?php foreach ($category->getActiveJobs() as $i => $job): ?>
    <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
      <td class="location">
        <?php echo $job->getLocation() ?>
      </td>
      <td class="position">
        <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
      </td>
      <td class="company">
        <?php echo $job->getCompany() ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>
```

7.5. Elementos parciales

Si te fijas en el código de la plantilla anterior, verás que hemos copiado y pegado la etiqueta `<table>` que muestra el listado de ofertas de trabajo directamente de la plantilla `indexSuccess.php`. Como hemos dicho muchas veces, copiar y pegar siempre es mala idea, por lo que ha llegado el momento de aprender otro concepto importante de Symfony.

Cuando quieres reutilizar un trozo de una plantilla, tienes que crear un **elemento parcial**. Los elementos parciales son trozos de código de plantilla que se pueden utilizar en varias plantillas. Técnicamente, un elemento parcial es otra plantilla con la única diferencia de que su nombre empieza obligatoriamente por un guión bajo (_).

Crea el siguiente archivo `_list.php`:

```
// apps/frontend/modules/job/templates/_list.php
<table class="jobs">
  <?php foreach ($jobs as $i => $job): ?>
    <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
      <td class="location">
        <?php echo $job->getLocation() ?>
      </td>
      <td class="position">
        <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
      </td>
      <td class="company">
        <?php echo $job->getCompany() ?>
```

```
      </td>
    </tr>
  <?php endforeach; ?>
</table>
```

Una vez creado, puedes incluir el elemento parcial en la plantilla mediante el helper `include_partial()`:

```
<?php include_partial('job/list', array('jobs' => $jobs)) ?>
```

El primer argumento de `include_partial()` es el nombre del elemento parcial, formado por el nombre del módulo, seguido por `/` y terminado por el nombre del elemento parcial sin el guión bajo inicial `_`. El segundo argumento es un array con las variables que se pasan al elemento parcial.

Nota

¿Por qué no se utiliza simplemente la función `include()` de PHP en vez del helper `include_partial()`? La principal diferencia entre los dos es que el helper `include_partial()` incluye soporte para la cache.

Ahora ya puedes reemplazar el código HTML de las tablas de las dos plantillas por la llamada al helper `include_partial()`:

```
// in apps/frontend/modules/job/templates/indexSuccess.php
<?php include_partial('job/list', array('jobs' =>
$category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')))) ?>

// in apps/frontend/modules/category/templates/showSuccess.php
<?php include_partial('job/list', array('jobs' => $category->getActiveJobs()))
?>
```

7.6. Paginación

Uno de los requisitos establecidos durante el día dos decía que *"el listado de ofertas de trabajo de la página de cada categoría incluye una paginación con 20 ofertas por página"*.

La paginación de los listados de objetos Propel se realiza mediante una clase específica llamada sfPropelPager (http://www.symfony-project.org/api/1_2/sfPropelPager) . En la acción `category`, en vez de pasar a la plantilla `showSuccess` los objetos que representan las ofertas de trabajo, pasamos un objeto paginador:

```
// apps/frontend/modules/category/actions/actions.class.php
public function executeShow(sfWebRequest $request)
{
  $this->category = $this->getRoute()->getObject();

  $this->pager = new sfPropelPager(
    'JobeetJob',
    sfConfig::get('app_max_jobs_on_category')
  );
  $this->pager->setCriteria($this->category->getActiveJobsCriteria());
  $this->pager->setPage($request->getParameter('page', 1));
```

```
  $this->pager->init();
}
```

Sugerencia

El método `sfRequest::getParameter()` admite un segundo parámetro que indica el valor por defecto cuando el primer argumento no existe. En el código de la acción anterior, si el parámetro page de la petición no existe, el método `getParameter()` devuelve 1.

El constructor de `sfPropelPager` toma como argumentos la clase del modelo y el máximo número de elementos por página. Por tanto, es necesario que añadas este último valor al archivo de configuración:

```
# apps/frontend/config/app.yml
all:
  active_days:          30
  max_jobs_on_homepage: 10
  max_jobs_on_category: 20
```

Por su parte, el método `sfPropelPager::setCriteria()` toma como primer argumento el objeto `Criteria` que se debe utilizar para obtener los registros de la base de datos.

Añade el método `getActiveJobsCriteria()`:

```
// lib/model/JobeetCategory.php
public function getActiveJobsCriteria()
{
  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::CATEGORY_ID, $this->getId());

  return JobeetJobPeer::addActiveJobsCriteria($criteria);
}
```

Ahora que hemos definido el método `getActiveJobsCriteria()`, podemos refactorizar los otros métodos de `JobeetCategory` para que lo utilicen:

```
// lib/model/JobeetCategory.php
public function getActiveJobs($max = 10)
{
  $criteria = $this->getActiveJobsCriteria();
  $criteria->setLimit($max);

  return JobeetJobPeer::doSelect($criteria);
}

public function countActiveJobs()
{
  $criteria = $this->getActiveJobsCriteria();

  return JobeetJobPeer::doCount($criteria);
}
```

Por último, actualiza la plantilla:

```
<!-- apps/frontend/modules/category/templates/showSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<?php slot('title', sprintf('Jobs in the %s category', $category->getName())) ?>

<div class="category">
  <div class="feed">
    <a href="">Feed</a>
  </div>
  <h1><?php echo $category ?></h1>
</div>

<?php include_partial('job/list', array('jobs' => $pager->getResults())) ?>

<?php if ($pager->haveToPaginate()): ?>
  <div class="pagination">
    <a href="<?php echo url_for('category', $category) ?>?page=1">
      <img src="/images/first.png" alt="First page" />
    </a>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo
$pager->getPreviousPage() ?>">
      <img src="/images/previous.png" alt="Previous page" title="Previous page"
/>
    </a>

    <?php foreach ($pager->getLinks() as $page): ?>
      <?php if ($page == $pager->getPage()): ?>
        <?php echo $page ?>
      <?php else: ?>
        <a href="<?php echo url_for('category', $category) ?>?page=<?php echo
$page ?>"><?php echo $page ?></a>
      <?php endif; ?>
    <?php endforeach; ?>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo
$pager->getNextPage() ?>">
      <img src="/images/next.png" alt="Next page" title="Next page" />
    </a>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo
$pager->getLastPage() ?>">
      <img src="/images/last.png" alt="Last page" title="Last page" />
    </a>
  </div>
<?php endif; ?>

<div class="pagination_desc">
  <strong><?php echo $pager->getNbResults() ?></strong> jobs in this category

  <?php if ($pager->haveToPaginate()): ?>
    - page <strong><?php echo $pager->getPage() ?>/<?php echo
$pager->getLastPage() ?></strong>
  <?php endif; ?>
</div>
```

La mayoría del código anterior se encarga de enlazar otras páginas del paginador. A continuación se muestran otros métodos de `sfPropelPager` que utiliza esta plantilla:

- `getResults()`: devuelve un array con los objetos Propel de la página actual
- `getNbResults()`: devuelve el número total de resultados
- `haveToPaginate()`: devuelve `true` si existe más de una página
- `getLinks()`: devuelve una lista de enlaces a todas las páginas del paginador
- `getPage()`: devuelve el número de la página actual
- `getPreviousPage()`: devuelve el número de la página anterior
- `getNextPage()`: devuelve el número de la página siguiente
- `getLastPage()`: devuelve el número de la última página

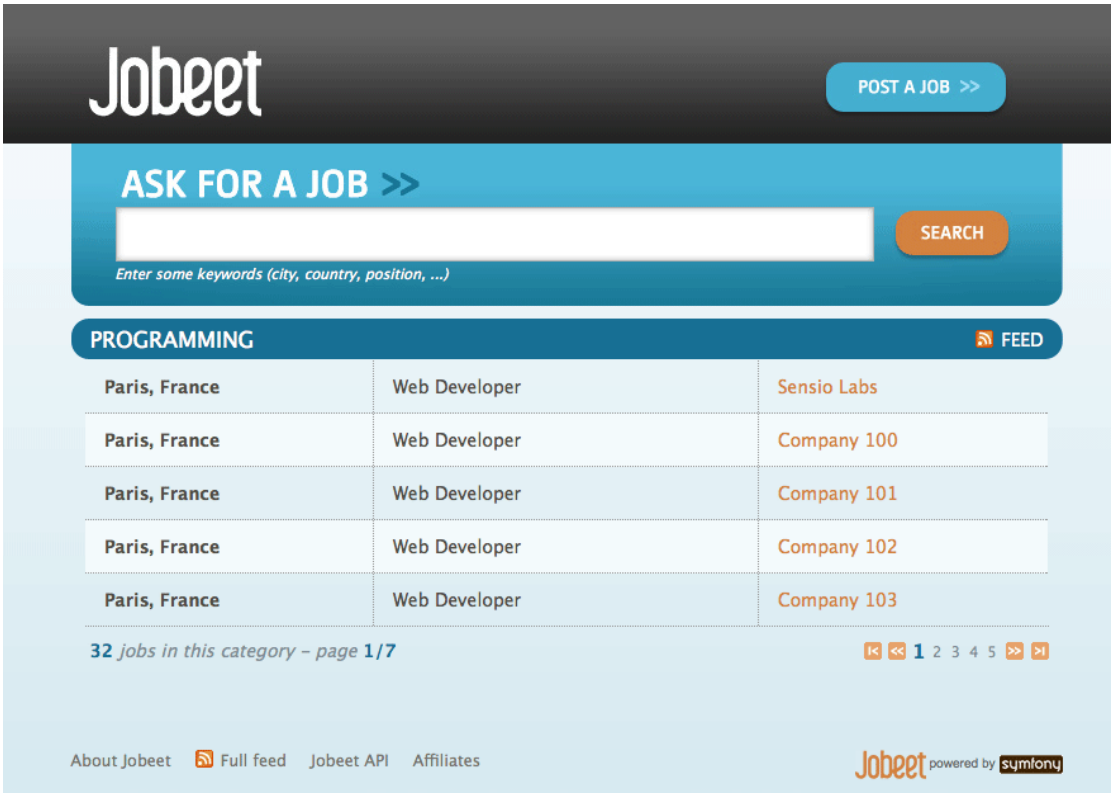

Figura 7.2. Paginación en el listado de ofertas de trabajo de una categoría

7.7. Nos vemos mañana

Si ayer hiciste el ejercicio por tu cuenta y sientes que hoy no has aprendido mucho, eso significa que ya comprendes la filosofía de trabajo de Symfony. El proceso para añadir una nueva característica en las aplicaciones Symfony siempre es idéntico: piensa en las URL de esa nueva característica, crea las acciones adecuadas, actualiza el modelo y crea varias plantillas. Y si mientras haces todo eso, aplicas algunas buenas prácticas del desarrollo web, te vas a convertir en un maestro de Symfony en muy poco tiempo.

Mañana comenzamos una nueva semana con Jobeet y para celebrarlo hablaremos de un tema completamente nuevo: las pruebas unitarias y funcionales.

Capítulo 8. Pruebas unitarias

Los dos últimos días los hemos dedicado a repasar los conceptos de Symfony que aprendimos durante los cinco primeros días, a retocar algunas funcionalidades de Jobeet y a añadir algunas nuevas características.

Hoy vamos a hablar de algo completamente diferente: las pruebas automáticas. Además, como se trata de un tema muy complejo, le vamos a dedicar dos días enteros para explicar hasta el último detalle.

8.1. Pruebas en Symfony

En Symfony se pueden crear dos tipos diferentes de pruebas automáticas: las **pruebas unitarias** y las **pruebas funcionales**.

Las pruebas unitarias comprueban que todas las funciones y todos los métodos funcionan correctamente. Cada una de las pruebas unitarias debe ser completamente independiente de las demás.

Por otra parte, las pruebas funcionales verifican que la aplicación funciona correctamente en su conjunto.

Las pruebas en Symfony se guardan en el directorio `test/` del proyecto. El directorio contiene a su vez dos subdirectorios, uno para las pruebas unitarias (`test/unit/`) y otro para las pruebas funcionales (`test/functional/`).

Hoy vamos a explicar las pruebas unitarias y mañana hablaremos de las pruebas funcionales.

8.2. Pruebas unitarias

Una de las buenas prácticas del desarrollo web que más cuesta a los programadores consiste en escribir pruebas unitarias. Como los programadores web normalmente no están acostumbrados a probar bien su trabajo, les surgen muchas dudas: ¿tengo que escribir las pruebas antes de programar la nueva funcionalidad? ¿qué debo probar? ¿las pruebas tienen que probar hasta los casos más extremos? ¿cómo puedo asegurarme de que estoy probando todo bien? Por suerte, la primera pregunta que se hacen es mucho más fácil: ¿por dónde empiezo?

Aunque somos completamente partidarios de las pruebas, la propuesta de Symfony es más pragmática: creemos que es mejor tener unas pocas pruebas a no tener ninguna. ¿Tienes un montón de código para el que no has creado pruebas? No pasa nada, ya que para disfrutar de las ventajas de las pruebas automáticas no es necesario disponer de pruebas para todas las funcionalidades de la aplicación.

Nuestra propuesta es que vayas añadiendo pruebas a medida que encuentres y soluciones errores en tu aplicación. Con el paso del tiempo tu código no sólo será mucho

mejor, sino que cada vez será mayor el porcentaje de la aplicación que está cubierto por pruebas (técnicamente, este porcentaje se conoce como *code coverage*). Utilizar esta filosofía de trabajo, hará que ganes confianza al escribir las pruebas. En poco tiempo estarás escribiendo las pruebas para las nuevas funcionalidades de la aplicación y más tarde te convertirás en un apasionado de las pruebas.

El principal problema de las librerías para crear pruebas es que son bastante difíciles de aprender a manejar. Por este motivo Symfony incluye su propia librería para pruebas llamada **lime** y que simplifica al máximo la creación de pruebas.

Nota

Aunque en este tutorial vamos a explicar detalladamente la librería **lime**, puedes utilizar cualquier otra librería de pruebas, como por ejemplo la excelente librería PHPUnit (http://www.phpunit.de/) .

8.3. El framework de pruebas lime

Todas las pruebas unitarias escritas para el framework `lime` empiezan con las mismas líneas de código:

```
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(1, new lime_output_color());
```

La primera línea incluye el archivo `unit.php`, que se encarga de realizar la inicialización. Después se crea un objeto de tipo `lime_test` y se le pasa como argumento el número de pruebas que se quieren realizar.

Nota

Indicar el número de pruebas esperadas permite que `lime` muestre un error en caso de que no se hayan realizado suficientes pruebas, como por ejemplo cuando una determinada prueba provoca un error fatal de PHP.

Las pruebas consisten en invocar un método o una función, pasarles una serie de argumentos y comparar su respuesta con la respuesta esperada. Esta última comparación es la que permite determinar si una prueba se ha superado o ha fallado.

Para facilitar las comparaciones, el objeto `lime_test` incluye varios métodos útiles:

Método	Descripción
`ok($condicion)`	Si la condición que se indica es `true`, la prueba tiene éxito
`is($valor1, $valor2)`	Compara dos valores y la prueba pasa si los dos son iguales (==)
`isnt($valor1, $valor2)`	Compara dos valores y la prueba pasa si no son iguales
`like($cadena, $expresionRegular)`	Prueba que una cadena cumpla con el patrón de una expresión regular

`unlike($cadena, $expresionRegular)`	Prueba que una cadena no cumpla con el patrón de una expresión regular
`is_deeply($array1, $array2)`	Comprueba que dos arrays tengan los mismos valores

Sugerencia

Quizás te preguntas por qué motivo `lime` define tantos métodos si todas las pruebas se podrían escribir utilizando solamente el método `ok()`. Las ventajas de utilizar diferentes métodos residen en la posibilidad de mostrar mensajes de error más explícitos cuando falla la prueba y una mejora de la facilidad de lectura de las pruebas.

El objeto `lime_test` también incluye otros métodos útiles para pruebas:

Método	Descrición
`fail()`	Provoca que la prueba siempre falle (es útil para probar las excepciones)
`pass()`	Provoca que la prueba siempre se pase (es útil para probar las excepciones)
`skip($mensaje, $numeroPruebas)`	Cuenta como si fueran $numeroPruebas pruebas (es útil para las pruebas condicionales)
`todo()`	Cuenta como si fuera una prueba (es útil para las pruebas que todavía no se han escrito)

Por último, el método `comment($mensaje)` muestra un comentario o mensaje pero no realiza ninguna prueba.

8.4. Ejecutando pruebas unitarias

Todas las pruebas unitarias se guardan en el directorio `test/unit/`. Además, Symfony utiliza la convención de nombrar las pruebas mediante el nombre de la clase que prueban seguido de la palabra `Test`. Aunque puedes organizar los archivos del directorio `test/unit/` tal como quieras, te recomendamos que sigas la estructura del directorio `lib/`.

Para ilustrar el uso de las pruebas unitarias, vamos a probar la clase `Jobeet`. Crea el archivo `test/unit/JobeetTest.php` y copia el siguiente código en su interior:

```
// test/unit/JobeetTest.php
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(1, new lime_output_color());
$t->pass('This test always passes.');
```

Para lanzar las pruebas puedes ejecutar directamente el archivo:

```
$ php test/unit/JobeetTest.php
```

También puedes hacer uso de la tarea `test:unit`:

```
$ php symfony test:unit Jobeet
```

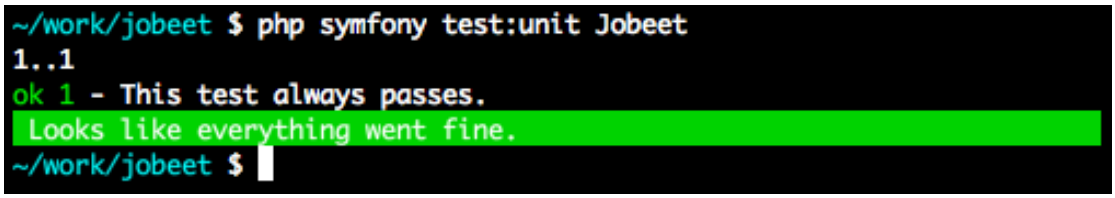

Figura 8.1. Ejecutando pruebas en la línea de comandos

Nota

Desafortunadamente, la línea de comandos de Windows no es capaz de resaltar las líneas de los resultados de las pruebas en color rojo o color verde.

8.5. Probando el método slugify

Vamos a comenzar a adentrarnos en el mundo de las pruebas unitarias escribiendo pruebas para el método `Jobeet::slugify()`.

El método `slugify()` lo creamos en el tutorial del día 5 para *limpiar* una cadena de texto de forma que su contenido se pueda incluir como parte de una URL. La transformación que se realiza es bastante sencilla, ya que consiste en convertir todos los caracteres que no sean ASCII en un guión medio (-) y pasar la cadena de texto a minúsculas:

Cadena original	Cadena transformada
Sensio Labs	`sensio-labs`
Paris, France	`paris-france`

Para probar el método `slugify`, reemplaza el contenido de la prueba unitaria por el siguiente código:

```
// test/unit/JobeetTest.php
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(6, new lime_output_color());

$t->is(Jobeet::slugify('Sensio'), 'sensio');
$t->is(Jobeet::slugify('sensio labs'), 'sensio-labs');
$t->is(Jobeet::slugify('sensio   labs'), 'sensio-labs');
$t->is(Jobeet::slugify('paris,france'), 'paris-france');
$t->is(Jobeet::slugify('  sensio'), 'sensio');
$t->is(Jobeet::slugify('sensio  '), 'sensio');
```

Si te fijas en las pruebas que acabamos de escribir, verás que cada línea sólo prueba una cosa. Este es uno de los conceptos fundamentales que siempre tienes que tener presente. Prueba una sola cosa cada vez.

Ahora ya puedes volver a ejecutar las pruebas. Si todas las pruebas pasan correctamente, que es lo que esperamos en este ejemplo, verás una barra de color verde. Por el contrario, si alguna prueba falla verás una barra de color rojo indicando que

algunas pruebas han fallado y que tienes que arreglarlas (recuerda que los colores no se ven en sistemas operativos tipo Windows).

Figura 8.2. Pruebas del método slugify

Si una prueba falla, se muestran mensajes de ayuda con información sobre el motivo por el que ha fallado. Sin embargo, si tienes cientos de pruebas en un archivo, es bastante complicado identificar la característica exacta que ha fallado.

Por ese motivo, todas los métodos de pruebas de `lime` admiten como último argumento una cadena de texto que se utiliza como descripción de la prueba. Incluir este argumento es muy útil porque te obliga a describir exactamente lo que estás probando. Además, esta descripción puede servir como documentación del comportamiento esperado por el método. Por lo tanto, vamos a añadir algunos mensajes en las pruebas del método `slugify`:

```
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(6, new lime_output_color());

$t->comment('::slugify()');
$t->is(Jobeet::slugify('Sensio'), 'sensio', '::slugify() converts all
characters to lower case');
$t->is(Jobeet::slugify('sensio labs'), 'sensio-labs', '::slugify() replaces a
white space by a -');
$t->is(Jobeet::slugify('sensio   labs'), 'sensio-labs', '::slugify() replaces
several white spaces by a single -');
$t->is(Jobeet::slugify('  sensio'), 'sensio', '::slugify() removes - at the
beginning of a string');
$t->is(Jobeet::slugify('sensio  '), 'sensio', '::slugify() removes - at the end
of a string');
$t->is(Jobeet::slugify('paris,france'), 'paris-france', '::slugify() replaces
non-ASCII characters by a -');
```

```
~/work/jobeet $ php symfony test:unit Jobeet
1..6
# ::slugify()
ok 1 - ::slugify() converts all characters to lower case
ok 2 - ::slugify() replaces a white space by a -
ok 3 - ::slugify() replaces several white spaces by a single -
ok 4 - ::slugify() replaces non-ASCII characters by a -
ok 5 - ::slugify() removes - at the beginning of a string
ok 6 - ::slugify() removes - at the end of a string
Looks like everything went fine.
~/work/jobeet $
```

Figura 8.3. Pruebas del método slugify con mensajes descriptivos

La descripción de cada prueba también es muy útil cuando intentas descubrir qué tienes que probar. Como habrás observado, las descripciones de las pruebas siempre siguen el mismo patrón: son frases que describen cómo se debe comportar el método y siempre empiezan con el nombre del método que se prueba.

Code coverage

Cuando escribes pruebas es muy fácil olvidarse de probar algunas partes del código.

Symfony incluye una tarea llamada `test:coverage` que te permite comprobar que todo tu código está bien probado. Para comprobar el porcentaje de código que está cubierto por las pruebas (llamado *code coverage*) indica como primer argumento el nombre de un archivo o directorio con pruebas y como segundo argumento el nombre de un archivo o directorio con código.

```
$ php symfony test:coverage test/unit/JobeetTest.php lib/Jobeet.class.php
```

Si quieres ver las líneas de código exactas que no están probadas por tus pruebas, utiliza la opción `--detailed`:

```
$ php symfony test:coverage --detailed test/unit/JobeetTest.php lib/
Jobeet.class.php
```

Cuando esta tarea indica que tu código está completamente probado, debes tener en cuenta que sólo significa que todas las líneas de tu código se han probado, pero no significa que se han probado todos los casos extremos que se deberían probar en cada método.

Como la tarea `test:coverage` hace uso de XDebug para obtener su información, en primer lugar debes instalar y activar XDebug.

8.6. Añadiendo pruebas para las nuevas características

El *slug* de una cadena de texto vacía es otra cadena de texto vacía. Si pruebas el comportamiento anterior, la prueba pasará correctamente. El problema es que no parece una buena idea añadir una cadena de texto vacía como parte de la URL. Por tanto, vamos a modificar el método `slugify()` para que devuelva la cadena de texto `n-a` (del inglés *not available*, "no disponible") cuando se le pase una cadena de texto vacía.

Si quieres puedes escribir primero la prueba y después actualizar el método, aunque también puedes hacer lo contrario. Hacerlo de una u otra forma es una cuestión de gusto

personal, aunque escribir primero la prueba te da más confianza de que lo que programas es exactamente lo que habías planeado:

```
$t->is(Jobeet::slugify(''), 'n-a', '::slugify() converts the empty string to
n-a');
```

Si vuelves a ejecutar las pruebas, verás que se muestra la barra de color rojo. En caso contrario, o ya has añadido esa funcionalidad al método o esta prueba no está probando lo que debería probar.

A continuación edita la clase `Jobeet` y añade la siguiente condición al principio del todo:

```
// lib/Jobeet.class.php
static public function slugify($text)
{
  if (empty($text))
  {
    return 'n-a';
  }

  // ...
}
```

La prueba ahora sí que debe pasar satisfactoriamente y se debe mostrar la barra verde, aunque sólo si te has acordado de actualizar el plan de pruebas. Si no lo has hecho, verás un mensaje de error que indica que habías planeado seis pruebas y has realizado una más. Actualizar el número de pruebas de cada archivo es muy importante, ya que permite comprobar si el script ha finalizado antes de realizar todas las pruebas.

8.7. Añadir pruebas al corregir un error

Imagina que ya has publicado la aplicación web y uno de tus usuarios te informa de un error bastante extraño: al pinchar los enlaces de algunas ofertas de trabajo se muestra una página de error 404. Después de investigar el error, descubres que esas ofertas de trabajo que están fallando tienen vacíos los campos de la empresa, el puesto de trabajo y/o la localidad. ¿Cómo puede suceder esto? Sigues investigando y ves que las columnas de la base de datos no están vacías.

Después de pensar un poco más, por fin descubres la causa del error. Si una cadena de texto sólo contiene caracteres que no son ASCII, el método `slugify()` la convierte en una cadena de texto vacía. Como estás tan contento de haber descubierto el error, editas la clase `Jobeet` y corriges el error directamente. Lo que acabas de hacer no es una buena idea, ya que en primer lugar deberías añadir una prueba unitaria:

```
$t->is(Jobeet::slugify(' - '), 'n-a', '::slugify() converts a string that only
contains non-ASCII characters to n-a');
```

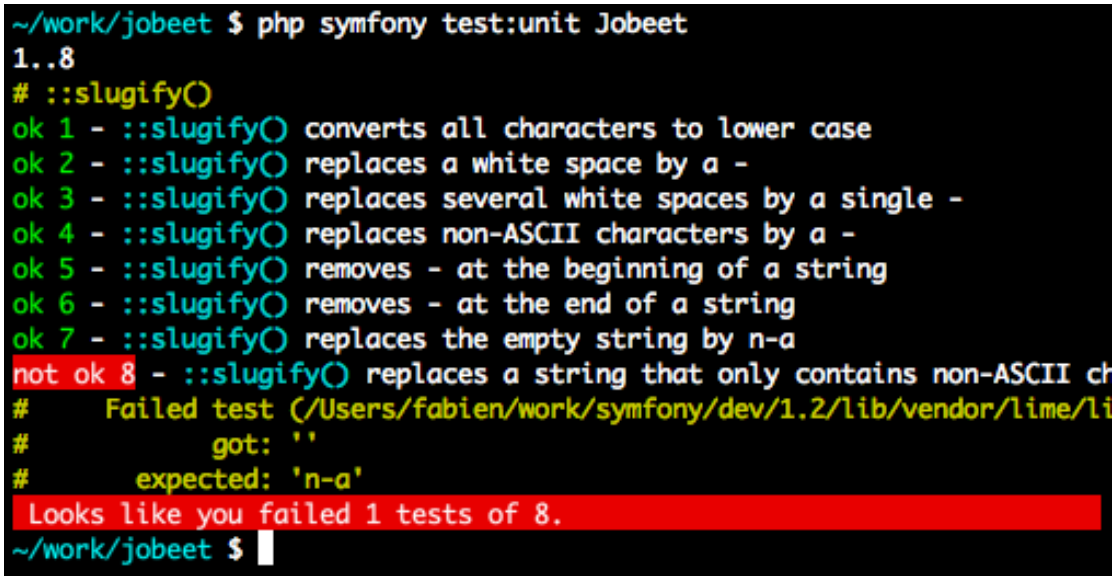

Figura 8.4. Fallo descubierto en el método slugify()

Después de comprobar que se produce un error al ejecutar la prueba unitaria, edita la clase `Jobeet` y mueve la comprobación de si una cadena es vacía al final del método:

```
static public function slugify($text)
{
  // ...

  if (empty($text))
  {
    return 'n-a';
  }

  return $text;
}
```

La nueva prueba unitaria ahora sí que pasa, al igual que siguen pasando todas las anteriores. Aunque el código tenía un 100% de *code coverage*, el método `slugify()` tenía un error.

Obviamente no puedes pensar en todos los posibles casos extremos cuando creas pruebas unitarias. Sin embargo, cuando descubres un nuevo caso extremo, debes escribir una prueba unitaria antes de intentar solucionarlo. Además, trabajar de esta manera hace que el código de tu aplicación sea cada vez mejor, lo que es una buena consecuencia de las pruebas unitarias.

Mejorando el método slugify

Seguramente ya sabes que Symfony ha sido creado por una empresa francesa, por lo que vamos a añadir una prueba para una palabra en francés que contiene un acento:

```
$t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web',
'::slugify() removes accents');
```

La prueba va a fallar, ya que el método `slugify()` en vez de reemplazar la letra é por e, la ha reemplazado por un guión medio (-). Para solucionar este problema tenemos que usar un proceso conocido como *transliteración*. Si tu instalación de PHP cuenta con `iconv`, esta función se encarga de todo. Reemplaza el código del método `slugify()` por lo siguiente:

```
// code derived from http://php.vrana.cz/
vytvoreni-pratelskeho-url.php
static public function slugify($text)
{
  // replace non letter or digits by -
  $text = preg_replace('~[^\\pL\d]+~u', '-', $text);

  // trim
  $text = trim($text, '-');

  // transliterate
  if (function_exists('iconv'))
  {
    $text = iconv('utf-8', 'us-ascii//TRANSLIT', $text);
  }

  // lowercase
  $text = strtolower($text);

  // remove unwanted characters
  $text = preg_replace('~[^-\w]+~', '', $text);

  if (empty($text))
  {
    return 'n-a';
  }

  return $text;
}
```

No te olvides de guardar todos tus archivos de PHP con la codificación `UTF-8`, ya que esta es la codificación por defecto de Symfony y también es la codificación que utiliza `iconv` para realizar la transliteración de las cadenas de texto.

Por último, modifica la prueba para que sólo se realice si la función `iconv` está disponible:

```
if (function_exists('iconv'))
{
  $t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web',
'::slugify() removes accents');
}
else
{
  $t->skip('::slugify() removes accents - iconv not installed');
}
```

8.8. Pruebas unitarias para Propel

8.8.1. Configuración de la base de datos

Escribir pruebas unitarias para la clase de un modelo es un poco más complicado porque requiere una conexión con la base de datos. Aunque ya disponemos de la conexión que configuramos para el entorno de desarrollo, es una buena práctica crear una conexión con la base de datos específica para las pruebas.

Durante el tutorial del primer día explicamos el concepto de entornos de ejecución como una forma sencilla de modificar las opciones con las que se ejecuta una aplicación. Por defecto, las pruebas se ejecutan en un entorno llamado `test`, por lo que vamos a configurar una base de datos diferente para este entorno `test`:

```
$ php symfony configure:database --env=test
"mysql:host=localhost;dbname=jobeet_test" root ConTraSenA
```

La opción `env` le indica a la tarea `configure:database` que esta conexión con la base de datos sólo se emplea en el entorno `test`. Cuando utilizamos esta tarea en el tutorial del día 3, no pasamos ninguna opción `env`, por lo que la configuración se realizó para todos los entornos.

Nota

Si sientes curiosidad, abre el archivo de configuración `config/databases.yml` para ver lo fácil que es en Symfony modificar la configuración en función del entorno.

Después de configurar la base de datos, podemos inicializarla mediante la tarea `propel:insert-sql`:

```
$ mysqladmin -uroot -pConTraSenA create jobeet_test
$ php symfony propel:insert-sql --env=test
```

Así funciona la configuración en Symfony

Durante el tutorial del día 4 vimos cómo se pueden definir en diferentes niveles las opciones de los archivos de configuración.

Estas opciones también pueden depender del entorno de ejecución. De hecho, esto es posible en la mayor parte de los archivos de configuración que hemos utilizado hasta el momento: `databases.yml`, `app.yml`, `view.yml` y `settings.yml`. En todos estos archivos de configuración, la clave de primer nivel en los archivos YAML indica el entorno para el que se aplican las opciones, siendo `all` la clave que indica que esas opciones se aplican a todos los entornos:

```
# config/databases.yml
dev:
  propel:
    class: sfPropelDatabase
    param:
      classname: DebugPDO

test:
```

```
  propel:
    class: sfPropelDatabase
    param:
      classname: DebugPDO
      dsn: 'mysql:host=localhost;dbname=jobeet_test'

all:
  propel:
    class: sfPropelDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=jobeet'
      username: root
      password: null
```

8.8.2. Datos para pruebas

Ahora que ya tenemos una base de datos sólo para pruebas, tenemos que llenarla con datos de prueba. Durante el día 3 aprendimos a utilizar la tarea `propel:data-load`, pero en las pruebas es necesario volver a cargar los datos cada vez que ejecutamos las pruebas para conocer el estado inicial de la base de datos. La tarea `propel:data-load` utiliza internamente la clase sfPropelData (http://www.symfony-project.org/api/1_2/sfPropelData) para cargar los datos:

```
$loader = new sfPropelData();
$loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Nota

El objeto `sfConfig` se puede utilizar para obtener la ruta completa hasta un subdirectorio del proyecto. Utilizando este método se puede modificar la estructura de directorios por defecto de Symfony.

El método `loadData()` acepta como primer argumento el nombre de un directorio o un archivo. Este método también admite un array de directorios y/o archivos.

Los días anteriores ya creamos algunos datos de pruebas que guardamos en el directorio `data/fixtures/`. Los archivos de datos para pruebas los vamos a guardar en el directorio `test/fixtures/`. Estos archivos de datos los va a utilizar Propel para las pruebas unitarias y funcionales.

Por el momento, copia los archivos del directorio `data/fixtures/` al directorio `test/fixtures/`.

8.8.3. Probando JobeetJob

A continuación vamos a crear pruebas unitarias para la clase del modelo `JobeetJob`.

Como todas nuestras pruebas unitarias relacionadas con Propel empiezan con las mismas líneas de código, crea un archivo llamado `propel.php` en el directorio `bootstrap/` de las pruebas y que contenga el siguiente código:

```
// test/bootstrap/propel.php
include(dirname(__FILE__).'/unit.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'test', true);

new sfDatabaseManager($configuration);

$loader = new sfPropelData();
$loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

El script anterior es bastante sencillo de entender:

- Como sucede en los controladores frontales, inicializamos un objeto de tipo configuración para el entorno `test`:

```
$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'test', true);
```

- Creamos un gestor de bases de datos e inicializamos la conexión Propel cargando el archivo de configuración `databases.yml`.

```
new sfDatabaseManager($configuration);
```

- Cargamos los datos de prueba mediante `sfPropelData`:

```
$loader = new sfPropelData();
$loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Nota

Propel sólo se conecta con la base de datos si existen sentencias SQL pendientes de ejecutar.

Ahora que ya tenemos todo preparado, podemos empezar a probar la clase `JobeetJob`.

En primer lugar, crea el archivo `JobeetJobTest.php` en `test/unit/model`:

```
// test/unit/model/JobeetJobTest.php
include(dirname(__FILE__).'/../../bootstrap/propel.php');

$t = new lime_test(1, new lime_output_color());
```

A continuación, creamos una prueba unitaria para el método `getCompanySlug()`:

```
$t->comment('->getCompanySlug()');
$job = JobeetJobPeer::doSelectOne(new Criteria());
$t->is($job->getCompanySlug(), Jobeet::slugify($job->getCompany()),
'->getCompanySlug() return the slug for the company');
```

Como puedes observar en el código anterior, sólo estamos probando el método `getCompanySlug()` y no si el *slug* generado es correcto o no, porque eso ya lo hemos probado en otras pruebas.

Crear una prueba para el método `save()` es un poco más complicado:

```
$t->comment('->save()');
$job = create_job();
```

```
$job->save();
$expiresAt = date('Y-m-d', time() + 86400 * sfConfig::get('app_active_days'));
$t->is($job->getExpiresAt('Y-m-d'), $expiresAt, '->save() updates expires_at if
not set');

$job = create_job(array('expires_at' => '2008-08-08'));
$job->save();
$t->is($job->getExpiresAt('Y-m-d'), '2008-08-08', '->save() does not update
expires_at if set');

function create_job($defaults = array())
{
  static $category = null;

  if (is_null($category))
  {
    $category = JobeetCategoryPeer::doSelectOne(new Criteria());
  }

  $job = new JobeetJob();
  $job->fromArray(array_merge(array(
    'category_id'  => $category->getId(),
    'company'      => 'Sensio Labs',
    'position'     => 'Senior Tester',
    'location'     => 'Paris, France',
    'description'  => 'Testing is fun',
    'how_to_apply' => 'Send e-Mail',
    'email'        => 'job@example.com',
    'token'        => rand(1111, 9999),
    'is_activated' => true,
  ), $defaults), BasePeer::TYPE_FIELDNAME);

  return $job;
}
```

Nota

Cada vez que añades nuevas pruebas, no te olvides de actualizar en el constructor del método `lime_test` el número de pruebas que esperas realizar. En el archivo `JobeetJobTest` tienes que reemplazar el valor `1` original por `3`.

8.8.4. Probando otras clases de Propel

Ahora ya puedes probar otras clases de Propel. Como poco a poco te estás acostumbrando a crear pruebas unitarias, no será muy complicado escribir esas pruebas.

8.9. Conjuntos de pruebas unitarias

La tarea `test:unit` también se puede utilizar para lanzar todas las pruebas unitarias de un proyecto:

```
$ php symfony test:unit
```

Esta tarea muestra si ha pasado o ha fallado cada uno de los archivos de pruebas:

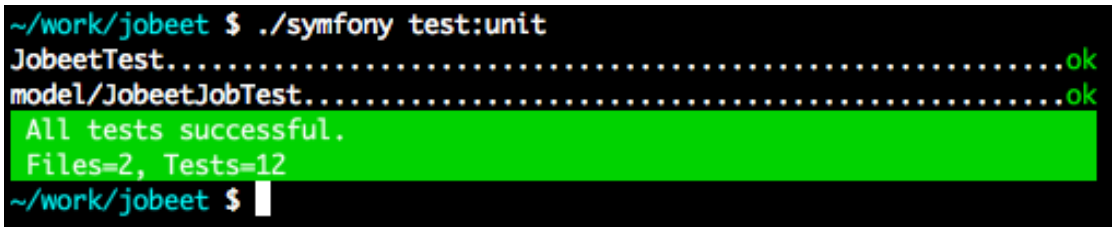

Figura 8.5. Conjuntos de pruebas unitarias

Sugerencia

Si la tarea `test:unit` muestra un estado `dubious` en un archivo, eso significa que el script ha finalizado su ejecución antes de llegar al final. Si quieres averiguar la causa exacta del error, ejecuta ese archivo de pruebas de forma individual.

8.10. Nos vemos mañana

Aunque probar bien las aplicaciones es algo muy importante, estoy seguro de que algunos de vosotros habéis pensado en saltaros este tutorial. Me alegro de que no lo hayáis hecho.

Aprender a programar con Symfony es mucho más que aprender todas las características del framework, ya que también se trata de aprender su filosofía de trabajo y seguir las buenas prácticas que recomienda. Y las pruebas son una de esas buenas prácticas. Más tarde o más temprando las pruebas unitarias te van a ayudar mucho en tus desarrollos. Las pruebas aumentan la confianza en tu código y te permiten refactorizar la aplicación sin miedo a introducir nuevos errores. Las pruebas unitarias son como un vigilante que te avisa en cuanto rompes algo. De hecho, el propio framework Symfony tiene más de 9000 pruebas.

Mañana vamos a escribir algunas pruebas funcionales para los módulos `job` y `category`. Hasta entonces, no te olvides de escribir algunas pruebas unitarias para las clases del modelo de Jobeet.

Capítulo 9. Pruebas funcionales

En la lección de ayer vimos cómo añadir pruebas unitarias a las clases de Jobeet utilizando la librería de pruebas `lime` que incluye Symfony.

Hoy vamos a escribir pruebas funcionales para las características que ya hemos desarrollado en los módulos `job` y `category`.

9.1. Pruebas funcionales

Las pruebas funcionales son la mejor forma de probar tu aplicación de extremo a extremo: desde la petición realizada por un navegador hasta la respuesta enviada por el servidor. Las pruebas funcionales prueban todas las capas de la aplicación: el sistema de enrutamiento, el modelo, las acciones y las plantillas. En realidad, son muy similares a lo que tu mismo haces manualmente: cada vez que añades o modificas una acción, la pruebas en el navegador para comprobar que todo funciona bien al pulsar sobre los enlaces y botones y que todos los elementos se muestran correctamente en la página. En otras palabras, lo que haces es probar un escenario correspondiente al caso de uso que acabas de implementar en la aplicación.

Como el proceso anterior es manual, no sólo es muy aburrido, sino que es muy propenso a cometer errores. Cada vez que realizas un cambio en el código, tienes que volver a probar todos los escenarios para asegurarte que los cambios no han roto nada en la aplicación. Obviamente trabajar así es una locura. Las pruebas funcionales de Symfony permiten describir de forma sencilla los escenarios de la aplicación. Una vez definidos, los escenarios se pueden ejecutar automáticamente una y otra vez de forma que simule el comportamiento de un usuario con su navegador. Al igual que las pruebas unitarias, las pruebas funcionales te dan la confianza y tranquilidad de saber que lo que estás programando no va a romper nada en la aplicación.

Nota

El subframework de pruebas funcionales no reemplaza a herramientas como Selenium (http://selenium.seleniumhq.org/) . La herramienta Selenium se ejecuta directamente en un navegador y se emplea para automatizar las pruebas en muchos navegadores y sistemas operativos diferentes, por lo que también es capaz de probar el código JavaScript de la aplicación.

9.2. La clase sfBrowser

En Symfony, las pruebas funcionales se realizan mediante un navegador especial creado con la clase sfBrowser (http://www.symfony-project.org/api/1_2/sfBrowser) . Esta clase actúa como un navegador completamente adaptado a tu aplicación y conectado directamente a ella, de forma que no necesitas un servidor web. La clase `sfBrowser` te da

acceso a todos los objetos de Symfony antes y después de cada petición, permitiendo la introspección de los objetos para realizar las comprobaciones automáticamente.

La clase `sfBrowser` incluye métodos que simulan la navegación que se realiza en cualquier navegador tradicional:

Método	**Descripción**
`get()`	Obtiene una URL
`post()`	Envía datos a una URL
`call()`	Realiza una llamada a una URL (se utiliza para los métodos PUT y DELETE)
`back()`	Vuelve a la página anterior almacenada en el historial
`forward()`	Va a la página siguiente almacenada en el historial
`reload()`	Recarga la página actual
`click()`	Pulsa sobre un enlace o un botón
`select()`	Selecciona un *radiobutton* o un *checkbox*
`deselect()`	Deselecciona un *radiobutton* o un *checkbox*
`restart()`	Reinicia el navegador

A continuación se muestran algunos ejemplos de uso de los métodos de `sfBrowser`:

```
$browser = new sfBrowser();

$browser->
  get('/')->
  click('Design')->
  get('/category/programming?page=2')->
  get('/category/programming', array('page' => 2))->
  post('search', array('keywords' => 'php'))
;
```

La clase `sfBrowser` también incluye métodos para configurar el comportamiento del navegador:

Método	**Descripción**
`setHttpHeader()`	Establece el valor de una cabecera HTTP
`setAuth()`	Establece las credenciales de la autenticación básica
`setCookie()`	Establece una cookie
`removeCookie()`	Elimina una cookie
`clearCookie()`	Borra todas las cookies actuales
`followRedirect()`	Sigue una redirección

9.3. La clase sfTestFunctional

Aunque ya disponemos de un navegador, todavía no es posible la introspección de los objetos de Symfony para realizar las pruebas y comprobaciones. Esta introspección se podría realizar con `lime` y los métodos `getResponse()` y `getRequest()` de `sfBrowser`, pero Symfony permite hacerlo de otra forma mejor.

Los métodos para pruebas se incluyen en otra clase llamada sfTestFunctional (http://www.symfony-project.org/api/1_2/sfTestFunctional) y que utiliza como argumento de su constructor un objeto de tipo `sfBrowser`. La clase `sfTestFunctional` delega las pruebas en objetos de tipo **tester**. Symfony ya incluye varios *testers*, pero también puedes crear todos los *testers* propios que necesites.

Como se vio en la lección de ayer, las pruebas funcionales se almacenan en el directorio `test/functional/`. Las pruebas de Jobeet se almacenan en el subdirectorio `test/functional/frontend/`, ya que cada aplicación utiliza su propio subdirectorio. Este directorio ya contiene dos archivos llamados `categoryActionsTest.php` y `jobActionsTest.php`, ya que todas las tareas que generan módulos de forma automática crean un archivo muy básico de pruebas funcionales:

```
// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new sfTestFunctional(new sfBrowser());

$browser->
  get('/category/index')->

  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'index')->
  end()->

  with('response')->begin()->
    isStatusCode(200)->
    checkElement('body', '!/This is a temporary page/')->
  end()
;
```

Al principio, el código anterior puede parecerte un poco extraño. El motivo es que los métodos de `sfBrowser` y `sfTestFunctional` siempre devuelven el objeto `$this` para permitir lo que se conoce con el nombre de *interfaz fluida* (http://es.wikipedia.org/wiki/Interface_fluida) . De esta forma, es posible encadenar varios métodos para mejorar la facilidad de lectura del código. El código anterior es equivalente a:

```
// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new sfTestFunctional(new sfBrowser());

$browser->get('/category/index');
```

```
$browser->with('request')->begin();
$browser->isParameter('module', 'category');
$browser->isParameter('action', 'index');
$browser->end();

$browser->with('response')->begin();
$browser->isStatusCode(200);
$browser->checkElement('body', '!/This is a temporary page/');
$browser->end();
```

Las pruebas se ejecutan dentro de un contexto de bloque de *tester*. Los contextos de bloque de *testers* siempre empiezan por `with('NOMBRE_DEL_TESTER')->begin()` y terminan con `end()`:

```
$browser->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'index')->
  end()
;
```

El código anterior prueba que el parámetro `module` de la petición sea igual a `category` y el parámetro `action` sea igual a `index`.

Sugerencia

Si sólo vas a utilizar un método del *tester*, no es necesario que crees un bloque: `with('request')->isParameter('module', 'category')`

9.3.1. El tester request

El *tester* `request` incluye métodos para realizar la introspección y probar los objetos de tipo `sfWebRequest`:

Método	Descripción
`isParameter()`	Comprueba el valor de un parámetro de la petición
`isFormat()`	Comprueba el formato de la petición
`isMethod()`	Comrpueba el método utilizado
`hasCookie()`	Comprueba si la petición incluye una cookie con el nombre indicado
`isCookie()`	Comprueba el valor de una cookie

9.3.2. El tester response

También existe un *tester* `response` que incluye los métodos equivalente para los objetos de tipo `sfWebResponse`:

Método	Descripción
`checkElement()`	Comprueba si un selector CSS sobre la respuesta cumple el criterio indicado

`isHeader()`	Comprueba el valor de una cabecera
`isStatusCode()`	Comprueba el el código de estado de la respuesta
`isRedirected()`	Comprueba si la respuesta actual es en realidad una redirección

Nota

Durante los próximos días explicaremos muchos otros testers (http://www.symfony-project.org/api/1_2/test) utilizados para formularios, usuarios cache, etc.

9.4. Ejecutando pruebas funcionales

Al igual que sucede en las pruebas unitarias, puedes ejecutar las pruebas funcionales directamente a partir de un archivo de pruebas:

```
$ php test/functional/frontend/categoryActionsTest.php
```

También puedes utilizar la tarea `test:functional`:

```
$ php symfony test:functional frontend categoryActions
```

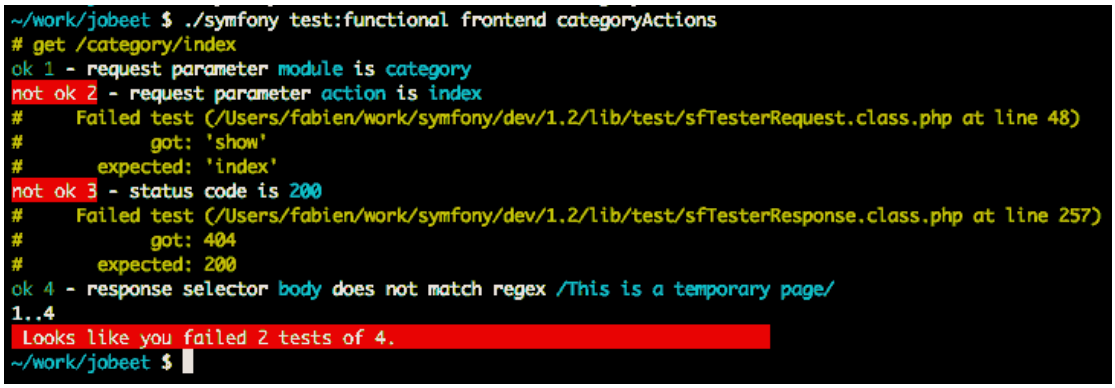

Figura 9.1. Ejecutando pruebas en la línea de comandos

9.5. Datos de prueba

De la misma forma que para las pruebas unitarias de Propel, cada vez que ejecutamos una prueba funcional tenemos que volver a cargar los datos de prueba. Por lo tanto, podemos reutilizar el código que escribimos ayer:

```
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$loader = new sfPropelData();
$loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Cargar los datos en una prueba funcional es un poco más sencillo que hacerlo en las pruebas unitarias, ya que en este caso la base de datos ya ha sido inicializada mediante el script de inicialización de la prueba.

Como sucedía en las pruebas unitarias, no vamos a copiar y pegar continuamente el trozo de código anterior en cada archivo de pruebas, sino que vamos a crear nuestra propia clase para pruebas funcionales que herede de la clase `sfTestFunctional`:

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    $loader = new sfPropelData();
    $loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');

    return $this;
  }
}
```

9.6. Escribiendo pruebas funcionales

Crear las pruebas funcionales es similar a ejecutar un determinado escenario en el navegador. En nuestro caso, las historias que escribimos para el tutorial del día 2 ya describen todos los escenarios que debemos probar.

En primer lugar vamos a probar la página principal de Jobeet mediante el archivo de pruebas `jobActionsTest.php`. Reemplaza su contenido por el siguiente código:

9.6.1. Las ofertas de trabajo expiradas no se muestran en el listado

```
// test/functional/frontend/jobActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->info('1 - The homepage')->
  get('/')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'index')->
  end()->
  with('response')->begin()->
    info('  1.1 - Expired jobs are not listed')->
    checkElement('.jobs td.position:contains("expired")', false)->
  end()
;
```

Como sucede en `lime`, puedes utilizar el método `info()` para mostrar mensajes informativos y hacer que la salida del programa sea más fácil de leer. Para comprobar que no se muestran ofertas de trabajo expiradas, comprobamos que el selector CSS `.jobs td.position:contains("expired")` no encuentra ningún elemento dentro del contenido HTML de la respuesta (recuerda que en los archivos de datos que utilizamos, la única oferta de trabajo expirada contiene el valor `expired` en el campo `position`). Si el

segundo argumento del método `checkElement()` es un valor booleano, el método prueba si existen nodos que cumplan con ese selector CSS.

Sugerencia

El método `checkElement()` es capaz de interpretar correctamente la mayoría de selectores CSS3 válidos.

9.6.2. Sólo se muestran N ofertas de trabajo en el listado de cada categoría

Aañade el siguiente código al final del archivo de pruebas:

```
// test/functional/frontend/jobActionsTest.php
$max = sfConfig::get('app_max_jobs_on_homepage');

$browser->info('1 - The homepage')->
  get('/')->
  info(sprintf('  1.2 - Only %s jobs are listed for a category', $max))->
  with('response')->
    checkElement('.category_programming tr', $max)
;
```

Si el segundo argumento del método `checkElement()` es un número entero, el método prueba si existen N nodos que cumplan con ese selector CSS.

9.6.3. Las categorías muestran un enlace a la página de categoría sólo si tienen demasiadas ofertas de trabajo

```
// test/functional/frontend/jobActionsTest.php
$browser->info('1 - The homepage')->
  get('/')->
  info('  1.3 - A category has a link to the category page only if too many
jobs')->
  with('response')->begin()->
    checkElement('.category_design .more_jobs', false)->
    checkElement('.category_programming .more_jobs')->
  end()
;
```

En este caso comprobamos que no se muestre un enlace llamado *"more jobs"* en la categoría `design` (es decir, que no exista `.category_design .more_jobs`) y que se muestre un enlace llamado *"more jobs"* en la categoría `programming` (es decir, que exista `.category_programming .more_jobs`).

9.6.4. Las ofertas de trabajo se ordenan cronológicamente

```
// most recent job in the programming category
$criteria = new Criteria();
$criteria->add(JobeetCategoryPeer::SLUG, 'programming');
$category = JobeetCategoryPeer::doSelectOne($criteria);

$criteria = new Criteria();
```

```
$criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);
$criteria->add(JobeetJobPeer::CATEGORY_ID, $category->getId());
$criteria->addDescendingOrderByColumn(JobeetJobPeer::CREATED_AT);

$job = JobeetJobPeer::doSelectOne($criteria);

$browser->info('1 - The homepage')->
  get('/')->
  info('  1.4 - Jobs are sorted by date')->
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]',
$job->getId()))->
  end()
;
```

Para probar que las ofertas de trabajo se ordenan cronológicamente, comprobamos que la primera oferta de trabajo del listado de la portada es la oferta que esperamos. Por tanto, debemos comprobar que la URL contiene el valor que esperamos para la clave primaria. Además, como la clave primaria puede cambiar de una ejecución a otra, en primer lugar debemos obtener el objeto Propel de la base de datos.

Aunque la prueba anterior ya funciona correctamente, vamos a refactorizar su código para poder reutilizar en otras pruebas la lógica que obtiene la primera oferta de trabajo de la categoría `programming`. Como se trata de un código específico para pruebas, en este caso no vamos a moverlo a la capa del modelo, sino que vamos a colocarlo en la clase `JobeetTestFunctional` que hemos creado anteriormente. De esta forma, esta clase actúa como una clase de pruebas funcionales específicas para el dominio de Jobeet.

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function getMostRecentProgrammingJob()
  {
    // most recent job in the programming category
    $criteria = new Criteria();
    $criteria->add(JobeetCategoryPeer::SLUG, 'programming');
    $category = JobeetCategoryPeer::doSelectOne($criteria);

    $criteria = new Criteria();
    $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);
    $criteria->add(JobeetJobPeer::CATEGORY_ID, $category->getId());
    $criteria->addDescendingOrderByColumn(JobeetJobPeer::CREATED_AT);

    return JobeetJobPeer::doSelectOne($criteria);
  }

  // ...
}
```

Ahora puedes reemplazar el código de la prueba anterior por el siguiente código:

```
// test/functional/frontend/jobActionsTest.php
$browser->info('1 - The homepage')->
  get('/')->
```

```
  info('  1.4 - Jobs are sorted by date')->
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]',
      $browser->getMostRecentProgrammingJob()->getId()))->
  end()
;
```

9.6.5. Cada oferta de trabajo de la portada incluye un enlace

```
$browser->info('2 - The job page')->
  get('/')->

  info('  2.1 - Each job on the homepage is clickable and give detailed
information')->
  click('Web Developer', array(), array('position' => 1))->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'show')->
    isParameter('company_slug', 'sensio-labs')->
    isParameter('location_slug', 'paris-france')->
    isParameter('position_slug', 'web-developer')->
    isParameter('id', $browser->getMostRecentProgrammingJob()->getId())->
  end()
;
```

Para probar el enlace que muestra cada oferta de trabajo de la portada, simulamos que hemos pinchado sobre el texto *"Web Developer"*. Como en la página existen muchos enlaces con ese texto, le pedimos al navegador de forma explícita que pinche sobre el primero que encuentre (`array('position' => 1)`).

A continuación se prueban los parámetros de la petición para asegurarnos que el sistema de enrutamiento ha funcionado correctamente.

9.7. Aprendiendo con un ejemplo

En esta sección hemos incluido el código necesario para probar las páginas de cada categoría y la página de detalle de una oferta de trabajo. Te recomendamos que leas con atención todo el código porque te va a servir para aprender algunos trucos muy interesantes:

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    $loader = new sfPropelData();
    $loader->loadData(sfConfig::get('sf_test_dir').'/fixtures');

    return $this;
  }

  public function getMostRecentProgrammingJob()
  {
```

```
    // most recent job in the programming category
    $criteria = new Criteria();
    $criteria->add(JobeetCategoryPeer::SLUG, 'programming');
    $category = JobeetCategoryPeer::doSelectOne($criteria);

    $criteria = new Criteria();
    $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::GREATER_THAN);
    $criteria->addDescendingOrderByColumn(JobeetJobPeer::CREATED_AT);

    return JobeetJobPeer::doSelectOne($criteria);
  }

  public function getExpiredJob()
  {
    // expired job
    $criteria = new Criteria();
    $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::LESS_THAN);

    return JobeetJobPeer::doSelectOne($criteria);
  }
}
// test/functional/frontend/jobActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->info('1 - The homepage')->
  get('/')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'index')->
  end()->
  with('response')->begin()->
    info('  1.1 - Expired jobs are not listed')->
    checkElement('.jobs td.position:contains("expired")', false)->
  end()
;

$max = sfConfig::get('app_max_jobs_on_homepage');

$browser->info('1 - The homepage')->
  info(sprintf('  1.2 - Only %s jobs are listed for a category', $max))->
  with('response')->
    checkElement('.category_programming tr', $max)
;

$browser->info('1 - The homepage')->
  get('/')->
  info('  1.3 - A category has a link to the category page only if too many
jobs')->
  with('response')->begin()->
    checkElement('.category_design .more_jobs', false)->
    checkElement('.category_programming .more_jobs')->
  end()
```

```
;

$browser->info('1 - The homepage')->
  info('  1.4 - Jobs are sorted by date')->
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]',
$browser->getMostRecentProgrammingJob()->getId()))->
  end()
;

$browser->info('2 - The job page')->
  info('  2.1 - Each job on the homepage is clickable and give detailed
information')->
  click('Web Developer', array(), array('position' => 1))->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'show')->
    isParameter('company_slug', 'sensio-labs')->
    isParameter('location_slug', 'paris-france')->
    isParameter('position_slug', 'web-developer')->
    isParameter('id', $browser->getMostRecentProgrammingJob()->getId())->
  end()->

  info('  2.2 - A non-existent job forwards the user to a 404')->
  get('/job/foo-inc/milano-italy/0/painter')->
  with('response')->isStatusCode(404)->

  info('  2.3 - An expired job page forwards the user to a 404')->
  get(sprintf('/job/sensio-labs/paris-france/%d/web-developer',
$browser->getExpiredJob()->getId()))->
  with('response')->isStatusCode(404)
;
// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->info('1 - The category page')->
  info('  1.1 - Categories on homepage are clickable')->
  get('/')->
  click('Programming')->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'show')->
    isParameter('slug', 'programming')->
  end()->

  info(sprintf('  1.2 - Categories with more than %s jobs also have a "more"
link', sfConfig::get('app_max_jobs_on_homepage')))->
  get('/')->
  click('22')->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'show')->
```

```
    isParameter('slug', 'programming')->
  end()->

  info(sprintf('  1.3 - Only %s jobs are listed',
sfConfig::get('app_max_jobs_on_category')))->
  with('response')->checkElement('.jobs tr',
sfConfig::get('app_max_jobs_on_category'))->

  info('  1.4 - The job listed is paginated')->
  with('response')->begin()->
    checkElement('.pagination_desc', '/32 jobs/')->
    checkElement('.pagination_desc', '#page 1/2#')->
  end()->

  click('2')->
  with('request')->begin()->
    isParameter('page', 2)->
  end()->
  with('response')->checkElement('.pagination_desc', '#page 2/2#')
;
```

9.8. Depurando las pruebas funcionales

En ocasiones se producen errores al ejecutar las pruebas funcionales. Como el navegador que utiliza Symfony no tiene ningún tipo de interfaz gráfica, puede resultar muy difícil detectar el error. Afortunadamente, Symfony incluye un método llamado `debug()` que muestra las cabeceras y el contenido de la respuesta:

```
$browser->with('response')->debug();
```

Puedes incluir el método `debug()` en cualquier parte de un bloque *tester* de tipo `response` para detener la ejecución del script.

9.9. Conjuntos de pruebas funcionales

Si quieres ejecutar todas las pruebas funcionales de una aplicación, puedes utilizar la tarea `test:functional`:

```
$ php symfony test:functional frontend
```

La tarea `test:functional` muestra como resultado una sola línea para cada archivo de pruebas:

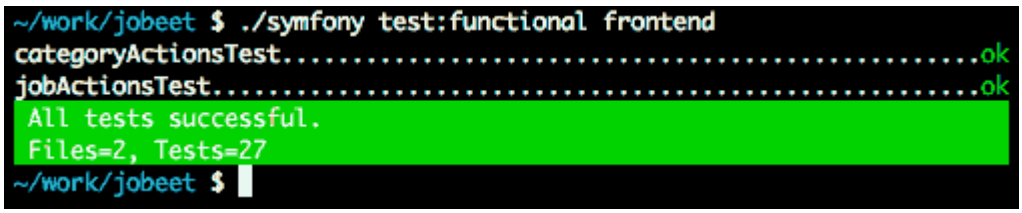

Figura 9.2. Conjuntos de pruebas funcionales

9.10. Conjuntos de pruebas

Como habrás supuesto, también existe una tarea que permite lanzar todas las pruebas (unitarias y funcionales) de un proyecto:

```
$ php symfony test:all
```

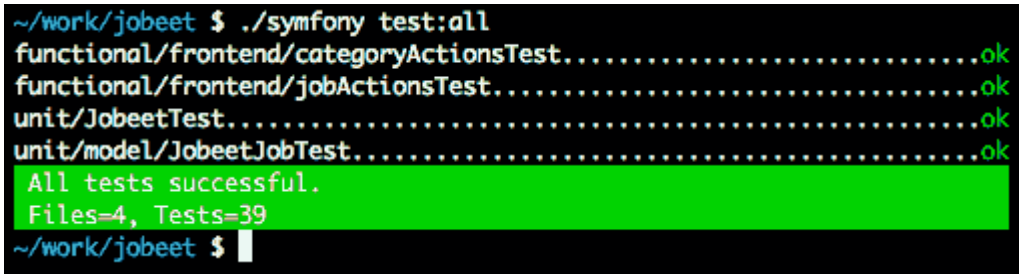

Figura 9.3. Conjuntos de pruebas unitarias y funcionales

9.11. Nos vemos mañana

Con este tutorial finalizamos el recorrido que hemos realizado por las herramientas que incluye Symfony para crear pruebas. A estas alturas ya no tienes ninguna excusa para no probar correctamente tus aplicaciones. Gracias al subframework `lime` y al subframework para pruebas funcionales de Symfony, puedes crear pruebas con muy poco esfuerzo.

No obstante, ten en cuenta que no hemos profundizado en las posibilidades de las pruebas funcionales. Por ese motivo, a partir de ahora, cada vez que añadamos una nueva funcionalidad en la aplicación, también vamos a escribir las pruebas necesarias para aprender las características más avanzadas del subframework de pruebas.

Mañana hablaremos de uno de los componentes más espectaculares de Symfony: el subframework de formularios.

Capítulo 10. Los formularios

La segunda semana del tutorial Jobeet arrancó muy intensamente con la introducción del framework de pruebas de Symfony. En la lección de hoy vamos a estudiar el framework de formularios.

10.1. El framework de formularios

La mayoría de sitios web incluye algún tipo de formulario, desde el formulario simple de contacto hasta formularios complejos con decenas de campos. Además, crear los formularios es una de las tareas más aburridas y difíciles de los programadores web: tienes que crear el código HTML del formulario, incluir reglas de validación para los datos de todos los campos, procesar los valores enviados por los usuarios y guardarlos en la base de datos, mostrar los posibles mensajes de error, volver a mostrar los datos en el formulario si se produce un error, etc.

Para no tener que reinventar la rueda continuamente, Symfony incluye un framework que facilita la gestión de los formularios. El framework de formularios de Symfony se compone de tres partes:

- **validación**: el subframework de validación incluye las clases necesarias para validar los datos (números enteros, cadenas de texto, direcciones de email, etc.)
- **widgets**: el subframework de widgets incluye las clases que muestra el código HTML de los campos del formulario (`<input>`, `<textarea>`, `<select>`, ...)
- **formularios**: las clases de formulario representan a los formularios construidos con widgets y validadores y proporcionan métodos para facilitar la gestión del formulario. Cada campo del formulario dispone de su propio validador y su propio widget.

10.2. Formularios

Un formulario de Symfony es una clase formada por campos de formulario. Cada campo dispone de un nombre, un validador y un widget. A continuación se muestra cómo se puede crear un formulario de contacto sencillo llamado `ContactForm`:

```
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'email'   => new sfWidgetFormInput(),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->setValidators(array(
      'email'   => new sfValidatorEmail(),
```

```
      'message' => new sfValidatorString(array('max_length' => 255)),
    ));
  }
}
```

Los campos del formulario se configuran en el método `configure()` mediante los métodos `setValidators()` y `setWidgets()`.

Sugerencia

El framework de formularios incluye muchos widgets (http://www.symfony-project.org/api/1_2/widget) y validadores (http://www.symfony-project.org/api/1_2/validator) . La API de Symfony describe cada uno detalladamente, con todas sus opciones, errores y mensajes de error por defecto.

Los nombres de las clases de los widgets y validadores son muy explícitos: el campo `email` se representará mediante una etiqueta `<input>` de HTML (`sfWidgetFormInput`) y se validará que su valor sea una dirección de correo electrónico válida (`sfValidatorEmail`). El campo `message` se representará como una etiqueta `<textarea>` (`sfWidgetFormTextarea`) y se validará que su valor sea una cadena de texto de no más de 255 caracteres de longitud (`sfValidatorString`).

Por defecto todos los campos del formulario son obligatorios, ya que el valor por defecto de la opción `required` es `true`. Por tanto, la validación anterior del campo `email` es equivalente a `new sfValidatorEmail(array('required' => true))`.

Sugerencia

También es posible combinar dos formularios mediante el método `mergeForm()` o incluir un formulario dentro de otro mediante el método `embedForm()`:

```
$this->mergeForm(new AnotherForm());
$this->embedForm('name', new AnotherForm());
```

10.3. Formularios de Propel

Normalmente, los valores enviados con el formulario se guardan o serializan en una base de datos. Como Symfony ya dispone de toda la información sobre el modelo de tu base de datos, es capaz de generar automáticamente los formularios a partir de esa información. De hecho, cuando ejecutábamos la tarea `propel:build-all` durante el tutorial del día 3, Symfony ejecutaba internamente la tarea `propel:build-forms`:

```
$ php symfony propel:build-forms
```

Los formularios que genera la tarea `propel:build-forms` se guardan en el directorio `lib/form/`. La forma en la que se organizan estos archivos generados automáticamente es similar a la del directorio `lib/model/`. Cada clase del modelo dispone de una clase de formulario (la clase `JobeetJob` dispone por ejemplo de `JobeetJobForm`). Inicialmente estas clases de formulario están vacías, ya que heredan de una clase base de formularios:

```
// lib/form/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
  }
}
```

Sugerencia

Si echas un vistazo a los archivos generados automáticamente en el subdirectorio `lib/form/base/`, verás muchos buenos ejemplos de cómo utilizar los widgets y validadores incluidos en Symfony.

10.3.1. Personalizando el formulario de las ofertas de trabajo

El formulario de las ofertas de trabajo es un buen ejemplo para aprender a personalizar los formularios. A continuación se muestran todos los pasos necesarios para personalizar este formulario.

En primer lugar, modifica el enlace *Post a Job* del layout para que puedas probar las modificaciones directamente en el navegador:

```
<!-- apps/frontend/templates/layout.php -->
<a href="<?php echo url_for('@job_new') ?>">Post a Job</a>
```

Por defecto los formularios de Propel muestran campos para todas las columnas de la tabla. No obstante, en el formulario para insertar una oferta de trabajo, algunos campos no deben ser editables por los usuarios. Eliminar campos en un formulario es tan sencillo como utilizar la función `unset()` de PHP:

```
// lib/form/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated']
    );
  }
}
```

Eliminar un campo de formulario significa que se eliminan tanto su widget como su validador.

Normalmente, la configuración del formulario debe ser más precisa de lo que se puede determinar a partir del esquema de la base de datos. La columna `email` por ejemplo es un campo de tipo `varchar` en el esquema, pero necesitamos que sea validado como si fuera un email. Para ello, modifica el validador `sfValidatorString` por `sfValidatorEmail`:

```
// lib/form/JobeetJobForm.class.php
public function configure()
{
  // ...

  $this->validatorSchema['email'] = new sfValidatorEmail();
}
```

Por su parte, aunque la columna `type` también es de tipo `varchar` en el esquema de datos, queremos restringir su valor a uno de los tres siguientes valores: *full time* (jornada completa), *part time* (jornada parcial) y *freelance*.

En primer lugar, define los posibles valores en la clase `JobeetJobPeer`:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public $types = array(
    'full-time' => 'Full time',
    'part-time' => 'Part time',
    'freelance' => 'Freelance',
  );

  // ...
}
```

A continuación, utiliza el widget `sfWidgetFormChoice` para el campo `type`:

```
$this->widgetSchema['type'] = new sfWidgetFormChoice(array(
  'choices'  => JobeetJobPeer::$types,
  'expanded' => true,
));
```

El widget `sfWidgetFormChoice` no tiene un equivalente directo en forma de etiqueta HTML, ya que se muestra de forma diferente en función del valor de sus opciones de configuración `expanded` y `multiple`:

- Lista desplegable (`<select>`): `array('multiple' => false, 'expanded' => false)`
- Lista desplegable que permite seleccionar varios valores (`<select multiple="multiple">`): `array('multiple' => true, 'expanded' => false)`
- Lista de *radio buttons*: `array('multiple' => false, 'expanded' => true)`
- Lista de *checkboxes*: `array('multiple' => true, 'expanded' => true)`

Nota

Si quieres que uno de los *radio button* se muestre seleccionado inicialmente (`full-time` por ejemplo), puedes modificar su valor por defecto en el esquema de datos.

Restringir los posibles valores de un campo de formulario no evita que usuarios malintencionados con conocimientos avanzados puedan manipular sus valores con

herramientas como curl (http://curl.haxx.se/) o la extensión Web Developer Toolbar de Firefox (http://chrispederick.com/work/web-developer/) . Por este motivo, vamos a modificar también el validador para restringir los posibles valores a elegir:

```
$this->validatorSchema['type'] = new sfValidatorChoice(array(
  'choices' => array_keys(JobeetJobPeer::$types),
));
```

Por otra parte, la columna `logo` almacena el nombre del archivo que contiene el logotipo asociado con la oferta de trabajo, por lo que debemos cambiar su widget para que muestre un campo de formulario para elegir un archivo:

```
$this->widgetSchema['logo'] = new sfWidgetFormInputFile(array(
  'label' => 'Company logo',
));
```

Symfony también genera para cada campo una etiqueta o título que se muestra en la etiqueta `<label>`. La etiqueta generada se puede modificar con la opción `label`. También es posible modificar varias etiquetas a la vez utilizando el método `setLabels()` del array de widgets:

```
$this->widgetSchema->setLabels(array(
  'category_id'    => 'Category',
  'is_public'      => 'Public?',
  'how_to_apply'   => 'How to apply?',
));
```

Además, debemos modificar el validador por defecto del campo `logo`:

```
$this->validatorSchema['logo'] = new sfValidatorFile(array(
  'required'   => false,
  'path'       => sfConfig::get('sf_upload_dir').'/jobs',
  'mime_types' => 'web_images',
));
```

El validador `sfValidatorFile` es muy interesante porque realiza varias tareas:

- Valida que el archivo subido sea una imagen en un formato adecuado para las páginas web (gracias a la opción `mime_types`)
- Cambia el nombre del archivo por un valor único
- Guarda el archivo en la ruta indicada con la opción `path`
- Actualiza el valor de la columna `logo` con el nombre generado anteriormente

Nota

No te olvides de crear el directorio para guardar los logotipos (`web/uploads/jobs/`) y asegúrate que el servidor web tenga permisos de escritura sobre ese directorio.

Como el validador sólo guarda en la base de datos la ruta relativa hasta la imagen, modifica la ruta utilizada en la plantilla `showSuccess`:

```
// apps/frontend/modules/job/template/showSuccess.php
<img src="/uploads/jobs/<?php echo $job->getLogo() ?>" alt="<?php echo
$job->getCompany() ?> logo" />
```

Sugerencia

Si en el modelo existe un método llamado `generateLogoFilename()`, el validador utiliza este método para generar automáticamente el nombre del archivo subido. Al método anterior se le pasa como argumento el objeto `sfValidatedFile`.

Además de poder redefinir el valor de las etiquetas generadas para los campos del formulario, también puedes establecer un mensaje de ayuda. Vamos a añadir un mensaje de ayuda para explicar mejor la finalidad del campo `is_public`:

```
$this->widgetSchema->setHelp('is_public', 'Whether the job can also be
published on affiliate websites or not.');
```

Combinando todo lo que hemos hecho en esta sección, la clase `JobeetJobForm` definitiva contiene el siguiente código:

```
// lib/form/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated']
    );

    $this->validatorSchema['email'] = new sfValidatorEmail();

    $this->widgetSchema['type'] = new sfWidgetFormChoice(array(
      'choices' => JobeetJobPeer::$types,
      'expanded' => true,
    ));
    $this->validatorSchema['type'] = new sfValidatorChoice(array(
      'choices' => array_keys(JobeetJobPeer::$types),
    ));

    $this->widgetSchema['logo'] = new sfWidgetFormInputFile(array(
      'label' => 'Company logo',
    ));

    $this->widgetSchema->setLabels(array(
      'category_id'    => 'Category',
      'is_public'      => 'Public?',
      'how_to_apply'   => 'How to apply?',
    ));

    $this->validatorSchema['logo'] = new sfValidatorFile(array(
      'required'   => false,
      'path'       => sfConfig::get('sf_upload_dir').'/jobs',
      'mime_types' => 'web_images',
    ));
```

```
    $this->widgetSchema->setHelp('is_public', 'Whether the job can also be
published on affiliate websites or not.');
  }
}
```

10.3.2. La plantilla del formulario

Después de personalizar los campos del formulario, el siguiente paso consiste en mostrarlos. La plantilla del formulario es la misma para el formulario de insertar una oferta de trabajo y para el formulario de modificar los datos de una oferta existente. De hecho, tanto la plantilla `newSuccess.php` como la plantilla `editSuccess.php` son muy similares:

```
<!-- apps/frontend/modules/job/templates/newSuccess.php -->
<?php use_stylesheet('job.css') ?>

<h1>Post a Job</h1>

<?php include_partial('form', array('form' => $form)) ?>
```

Nota

Si todavía no has añadido la hoja de estilos `job`, debes añadirla en las dos plantillas mediante la instrucción `<?php use_stylesheet('job.css') ?>`

El formulario se muestra a través de un elemento parcial llamado `_form`. Reemplaza el contenido de ese elemento parcial `_form` por el siguiente código:

```
<!-- apps/frontend/modules/job/templates/_form.php -->
<?php include_stylesheets_for_form($form) ?>
<?php include_javascripts_for_form($form) ?>

<?php echo form_tag_for($form, '@job') ?>
  <table id="job_form">
    <tfoot>
      <tr>
        <td colspan="2">
          <input type="submit" value="Preview your job" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

Los helpers `include_javascripts_for_form()` y `include_stylesheets_for_form()` incluyen respectivamente los archivos JavaScript y CSS que utilizan los widgets del formulario.

Sugerencia

Aunque el formulario para insertar una nueva oferta de trabajo no utiliza ningún archivo JavaScript o CSS, te recomendamos que dejes la llamada a estos helpers *"por si acaso"*. Estas llamadas pueden venir muy bien posteriormente cuando decidas insertar algún widget que requiere JavaScript o CSS.

El helper `form_tag_for()` genera una etiqueta `<form>` a partir del formulario y ruta indicados y modifica el método HTTP a `POST` o `PUT` dependiendo de si el objeto es nuevo o no. Este helper también tiene en cuenta si es necesario añadir el atributo `enctype` en caso de que el formulario permite adjuntar archivos.

Por último, la instrucción `<?php echo $form ?>` se encarga de generar el código HTML de los widgets del formulario.

Modificando el aspecto de un formulario

La instrucción `<?php echo $form ?>` muestra por defecto cada widget del formulario en una fila de una tabla. No obstante, en muchas ocasiones necesitas cambiar la disposición de los elementos del formulario. Por este motivo, el objeto que representa al formulario incluye varios métodos útiles para modificar su disposición:

Método	Descripción
`render()`	Muestra el formulario (equivalente a lo que muestra `echo $form`)
`renderHiddenFields()`	Muestra los campos ocultos
`hasErrors()`	Devuelve `true` si existe algún error en el formulario
`hasGlobalErrors()`	Devuelve `true` si existe algún error global en el formulario
`getGlobalErrors()`	Devuelve un array con los errores globales
`renderGlobalErrors()`	Muestra los errores globales

El formulario también se puede manejar como si fuera un array de campos de formulario. Puedes acceder por ejemplo al campo `company` mediante `$form['company']`. El objeto devuelto incluye los métodos necesarios para mostrar cada campo del formulario:

Método	Descripción
`renderRow()`	Muestra la fila de un campo
`render()`	Muestra el widget asociado con el campo
`renderLabel()`	Muestra el título o etiqueta de un campo
`renderError()`	Muestra los posibles mensajes de error del campo
`renderHelp()`	Muestra el mensaje de ayuda del campo

La instrucción `echo $form` es equivalente a:

```
<?php foreach ($form as $widget): ?>
  <?php echo $widget->renderRow() ?>
<?php endforeach; ?>
```

10.3.3. La acción del formulario

Ahora que ya tenemos la clase del formulario y la plantilla que lo muestra, vamos a utilizarlo en algunas acciones. El formulario de las ofertas de trabajo lo utilizan los siguientes cinco métodos del módulo `job`:

- **new**: muestra un formulario vacío para insertar una nueva oferta de trabajo.
- **edit**: muestra un formulario para modificar los datos almacenados de una oferta de trabajo.
- **create**: crea una nueva oferta de trabajo a partir de los datos enviados por el usuario con el formulario.
- **update**: actualiza los datos de una oferta de trabajo existente a partir de los datos enviados por el usuario con el formulario.
- **processForm**: este método lo utilizan los métodos `create` y `update` para procesar el formulario (validación, volver a mostrar los datos del formulario y guardado o serialización en la base de datos).

El flujo de trabajo de todos los formularios se muestra en la siguiente imagen:

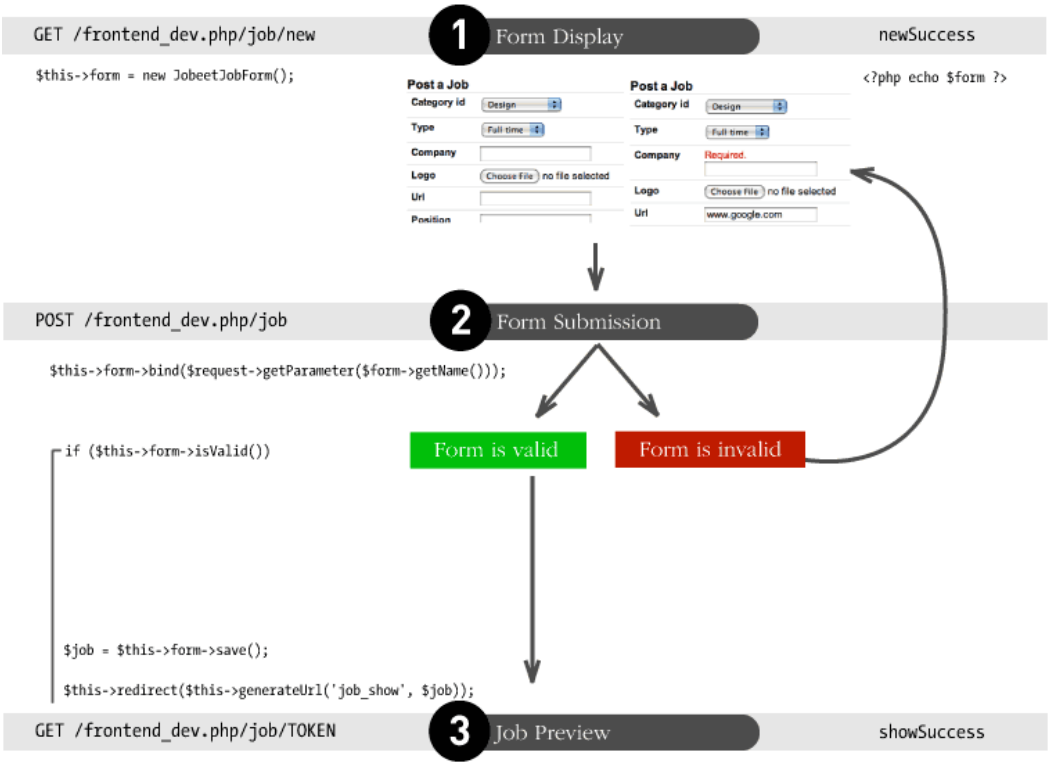

Figura 10.1. Flujo de trabajo de los formularios

Como en un tutorial pasado creamos una colección de rutas de Propel para el módulo `job`, podemos simplificar el código de los métodos que gestionan el formulario:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeNew(sfWebRequest $request)
```

```
{
  $this->form = new JobeetJobForm();
}

public function executeCreate(sfWebRequest $request)
{
  $this->form = new JobeetJobForm();
  $this->processForm($request, $this->form);
  $this->setTemplate('new');
}

public function executeEdit(sfWebRequest $request)
{
  $this->form = new JobeetJobForm($this->getRoute()->getObject());
}

public function executeUpdate(sfWebRequest $request)
{
  $this->form = new JobeetJobForm($this->getRoute()->getObject());
  $this->processForm($request, $this->form);
  $this->setTemplate('edit');
}

public function executeDelete(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->delete();

  $this->redirect('job/index');
}

protected function processForm(sfWebRequest $request, sfForm $form)
{
  $form->bind(
    $request->getParameter($form->getName()),
    $request->getFiles($form->getName())
  );

  if ($form->isValid())
  {
    $job = $form->save();

    $this->redirect($this->generateUrl('job_show', $job));
  }
}
```

Cada vez que se accede a la página `/job/new`, se crea una nueva instancia de un formulario y se pasa a la plantilla en la acción `new`.

Cuando el usuario envía el formulario (acción `create`), se asocia (mediante el método `bind()`) con los valores enviados por el usuario y se ejecuta la validación de los datos.

Cuando el formulario está asociado, ya se puede comprobar su validez con el método `isValid()`. Si el formulario es válido (el método `isValid()` devuelve `true`), la oferta de trabajo se guarda en la base de datos (`$form->save()`) y se redirige al usuario a la página que previsualiza la oferta. Si el formulario no es válido, se vuelve a mostrar la plantilla `newSuccess.php` con los mismos datos que envió el usuario y con todos los mensajes de error asociados.

Sugerencia

El método `setTemplate()` modifica la plantilla utilizada por la acción. Si el formulario enviado no es válido, los métodos `create` y `update` utilizan la misma plantilla para volver a mostrar en las acciones `new` y `edit` el formulario con los mensajes de error asociados.

La modificación de una oferta de trabajo existente es un proceso muy similar. La única diferencia entre la acción `new` y la acción `edit` es que en el segundo caso, se pasa como primer argumento del constructor del formulario el objeto que representa la oferta de trabajo que se va a modificar. Este objeto se emplea para establecer los valores iniciales de los widgets de la plantilla (en los formularios de Propel los valores iniciales forman un objeto, pero en los formularios sencillos se indican en forma de array simple).

El formulario para insertar una nueva oferta de trabajo también puede mostrar unos determinados valores iniciales. Una forma sencilla de conseguirlo es declarar esos valores iniciales en el esquema de la base de datos. Otra forma consiste en pasar un objeto modificado de tipo `Job` al constructor del formulario.

Modifica el método `executeNew()` para establecer el valor `full-time` como valor por defecto de la columna `type`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeNew(sfWebRequest $request)
{
  $job = new JobeetJob();
  $job->setType('full-time');

  $this->form = new JobeetJobForm($job);
}
```

Nota

Cuando el formulario se asocia a los datos del usuario, los valores iniciales se reemplazan por los valores enviados por el usuario. Estos valores se utilizan cuando el formulario debe volver a mostrar los datos introducidos por el usuario después de que la validación no haya sido satisfactoria.

10.3.4. Protegiendo el formulario de las ofertas de trabajo con un token

Ahora mismo el formulario funciona correctamente y el usuario debe indicar el token de la oferta de trabajo. No obstante, el token asociado con la oferta de trabajo se debe

generar automáticamente cada vez que se crea una oferta de trabajo, ya que no queremos que sean los usuarios los que tengan que indicar un token único.

Para ello, modifica el método `save()` de `JobeetJob` para añadir la lógica que genera el token antes de guardar la oferta de trabajo:

```
// lib/model/JobeetJob.php
public function save(PropelPDO $con = null)
{
  // ...

  if (!$this->getToken())
  {
    $this->setToken(sha1($this->getEmail().rand(11111, 99999)));
  }

  return parent::save($con);
}
```

Ahora ya puedes eliminar el campo `token` del formulario:

```
// lib/form/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated'],
      $this['token']
    );

    // ...
  }

  // ...
}
```

Si recuerdas los escenarios que describimos durante el tutorial del día 2, una oferta de trabajo sólo se puede editar si el usuario conoce su token asociado. Ahora mismo es muy sencillo modificar o borrar cualquier oferta de trabajo adivinando su URL. El motivo es que la URL de la acción de modificar la oferta de trabajo siempre es `/job/ID/edit`, donde `ID` es la clave primaria de la oferta de trabajo.

Las rutas de tipo `sfPropelRouteCollection` generan por defecto URL que contienen el valor de la clave primaria, pero se puede modificar por cualquier otra columna cuyo valor sea único indicándolo en la opción `column`:

```
# apps/frontend/config/routing.yml
job:
  class:        sfPropelRouteCollection
  options:      { model: JobeetJob, column: token }
  requirements: { token: \w+ }
```

En la configuración de la ruta anterior también hemos modificado la opción `requirements` para la columna del `token`, ya que el requisito por defecto de Symfony para una clave primaria es `\d+`

Ahora, todas las rutas relacionadas con las ofertas de trabajo salvo `job_show_user`, incluyen el token y no la clave primaria. La ruta para editar una oferta de trabajo por ejemplo tiene el siguiente aspecto:

```
http://jobeet.localhost/job/TOKEN/edit
```

No te olvides de modificar también el enlace de la plantilla `showSuccess`:

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<a href="<?php echo url_for('job_edit', $job) ?>">Edit</a>
```

10.4. La página de previsualización

La página de previsualización de la oferta de trabajo es la misma que la página que muestra los detalles de una oferta. Gracias al sistema de enrutamiento, si el usuario accede con el token adecuado, su valor será accesible en el parámetro `token` de la petición.

Si el usuario accede con una URL que incluye el token, añadimos en la parte superior de la página una barra con opciones útiles para los administradores. Añade al principio de la plantilla `showSuccess` un elemento parcial para incluir la barra de administrador y elimina el enlace `edit` que se encuentra al final de la página:

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php if ($sf_request->getParameter('token') == $job->getToken()): ?>
  <?php include_partial('job/admin', array('job' => $job)) ?>
<?php endif; ?>
```

A continuación crea el elemento parcial `_admin`:

```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<div id="job_actions">
  <h3>Admin</h3>
  <ul>
    <?php if (!$job->getIsActivated()): ?>
      <li><?php echo link_to('Edit', 'job_edit', $job) ?></li>
      <li><?php echo link_to('Publish', 'job_edit', $job) ?></li>
    <?php endif; ?>
    <li><?php echo link_to('Delete', 'job_delete', $job, array('method' =>
'delete', 'confirm' => 'Are you sure?')) ?></li>
    <?php if ($job->getIsActivated()): ?>
      <li<?php $job->expiresSoon() and print ' class="expires_soon"' ?>>
        <?php if ($job->isExpired()): ?>
          Expired
        <?php else: ?>
          Expires in <strong><?php echo $job->getDaysBeforeExpires()
?></strong> days
        <?php endif; ?>
```

```
        <?php if ($job->expiresSoon()): ?>
         - <a href="">Extend</a> for another <?php echo
sfConfig::get('app_active_days') ?> days
        <?php endif; ?>
      </li>
    <?php else: ?>
      <li>
        [Bookmark this <?php echo link_to('URL', 'job_show', $job, true) ?> to
manage this job in the future.]
      </li>
    <?php endif; ?>
  </ul>
</div>
```

El elemento parcial anterior incluye mucho código, pero la mayor parte de su código es muy fácil de entender.

Para hacer que el código de la plantilla sea más fácil de leer, hemos añadido varios atajos en la clase `JobeetJob`:

```
// lib/model/JobeetJob.php
public function getTypeName()
{
  return $this->getType() ? JobeetJobPeer::$types[$this->getType()] : '';
}

public function isExpired()
{
  return $this->getDaysBeforeExpires() < 0;
}

public function expiresSoon()
{
  return $this->getDaysBeforeExpires() < 5;
}

public function getDaysBeforeExpires()
{
  return floor(($this->getExpiresAt('U') - time()) / 86400);
}
```

La barra de administrador es diferente en función del estado de la oferta de trabajo:

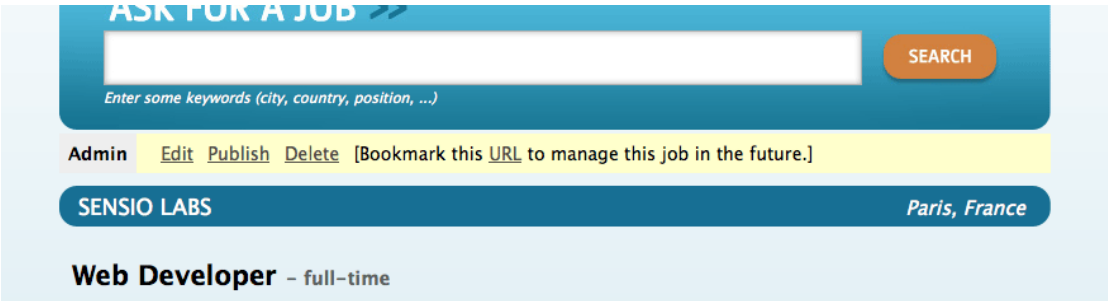

Figura 10.2. Oferta de trabajo sin activar

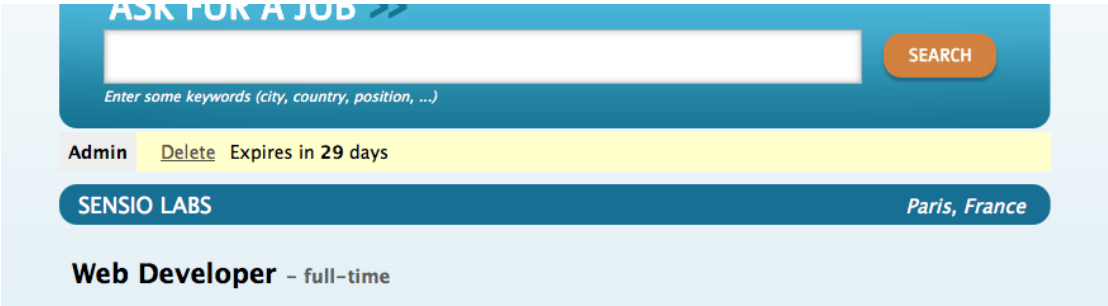

Figura 10.3. Oferta de trabajo activada

Nota

Hasta la próxima sección no vas a poder ver la barra de las ofertas de trabajo activadas.

10.5. Activando y publicando las ofertas de trabajo

En la sección anterior existe un enlace para publicar una oferta de trabajo. Debemos modificar la dirección del enlace para que apunte a una nueva acción llamada `publish`. En vez de crear una ruta nueva, podemos simplemente modificar la configuración de la ruta `job` existente:

```
# apps/frontend/config/routing.yml
job:
  class:   sfPropelRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: put }
  requirements:
    token: \w+
```

En la opción `object_actions` se incluye un array con las acciones adicionales del objeto, por lo que ahora ya podemos modificar el enlace *"Publish"*:

```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<li>
  <?php echo link_to('Publish', 'job_publish', $job, array('method' => 'put'))
?>
</li>
```

Por último, crea la acción `publish`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executePublish(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->publish();

  $this->getUser()->setFlash('notice', sprintf('Your job is now online for %s
days.', sfConfig::get('app_active_days')));
```

```
  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

Si te fijas atentamente, verás que el enlace *"Publish"* se envía con el método `PUT` de HTTP. Para simular el método `PUT`, el enlace se convierte automáticamente en un formulario cuando se pincha sobre el.

Además, como al crear la aplicación activamos la protección frente a los ataques CSRF, el helper `link_to()` incluye en el enlace un token para CSRF y el método `checkCSRFProtection()` del objeto que representa a la petición comprueba la validez del token después de realizar la petición.

El método `executePublish()` utiliza a su vez un método `publish()` nuevo que puede ser tan sencillo como el código que se muestra a continuación:

```
// lib/model/JobeetJob.php
public function publish()
{
  $this->setIsActivated(true);
  $this->save();
}
```

Ahora ya está todo preparado para que pruebes en el navegador la nueva funcionalidad para publicar ofertas de trabajo.

No obstante, todavía tenemos que retocar una cosa. Las ofertas de trabajo que no están activas no deberían verse, lo que significa que no se deben mostrar en la página principal de Jobeet y tampoco se deben poder acceder mediante su URL. Como en su día creamos un método llamado `addActiveJobsCriteria()` para restringir un objeto `Criteria` para que sólo obtenga las ofertas de trabajo activas, podemos modificar ese método para añadir este nuevo requerimiento:

```
// lib/model/JobeetJobPeer.php
static public function addActiveJobsCriteria(Criteria $criteria = null)
{
  // ...

  $criteria->add(self::IS_ACTIVATED, true);

  return $criteria;
}
```

Y eso es todo, por lo que ya puedes probarlo en tu navegador. En la portada de Jobeet ya no se muestra ninguna oferta de trabajo que no esté activada y tampoco se puede acceder a estas ofertas a través de su URL. No obstante, todavía se puede acceder a estas ofertas si se conoce la URL que contiene el token. En ese caso, se muestra la página de previsualización de la oferta de trabajo junto con la barra de administrador.

Esta es una de las grandes ventajas del patrón de diseño MVC y de la refactorización que hemos hecho hasta el momento: un solo cambio en un solo método es suficiente para añadir una nueva funcionalidad de la aplicación.

Nota

Cuando creamos el método `getWithJobs()`, se nos olvidó utilizar el método `addActiveJobsCriteria()`. Por tanto, modifica el método y añade este nuevo requrimiento:

```
class JobeetCategoryPeer extends BaseJobeetCategoryPeer
{
  static public function getWithJobs()
  {
    // ...
     $criteria->add(JobeetJobPeer::IS_ACTIVATED, true);

     return $criteria;
  }
```

10.6. Nos vemos mañana

El tutorial de hoy ha incluido un montón de información nueva, pero esperamos que ahora entiendas mejor el funcionamiento del framework de formularios de Symfony.

Somos conscientes de que algunos os habéis dado cuenta de que se nos ha olvidado algo, ya que no hemos creado ninguna prueba para las nuevas funcionalidades de la aplicación. Como crear pruebas es algo muy importante al desarrollar una aplicación, esto es lo primero que vamos a hacer en el tutorial de mañana.

Capítulo 11. Probando los formularios

Ayer creamos nuestro primer formulario con Symfony. Los usuarios de la aplicación ya pueden insertar una nueva oferta de trabajo en Jobeet, pero se nos acabó el tiempo antes de que pudiéramos crear algunas pruebas unitarias y funcionales.

Por tanto, durante el día de hoy vamos a añadir las pruebas necesarias para el nuevo formulario. Además, seguiremos aprendiendo nuevas características del framework de formularios.

Utilizando el framework de formularios fuera de Symfony

Los componentes de Symfony se encuentran muy desacoplados entre sí. Esto significa que la mayoría de componentes se pueden utilizar de forma individual sin tener que hacer uso de todo el framework. Este es el caso por ejemplo del framework de formularios, que no tiene ninguna dependencia con Symfony. Si quieres utilizarlo en cualquier aplicación PHP, sólo tienes que copiarte los directorios `lib/form/`, `lib/widgets/` y `lib/validators/`.

Otro de los componentes que puedes reutilizar en tus aplicaciones es el sistema de enrutamiento. Copia el directorio `lib/routing/` en tu proyecto y empieza a disfrutar de las URL *limpias* en cualquier aplicación que no sea Symfony.

A continuación se muestran los componentes que son independientes de la **plataforma Symfony**:

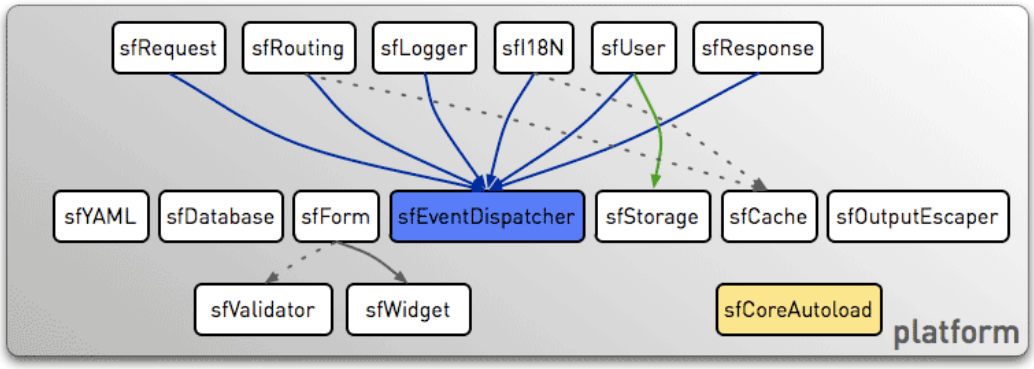

Figura 11.1. La plataforma Symfony

11.1. Enviando un formulario

Abre el archivo `jobActionsTest` para añadir las pruebas funcionales del proceso de creación y validación de una oferta de trabajo.

Añade el siguiente código al final de ese archivo para acceder a la página de inserción de una nueva oferta de trabajo:

```
// test/functional/frontend/jobActionsTest.php
$browser->info('3 - Post a Job page')->
  info('  3.1 - Submit a Job')->
```

```
  get('/job/new')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'new')->
  end()
;
```

Cuando hablamos de las pruebas funcionales ya vimos el método `click()` para simular que se ha pinchado sobre un enlace. El mismo método `click()` también se puede utilizar para enviar un formulario. En el caso del formulario, se puede pasar como segundo argumento del método `click()` un array con los valores que se quieren enviar en el formulario. Como si se tratara de un navegador *de verdad*, el objeto que simula el navegador combina los valores por defecto del formulario con los valores que se acaban de enviar.

Antes de pasar los nuevos valores de los campos del formulario, es necesario conocer el nombre de cada campo. Si visualizas el código fuente de la página o utilizas la opción *Forms > Display Form Details"* de la extensión *Web Developer Toolbar* del navegador Firefox, verás que el nombre del campo `company` es `jobeet_job[company]`.

Nota

Cuando PHP encuentra un campo de formulario con un nombre como `jobeet_job[company]`, lo convierte automáticamente en un array de nombre `jobeet_job`.

Para que el código sea un poco más limpio, vamos a cambiar el formato del nombre de los campos del formulario a `job[%s]`, por lo que es necesario que añadas el siguiente código al final del método `configure()` de `JobeetJobForm`:

```
// lib/form/JobeetJobForm.class.php
$this->widgetSchema->setNameFormat('job[%s]');
```

Después de realizar este cambio, el nombre del campo `company` en el navegador debería ser ahora `job[company]`. Ahora ya podemos pulsar en el botón *"Preview your job"* y ya podemos enviar valores en el formulario:

```
// test/functional/frontend/jobActionsTest.php
$browser->info('3 - Post a Job page')->
  info('  3.1 - Submit a Job')->

  get('/job/new')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'new')->
  end()->

  click('Preview your job', array('job' => array(
    'company'      => 'Sensio Labs',
    'url'          => 'http://www.sensio.com/',
    'logo'         => sfConfig::get('sf_upload_dir').'/jobs/sensio-labs.gif',
    'position'     => 'Developer',
    'location'     => 'Atlanta, USA',
```

```
    'description'  => 'You will work with symfony to develop websites for our
customers.',
    'how_to_apply' => 'Send me an email',
    'email'        => 'for.a.job@example.com',
    'is_public'    => false,
  )))->

  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'create')->
  end()->
;
```

El navegador también puede simular que se suben archivos adjuntos si pasas la ruta absoluta del archivo que se quiere subir.

El código anterior también comprueba que después de enviar el formulario, la acción que se ejecuta es `create`.

11.2. El tester de formularios

El formulario que hemos enviado en la prueba anterior debería ser válido. Para comprobar su validez, puedes utilizar el *tester* de formularios:

```
with('form')->begin()->
  hasErrors(false)->
end()->
```

El *tester* de formularios dispone de varios métodos para probar el estado del formulario actual, como por ejemplo sus posibles errores.

Si te equivocas al crear la prueba y no pasa satisfactoriamente, puedes utilizar la instrucción `with('response')->debug()` que explicamos durante el tutorial del día 9. Aún así tendrías que investigar el código HTML generado para comprobar si se muestra algún mensaje de error. Como esto último no es muy cómodo, el *tester* de formularios también incluye un método `debug()` que muestra el estado del formulario y todos sus mensajes de error asociados:

```
with('form')->debug()
```

11.3. Probando la redirección

Como el formulario es válido, la oferta de trabajo debería haberse insertado y el usuario debe haber sido redirigido a la página `show`:

```
isRedirected()->
followRedirect()->

with('request')->begin()->
  isParameter('module', 'job')->
  isParameter('action', 'show')->
end()->
```

El método `isRedirected()` comprueba si la página ha sido redirigida y el método `followRedirect()` sigue la redirección indicada.

> **Nota**
>
> La clase del navegador no sigue las redirecciones de forma automática porque puede ser necesario inspeccionar los objetos antes de realizar la redirección.

11.4. El tester de Propel

A continuación queremos probar que la oferta de trabajo se ha insertado en la base de datos y también vamos a comprobar que su columna `is_activated` vale `false` porque el usuario todavía no la ha publicado.

La mejor forma de realizar esta comprobación consiste en utilizar un nuevo *tester* específico para Propel. Como este *tester* de Propel no está registrado por defecto, lo primero que debes hacer es añadirlo al navegador:

```
$browser->setTester('propel', 'sfTesterPropel');
```

El *tester* de Propel incluye el método `check()` para comprobar que uno o más objetos de la base de datos cumplen con los criterios de búsqueda pasados como argumento.

```
with('propel')->begin()->
  check('JobeetJob', array(
    'location'     => 'Atlanta, USA',
    'is_activated' => false,
    'is_public'    => false,
  ))->
end()
```

El criterio de búsqueda se puede indicar como un array de valores (como en el ejemplo anterior) o mediante una instancia del objeto `Criteria`, que es más útil cuando las búsquedas son complejas. Si se pasa como tercer argumento del método `check()` un valor booleano, sólo se comprueba si existe o no existe al menos un objeto que cumpla los criterios de búsqueda. El valor por defecto de este tercer argumento es `true`. Este tercer argumento de `check()` también puede ser un número entero, en cuyo caso se comprueba si existen en la base de datos el número de objetos indicado en ese argumento.

11.5. Probando la existencia de errores

Cuando se envían datos válidos en el formulario, el proceso de creación de una oferta de trabajo funciona tal y como se esperaba. A continuación se va a probar su comportamiento cuando se envían datos no válidos:

```
$browser->
  info('  3.2 - Submit a Job with invalid values')->

  get('/job/new')->
  click('Preview your job', array('job' => array(
```

```
    'company'      => 'Sensio Labs',
    'position'     => 'Developer',
    'location'     => 'Atlanta, USA',
    'email'        => 'not.an.email',
  )))->

  with('form')->begin()->
    hasErrors(4)->
    isError('description', 'required')->
    isError('how_to_apply', 'required')->
    isError('email', 'invalid')->
  end()
;
```

Si se pasa un número entero al método `hasErrors()` se puede comprobar que existan exactamente ese número de errores en el formulario. Por su parte, el método `isError()` comprueba el código de error del campo indicado.

Sugerencia

En la prueba que hemos escrito para el caso en el que se envían datos no válidos, no hemos vuelto a probar el formulario entero. En este caso, sólo hemos añadido las pruebas necesarias para probar cosas muy específicas del formulario.

También es posible probar el código HTML generado para comprobar si contiene mensajes de error, pero en este caso no es necesario porque no hemos modificado la estructura del formulario.

A continuación vamos a probar la barra de administrador de la página de previsualización de una oferta de trabajo. Cuando una oferta de trabajo todavía no se ha activado, las acciones que se pueden realizar son editar, borrar y publicar la oferta. Para probar esos tres enlaces, en primer lugar tenemos que crear una oferta de trabajo. Como esto obligaría a copiar y pegar mucho código, vamos a añadir un método en la clase `JobeetTestFunctional` que se encargue de crear ofertas de trabajo:

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function createJob($values = array())
  {
    return $this->
      get('/job/new')->
      click('Preview your job', array('job' => array_merge(array(
        'company'      => 'Sensio Labs',
        'url'          => 'http://www.sensio.com/',
        'position'     => 'Developer',
        'location'     => 'Atlanta, USA',
        'description'  => 'You will work with symfony to develop websites for
our customers.',
        'how_to_apply' => 'Send me an email',
        'email'        => 'for.a.job@example.com',
        'is_public'    => false,
      ), $values)))->
```

```
      followRedirect()
    ;
  }

  // ...
}
```

El método `createJob()` crea una nueva oferta de trabajo, realiza la redirección y devuelve el objeto del navegador para no romper con la *interfaz fluida* de los métodos de pruebas. Si quieres también puedes pasar un array de valores que se combinan con los valores por defecto antes de enviar el formulario.

11.6. Indicando el método HTTP de un enlace

Ahora ya podemos probar el enlace *"Publish"* de forma sencilla:

```
$browser->info('  3.3 - On the preview page, you can publish the job')->
  createJob(array('position' => 'FOO1'))->
  click('Publish', array(), array('method' => 'put', '_with_csrf' => true))->

  with('propel')->begin()->
    check('JobeetJob', array(
      'position'     => 'FOO1',
      'is_activated' => true,
    ))->
  end()
;
```

Si te acuerdas del tutorial del día 10, el enlace *"Publish"* utiliza el método `PUT` de HTTP. Como los navegadores actuales no soportan las peticiones de tipo `PUT`, el helper `link_to()` convierte el enlace en un formulario con un poco de código JavaScript.

Como el navegador de pruebas no ejecuta código JavaScript, debemos indicar que el método es `PUT` pasándolo como tercer argumento del método `click()`. Además, el helper `link_to()` también incluye un token para realizar la protección frente a los ataques de tipo CSRF, por lo que debemos utilizar la opción `_with_csrf` para simular este token.

El proceso de probar el enlace *"Delete"* es muy similar:

```
$browser->info('  3.4 - On the preview page, you can delete the job')->
  createJob(array('position' => 'FOO2'))->
  click('Delete', array(), array('method' => 'delete', '_with_csrf' => true))->

  with('propel')->begin()->
    check('JobeetJob', array(
      'position' => 'FOO2',
    ), false)->
  end()
;
```

11.7. La seguridad que te dan las pruebas

Cuando la oferta de trabajo se publica, ya no es posible modificarla. Aunque el enlace *"Edit"* no se muestra en la página de previsualización, vamos a añadir algunas pruebas para asegurarnos del todo.

En primer lugar, añade otro argumento al método `createJob()` para permitir la publicación automática de una oferta de trabajo y crea un método llamado `getJobByPosition()` que devuelva una oferta de trabajo a partir del puesto de trabajo indicado:

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function createJob($values = array(), $publish = false)
  {
    $this->
      get('/job/new')->
      click('Preview your job', array('job' => array_merge(array(
        'company'      => 'Sensio Labs',
        'url'          => 'http://www.sensio.com/',
        'position'     => 'Developer',
        'location'     => 'Atlanta, USA',
        'description'  => 'You will work with symfony to develop websites for
our customers.',
        'how_to_apply' => 'Send me an email',
        'email'        => 'for.a.job@example.com',
        'is_public'    => false,
      ), $values)))->
      followRedirect()
    ;

    if ($publish)
    {
      $this->
        click('Publish', array(), array('method' => 'put', '_with_csrf' =>
true))->
        followRedirect()
      ;
    }

    return $this;
  }

  public function getJobByPosition($position)
  {
    $criteria = new Criteria();
    $criteria->add(JobeetJobPeer::POSITION, $position);

    return JobeetJobPeer::doSelectOne($criteria);
  }
```

```
  // ...
}
```

Si la oferta de trabajo está publicada, la página para editarla debe devolver un código de error `404`:

```
$browser->info('  3.5 - When a job is published, it cannot be edited anymore')->
  createJob(array('position' => 'FOO3'), true)->
  get(sprintf('/job/%s/edit', $browser->getJobByPosition('FOO3')->getToken()))->

  with('response')->begin()->
    isStatusCode(404)->
  end()
;
```

No obstante, si ejecutas las pruebas verás que el resultado no es el esperado, ya que ayer se nos olvidó añadir esta restricción de seguridad. Como acabas de comprobar, escribir pruebas es una forma excelente de descubrir errores en la aplicación porque te obliga a pensar en todos los posibles casos.

Solucionar este problema es muy sencillo, ya que sólo tenemos que redirigir al usuario a una página de error `404` cuando la oferta de trabajo está activada:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeEdit(sfWebRequest $request)
{
  $job = $this->getRoute()->getObject();
  $this->forward404If($job->getIsActivated());

  $this->form = new JobeetJobForm($job);
}
```

Aunque el código que hemos añadido es trivial, ¿puedes asegurar que este nuevo código no ha roto ninguna otra funcionalidad de la aplicación? Para asegurarte de ello podrías abrir el navegador y empezar a probar todas las posibles combinaciones para acceder a la página de editar una oferta. Otra alternativa mucho mejor para asegurarte de que el nuevo código no ha roto nada consiste en ejecutar las pruebas funcionales que acabas de crear. De esta forma, si el nuevo código produce errores en la aplicación, Symfony te lo mostrará en los mensajes de error de las pruebas.

11.8. Regresando al futuro en una prueba

Cuando una oferta de trabajo expira en menos de cinco días o si ya ha expirado, el usuario que la creó puede ampliar la validez de la oferta por otros 30 días a partir de la fecha actual.

Probar este requisito no es nada sencillo, ya que la fecha de expiración se establece automáticamente a dentro de 30 días cuando se crea la oferta de trabajo. Por tanto, cuando accedes a la página de una oferta de trabajo, no se visualiza el enlace para extener la validez de esa oferta. Aunque podrías modificar la fecha de expiración en la base de datos o podrías modificar la plantilla para que siempre muestre ese enlace, estas

soluciones no son más que chapuzas con las que es muy fácil equivocarse. Como ya habrás adivinado, vamos a escribir algunas pruebas para que hagan este trabajo por nosotros.

En primer lugar, añade una nueva ruta para el método `extend`:

```
# apps/frontend/config/routing.yml
job:
  class:   sfPropelRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: PUT, extend: PUT }
  requirements:
    token: \w+
```

A continuación, actualiza el código del enlace *"Extend"* en el elemento parcial `_admin`:

```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<?php if ($job->expiresSoon()): ?>
 - <?php echo link_to('Extend', 'job_extend', $job, array('method' => 'put'))
?> for another <?php echo sfConfig::get('app_active_days') ?> days
<?php endif; ?>
```

Después crea la acción `extend`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job validity has been
extend until %s.', $job->getExpiresAt('m/d/Y')));

  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

Tal y como espera la acción, el método `extend()` de `JobeetJob` devuelve el valor `true` si se ha ampliado la validez de la oferta de trabajo y `false` en cualquier otro caso:

```
// lib/model/JobeetJob.php
class JobeetJob extends BaseJobeetJob
{
  public function extend()
  {
    if (!$this->expiresSoon())
    {
      return false;
    }

    $this->setExpiresAt(time() + 86400 * sfConfig::get('app_active_days'));

    return $this->save();
```

```
  }

  // ...
}
```

Por último, añade el siguiente escenario a las pruebas:

```
$browser->info('  3.6 - A job validity cannot be extended before the job
expires soon')->
  createJob(array('position' => 'FOO4'), true)->
  call(sprintf('/job/%s/extend',
$browser->getJobByPosition('FOO4')->getToken()), 'put', array('_with_csrf' =>
true))->
  with('response')->begin()->
    isStatusCode(404)->
  end()
;

$browser->info('  3.7 - A job validity can be extended when the job expires
soon')->
  createJob(array('position' => 'FOO5'), true)
;

$job = $browser->getJobByPosition('FOO5');
$job->setExpiresAt(time());
$job->save();

$browser->
  call(sprintf('/job/%s/extend', $job->getToken()), 'put', array('_with_csrf'
=> true))->
  with('response')->isRedirected()
;

$job->reload();
$browser->test()->is(
  $job->getExpiresAt('y/m/d'),
  date('y/m/d', time() + 86400 * sfConfig::get('app_active_days'))
);
```

Este escenario de pruebas introduce algunos elementos nuevos:

- El método `call()` obtiene una URL utilizando un método HTTP diferente de `GET` o `POST`
- Después de que la acción actualice la oferta de trabajo, recargamos el objeto local mediante `$job->reload()`
- Al final utilizamos el objeto `lime` para probar de forma directa la fecha de expiración de la oferta

11.9. Seguridad de los formularios

11.9.1. La magia de la serialización de formularios

Los formularios de Propel son muy fáciles de utilizar porque realizan automáticamente la mayor parte del trabajo. Si quieres serializar o guardar un formulario en la base de datos, lo único que tienes que hacer es realizar una llamada al método `$form->save()`.

¿Cómo funciona este método? Básicamente, el método `save()` realiza los siguientes pasos:

- Iniciar una transacción (porque todos los formularios de Propel anidados se guardan de una vez)
- Procesar los valores enviados (ejecutando los métodos `update_NOMBRE_COLUMNA_Column()` si existen)
- Invocar el método `fromArray()` del objeto Propel para actualizar el valor de las columnas
- Guardar el objeto en la base de datos
- Realizar la transacción

11.9.2. Características de seguridad incluidas por defecto

El método `fromArray()` toma un array de valores y actualiza los valores de las columnas correspondientes. ¿No es esto un posible agujero de seguridad? ¿Y si alguien trata de enviar el valor de una columna para la que no tiene autorización? ¿Podría por ejemplo modifica el valor de la columna `token`?

Vamos a escribir una prueba para simular el envío de una oferta de trabajo con un campo llamado `token`:

```
// test/functional/frontend/jobActionsTest.php
$browser->
  get('/job/new')->
  click('Preview your job', array('job' => array(
    'token' => 'fake_token',
  )))->

  with('form')->begin()->
    hasErrors(7)->
    hasGlobalError('extra_fields')->
  end()
;
```

Si envías el formulario anterior te encontrarás con un error global de tipo `extra_fields`. El motivo es que por defecto los formularios no permiten incluir campos adicionales en los valores enviados. Este también es el motivo por el que todos los campos del formulario deben contar con un validador asociado.

Sugerencia

También puedes probar a enviar campos adicionales directamente desde el navegador gracias a herramientas como la extensión *Web Developer Toolbar* de Firefox.

Si quieres deshabilitar esta medida de seguridad, modifica el valor de la opción `allow_extra_fields` a `true`:

```
class MyForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->validatorSchema->setOption('allow_extra_fields', true);
  }
}
```

La prueba ahora sí que pasa satisfactoriamente, pero el valor del campo `token` se ha eliminado de los valores del campo. Así que todavía no es posible saltarse esta medida de seguridad. No obstante, si realmente quieres pasar ese valor, puedes establecer la opción `filter_extra_fields` a `false`:

```
$this->validatorSchema->setOption('filter_extra_fields', false);
```

Nota

Las pruebas creadas en esta sección son sólo para mostrar algunas de las opciones disponibles en el framework. Deberías borrarlas del proyecto Jobeet porque las pruebas no deben validar opciones de Symfony.

11.9.3. Protección frente a ataques XSS y CSRF

Durante el primer día creamos la aplicación `frontend` con el siguiente comando:

```
$ php symfony generate:app --escaping-strategy=on --csrf-secret=Unique$ecret
frontend
```

La opción `--escaping-strategy` activa la protección frente a ataques de tipo XSS. Esto significa que por defecto las plantillas aplican el mecanismo de escape a los valores de todas las variables. Si tratas por ejemplo de incluir código HTML en la descripción de una oferta de trabajo, verás que cuando Symfony muestra los detalles de la oferta, las etiquetas se ven tal y como están escritas y no se interpretan como etiquetas HTML.

Por su parte, la opción `--csrf-secret` activa la protección frente a ataques de tipo CSRF. Si activas esta opción, todos los formularios incluyen un campo oculto llamado `_csrf_token`.

Sugerencia

El tipo de mecanismo de escape que se aplica y el secreto de CSRF que se utiliza se pueden modificar en cualquier momento en el archivo de configuración `apps/frontend/config/`

`settings.yml`. Al igual que sucede con el archivo `databases.yml`, las opciones se pueden configurar para cada entorno de ejecución:

```
all:
  .settings:
    # Form security secret (CSRF protection)
    csrf_secret: Unique$ecret

     # Output escaping settings
    escaping_strategy: on
    escaping_method:   ESC_SPECIALCHARS
```

11.10. Tareas de mantenimiento

Aunque Symfony es un framework para desarrollar aplicaciones web, también incluye una herramienta para la línea de comandos. Esta herramienta ya la has utilizado para crear la estructura inicial de directorios del proyecto y de la aplicación y también para generar las clases del modelo de datos. Crear una nueva tarea es muy sencillo, ya que todas las herramientas necesarias se incluyen en el framework.

Cuando un usuario crea una nueva oferta de trabajo, es necesario que la active para que se publique en la web. Si no se activan las ofertas, la base de datos puede contener en poco tiempo muchas ofertas de trabajo inactivas. Por tanto, vamos a crear una tarea que elimina todas las ofertas de trabajo inactivas de la base de datos. Además, ejecutaremos esta tarea de forma periódica mediante una tarea programada.

```
// lib/task/JobeetCleanupTask.class.php
class JobeetCleanupTask extends sfBaseTask
{
  protected function configure()
  {
    $this->addOptions(array(
      new sfCommandOption('env', null, sfCommandOption::PARAMETER_REQUIRED,
'The environement', 'prod'),
      new sfCommandOption('days', null, sfCommandOption::PARAMETER_REQUIRED,
'', 90),
    ));

    $this->namespace = 'jobeet';
    $this->name = 'cleanup';
    $this->briefDescription = 'Cleanup Jobeet database';

    $this->detailedDescription = <<<EOF
The [jobeet:cleanup|INFO] task cleans up the Jobeet database:

  [./symfony jobeet:cleanup --env=prod --days=90|INFO]
EOF;
  }

  protected function execute($arguments = array(), $options = array())
  {
    $databaseManager = new sfDatabaseManager($this->configuration);
```

```
    $nb = JobeetJobPeer::cleanup($options['days']);

    $this->logSection('propel', sprintf('Removed %d stale jobs', $nb));
  }
}
```

La configuración de la tarea se realiza en el método `configure()`. Cada tarea debe tener un nombre único (`namespace:nombre`) y puede tener argumentos y opciones.

Sugerencia

Puedes echar un vistazo a las tareas que incluye Symfony (en el directorio `lib/task/`) para ver más ejemplos de uso.

La tarea `jobeet:cleanup` define dos opciones, `--env` y `--days`, que a su vez definen valores por defecto adecuados.

Las tareas propieas se ejecutan exactamente igual que cualquier otra tarea de Symfony:

```
$ php symfony jobeet:cleanup --days=10 --env=dev
```

Como siempre, el código que se encarga de limpiar la base de datos se ha incluido en la clase `JobeetJobPeer`:

```
// lib/model/JobeetJobPeer.php
static public function cleanup($days)
{
  $criteria = new Criteria();
  $criteria->add(self::IS_ACTIVATED, false);
  $criteria->add(self::CREATED_AT, time() - 86400 * $days, Criteria::LESS_THAN);

  return self::doDelete($criteria);
}
```

El método `doDelete()` elimina de la base de datos todos los registros que cumplen con los criterios de búsqueda del objeto `Criteria`. A este método también se le puede pasar un array de claves primarias.

Nota

Las tareas de Symfony devuelven un valor en función del éxito en la ejecución de la tarea. Si quieres devolver un valor específico, puedes hacerlo añadiendo al final de la tarea una instrucción `return` que devuelva un número entero.

11.11. Nos vemos mañana

Las pruebas son una de las partes fundamentales de las herramientas y filosofía de trabajo de Symfony. Hoy hemos aprendido cómo aprovechar las herramientas de Symfony para hacer que el desarrollo de una aplicación sea más sencillo, rápido y sobre todo, más seguro.

El framework de formularios de Symfony incluye mucho más que widgets y validadores, ya que proporciona una forma sencilla de probar los formularios y de asegurarte de que los formularios son seguros por defecto.

Nuestro recorrido por las mejores características de Symfony no finaliza hoy, ya que mañana vamos a crear la parte de administración de la aplicación Jobeet. La mayoría de proyectos web incluye una interfaz de administración y Jobeet también la va a incluir. ¿Pero cómo vamos a crear toda una interfaz de administración en una sola hora de trabajo? Muy fácilmente: utilizando el framework de generación de la parte de administración de las aplicaciones de Symfony.

Capítulo 12. El generador de la parte de administración

La aplicación `frontend` de Jobeet ya es completamente funcional tanto para los usuarios que buscan trabajo como para los que quiere publicar nuevas ofertas de trabajo. Por tanto, ahora ha llegado el momento de empezar a hablar de la parte de administración de la aplicación, que normalmente se conoce con el nombre de *backend*.

Durante el día de hoy, vamos a desarrollar en menos de una hora la parte de administración completa de la aplicación, gracias a las utilidades que incluye Symfony para generar automáticamente la interfaz de administración.

12.1. Creando la aplicación backend

Lo primero que tenemos que hacer es crear la aplicación `backend`. Si no te falla la memoria, te acordarás de que las aplicaciones de Symfony se crean con la tarea `generate:app`:

```
$ php symfony generate:app --escaping-strategy=on --csrf-secret=UniqueSecret1
backend
```

Aunque la aplicación `backend` sólo la van a utilizar los administradores de Jobeet, hemos activado todas las medidas de seguridad que incluye Symfony.

Sugerencia

Si quieres utilizar caracteres especiales en la contraseña de la opción `--csrf-secret`, como por ejemplo un signo de dólar (`$`), tienes que escapar cada carácter especial en la línea de comandos mediante la barra `\`:

```
$ php symfony generate:app --csrf-secret=Unique\$ecret backend
```

Después de ejecutar la tarea, ya puedes acceder a la nueva aplicación en http://jobeet.localhost/backend.php/ para el entorno de producción y en http://jobeet.localhost/backend_dev.php/ para el entorno de desarrollo.

Nota

Cuando creamos la aplicación `frontend`, el controlador frontal de producción se llamaba `index.php`. Como sólo se puede tener un archivo `index.php` en cada directorio, Symfony crea un archivo llamado `index.php` para el controlador frontal de la primera aplicación y el resto de controladores frontales se llaman igual que el resto de aplicaciones.

Si ahora intentas volver a cargar los archivos de datos con la tara `propel:data-load`, verás que ya no funciona. El motivo es que el método `JobeetJob::save()` debe tener acceso al archivo de configuración `app.yml` de la aplicación `frontend`. Como ahora

tenemos dos aplicaciones, Symfony utiliza el primer archivo `app.yml` que encuentra, que en este caso es el de la aplicación `backend`.

No obstante, como vimos durante el tutorial del día 8, las opciones de configuración se establecen en diferentes niveles. Si copias el contenido del archivo `apps/frontend/config/app.yml` al archivo `config/app.yml`, las opciones de configuración están disponibles en todas las aplicaciones del proyecto y por tanto, se corrige el error anterior. Realiza el cambio ahora porque el generador de la parte de administración utiliza mucho las clases del modelo y por tanto, también vamos a necesitar en la aplicación `backend` las variables definidas en el archivo `app.yml`.

Sugerencia

La tarea `propel:data-load` también permite el uso de la opción `--application`. De esta forma, si necesitas acceder a las opciones específicas de una aplicación, debes ejecutar la tarea con esta opción:

```
$ php symfony propel:data-load --application=frontend
```

12.2. Los módulos de la aplicación backend

En la aplicación `frontend` utilizamos la tarea `propel:generate-module` para generar automáticamente un módulo sencillo que permite realizar las opciones básicas sobre una clase del modelo. En la aplicación `backend` vamos a utilizar la tarea `propel:generate-admin` para generar una interfaz completa de administración para una clase del modelo:

```
$ php symfony propel:generate-admin backend JobeetJob --module=job
$ php symfony propel:generate-admin backend JobeetCategory --module=category
```

Los dos comandos anteriores crean respectivamente los módulos `job` y `category` para las clases del modelo `JobeetJob` y `JobeetCategory`.

La opción `--module` permite redefinir el nombre que la tarea genera por defecto para cada módulo (que en el caso de la clase `JobeetJob` hubiera sido `jobeet_job`).

La tarea `propel:generate-admin` también crea automáticamente una ruta propia para cada módulo:

```
# apps/backend/config/routing.yml
jobeet_job:
  class: sfPropelRouteCollection
  options:
    model:                JobeetJob
    module:               job
    prefix_path:          job
    column:               id
    with_wildcard_routes: true
```

Como era de esperar, el tipo de ruta que utiliza el generador de la parte de administración es `sfPropelRouteCollection`, ya que el objetivo de la interfaz de administración es la gestión completa de los objetos del modelo.

La definición de la ruta anterior también incluye algunas opciones que no habías visto hasta ahora:

- `prefix_path`: define el prefijo utilizado en las rutas generadas (en este ejemplo, la página de modificación de una oferta de trabajo será algo como `/job/1/edit`).
- `column`: define qué columna de la tabla se utiliza en las URL de los enlaces que hacen referencia a un objeto.
- `with_wildcard_routes`: como la interfaz de administración incluye muchas más posibilidades que las operaciones básicas (crear, actualizar, obtener y borrar objetos), esta opción permite definir más acciones sobre objetos y colecciones de objetos sin necesidad de modificar la ruta.

Sugerencia

Como siempre, es una buena idea leer la ayuda de una tarea antes de utilizarla:

```
$ php symfony help propel:generate-admin
```

La ayuda de Symfony muestra todos los argumentos y opciones de cada tarea y también muestra algunos ejemplos de uso.

12.3. El aspecto de la aplicación backend

Los módulos que se acaban de generar ya están listos para ser usados:

```
http://jobeet.localhost/backend_dev.php/job
http://jobeet.localhost/backend_dev.php/category
```

Los módulos de administración tienen muchas más funcionalidades que los módulos simples que hemos generado hasta el momento. Sin ni siquiera tener que escribir una sola línea de código PHP, cada módulo incluye las siguientes características:

- El listado de objetos muestra una **paginación**
- El listado se puede **ordenar**
- El listado se puede **filtrar**
- Se pueden **crear**, **modificar** y **borrar** objetos
- Se pueden borrar **varios** objetos **a la vez**
- Se aplica la **validación** en los formularios
- Se muestran **mensajes flash** para informar al usuario del resultado de las acciones
- ...y muchas otras características

El generador de la parte de administración incluye todas las características necesarias para crear una interfaz de administración en forma de módulos generados fácilmente configurables.

Si quieres mejorar la experiencia de usuario de la aplicación, tenemos que modificar el aspecto por defecto de la aplicación `backend`. Para facilitar la navegación entre los módulos de la aplicación, también vamos a añadir un sencillo menú de navegación.

Reemplaza el contenido por defecto de `layout.php` por el siguiente código:

```
// apps/backend/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>
    <title>Jobeet Admin Interface</title>
    <link rel="shortcut icon" href="/favicon.ico" />
    <?php use_stylesheet('admin.css') ?>
    <?php include_javascripts() ?>
    <?php include_stylesheets() ?>
  </head>
  <body>
    <div id="container">
      <div id="header">
        <h1>
          <a href="<?php echo url_for('@homepage') ?>">
            <img src="/images/logo.jpg" alt="Jobeet Job Board" />
          </a>
        </h1>
      </div>

      <div id="menu">
        <ul>
          <li>
            <?php echo link_to('Jobs', '@jobeet_job') ?>
          </li>
          <li>
            <?php echo link_to('Categories', '@jobeet_category') ?>
          </li>
        </ul>
      </div>

      <div id="content">
        <?php echo $sf_content ?>
      </div>

      <div id="footer">
        <img src="/images/jobeet-mini.png" />
        powered by <a href="http://www.symfony-project.org/">
        <img src="/images/symfony.gif" alt="symfony framework" /></a>
      </div>
    </div>
  </body>
</html>
```

Este layout utiliza una hoja de estilos llamada `admin.css`. La hoja de estilos debería encontrarse en el directorio `web/css/`, ya que la instalamos durante el día 4 junto con el resto de hojas de estilos.

Como hicimos en la aplicación `frontend`, hemos creado una hoja de estilos muy sencilla para la aplicación `backend`. Puedes descargar el archivo admin.css (http://svn.jobeet.org/tags/release_day_12/web/css/admin.css) directamente desde el repositorio de Subversion.

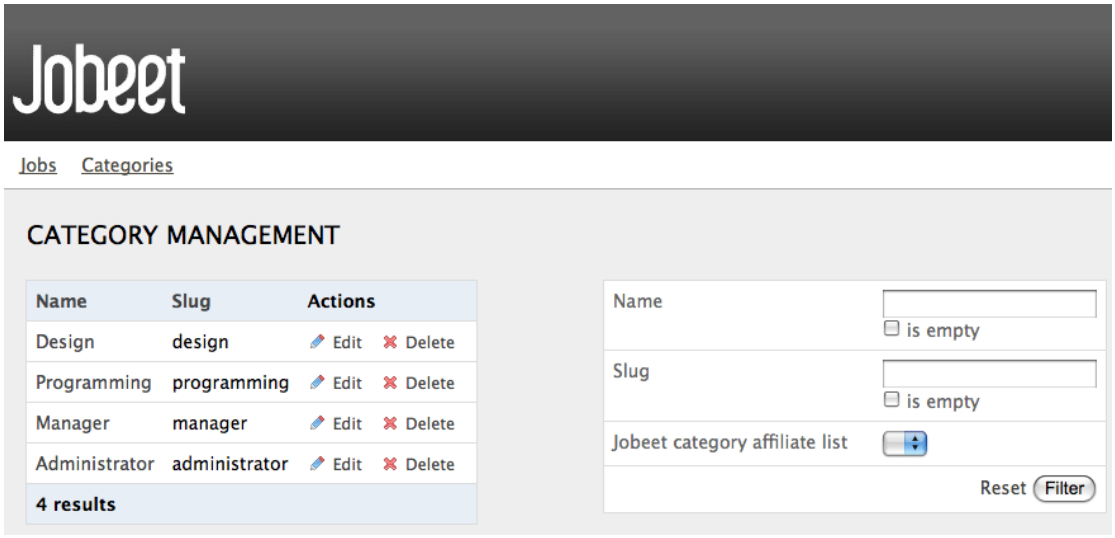

Figura 12.1. El aspecto del generador de la parte de administración

Por último, también puedes cambiar en el archivo `routing.yml` la portada por defecto de Symfony:

```
# apps/backend/config/routing.yml
homepage:
  url:   /
  param: { module: job, action: index }
```

12.4. La cache de Symfony

Si eres de los que sienten curiosidad por cómo funcionan las cosas, seguramente ya has abierto los archivos generados por la tarea `propel:generate-admin` en el directorio `apps/backend/modules/`. Si no lo habías hecho, este es el momento de hacerlo. ¡Sorpresa! Los directorios `templates` están vacíos y los archivos `actions.class.php` también están casi vacíos:

```
// apps/backend/modules/job/actions/actions.class.php
require_once dirname(__FILE__).'/../lib/jobGeneratorConfiguration.class.php';
require_once dirname(__FILE__).'/../lib/jobGeneratorHelper.class.php';

class jobActions extends autoJobActions
{
}
```

¿Cómo es posible que funcionen estos módulos? Si te fijas con atención, verás que la clase `jobActions` hereda de la clase `autoJobActions`. Si esta clase `autoJobActions` no existe, Symfony la genera automáticamente. En realidad, esta clase se encuentra en el directorio `cache/backend/dev/modules/autoJob/`, que contiene los archivos *verdaderos* del módulo:

```
// cache/backend/dev/modules/autoJob/actions/actions.class.php
class autoJobActions extends sfActions
{
  public function preExecute()
  {
    $this->configuration = new jobGeneratorConfiguration();

    if (!$this->getUser()->hasCredential(
      $this->configuration->getCredentials($this->getActionName())
    ))
    {

// ...
```

El funcionamiento del generador de la parte de administración te debería resultar familiar. En realidad, su funcionamiento es muy similar al de las clases del modelo y de los formularios. En base a la definición del esquema de datos, Symfony genera las clases del modelo y de los formularios. En el caso del generador de la parte de administración, el módulo generado automáticamente se configura modificando el archivo `config/generator.yml` que se encuentra dentro del propio módulo:

```
# apps/backend/modules/job/config/generator.yml
generator:
  class: sfPropelGenerator
  param:
    model_class:           JobeetJob
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_job
    with_propel_route:     1

    config:
      actions: ~
      fields:  ~
      list:    ~
      filter:  ~
      form:    ~
      edit:    ~
      new:     ~
```

Cada vez que modificas el archivo `generator.yml`, Symfony regenera su cache. Como veremos en el resto de secciones, personalizar un módulo de administración generado automáticamente es muy sencillo, rápido y hasta divertido.

Nota

La regeneración automática de los archivos de la cache sólo se realiza en el entorno de desarrollo. En el entorno de producción, debes borrar la cache manualmente mediante la tarea `cache:clear`.

12.5. La configuración de la aplicación backend

Los módulos de administración se pueden configurar añadiendo o modificando las opciones que se encuentran bajo la sección `config` del archivo `generator.yml`. La configuración se puede realizar en las siguientes siete secciones:

- `actions`: la configuración por defecto de las acciones que se encuentran en el listado y en los formularios
- `fields`: configuración por defecto de los campos de los formularios
- `list`: configuración del listado
- `filter`: configuración de los filtros
- `form`: configuración del formulario `new/edit`
- `edit`: configuración específica de la página `edit`
- `new`: configuración específica de la página `new`

A continuación vamos a empezar a personalizar los módulos de administración.

12.6. Configuración del título

El título de las secciones `list`, `edit` y `new` del módulo `category` se puede modificar estableciendo la opción `title`:

```
# apps/backend/modules/category/config/generator.yml
config:
  actions: ~
  fields:  ~
  list:
    title: Category Management
  filter:  ~
  form:    ~
  edit:
    title: Editing Category "%%name%%"
  new:
    title: New Category
```

La opción `title` de la sección `edit` contiene valores dinámicos: todas las cadenas de texto encerradas con `%%` se reemplazan por los valores correspondientes a esa columna del registro de la base de datos al que representa el objeto.

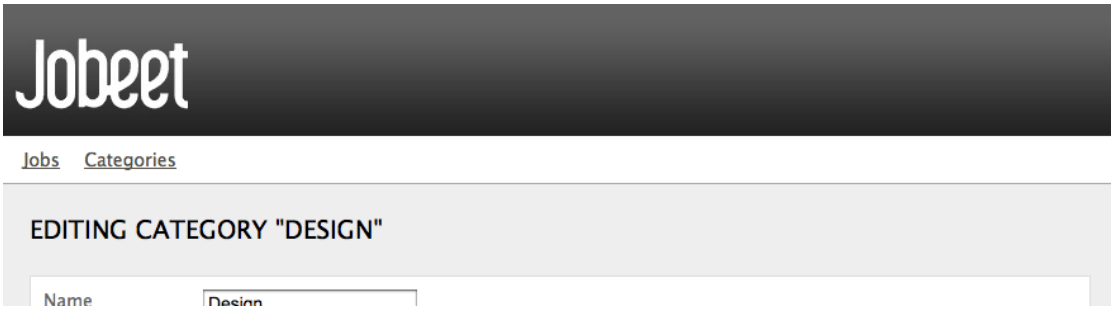

Figura 12.2. Personalizando los títulos

La configuración para el módulo `job` es muy similar:

```
# apps/backend/modules/job/config/generator.yml
config:
  actions: ~
  fields:  ~
  list:
    title: Job Management
  filter:  ~
  form:    ~
  edit:
    title: Editing Job "%%company%% is looking for a %%position%%"
  new:
    title: Job Creation
```

12.7. Configuración de los campos

Las diferentes vistas (`list`, `new` y `edit`) están compuestas por campos. Un campo puede ser una columna de una clase del modelo o una columna virtual, tal y como veremos más adelante.

La sección `fields` del archivo de configuración permite personalizar la configuración por defecto de los campos:

```
# apps/backend/modules/job/config/generator.yml
config:
  fields:
    is_activated: { label: Activated?, help: Whether the user has activated the
job, or not }
    is_public:    { label: Public? }
```

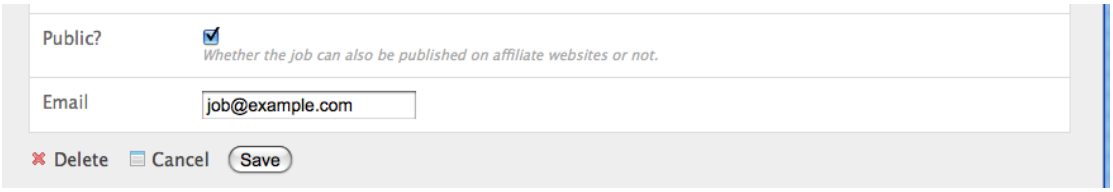

Figura 12.3. Aspecto de los campos configurados

La sección `fields` redefine la configuración de los campos de todas las páginas, lo que significa que el valor de la opción `label` del campo `is_activated` se modifica para las páginas `list`, `edit` y `new`.

La configuración del generador de la parte de administración se basa en el principio de configuración en cascada. Si quieres modificar por ejemplo la opción `label` sólo para la página `list`, debes definir una opción llamada `fields` bajo la sección `list`:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    fields:
      is_public:    { label: "Public? (label for the list)" }
```

Cualquier configuración realizada en la sección `fields` principal se puede redefinir en la configuración específica de cada página. Las reglas que se siguen en la configuración en cascada son las siguientes:

- `new` y `edit` heredan de `form` que a su vez hereda de `fields`
- `list` hereda de `fields`
- `filter` hereda de `fields`

Nota

En las secciones de formularios (`form`, `edit` y `new`), las opciones `label` y `help` redefinen el valor de las mismas opciones establecidas en las clases de los formularios.

12.8. Configuración de la página list

12.8.1. La opción display

La página del listado muestra por defecto todas las columnas del modelo, en el mismo orden en el que se indicaron en el archivo del esquema. La opción `display` establece las columnas que se muestran y el orden en el que lo hacen:

```
# apps/backend/modules/category/config/generator.yml
config:
  list:
    title:   Category Management
    display: [=name, slug]
```

El símbolo `=` delante de la columna `name` es una convención que indica que se debe convertir la cadena de texto en un enlace.

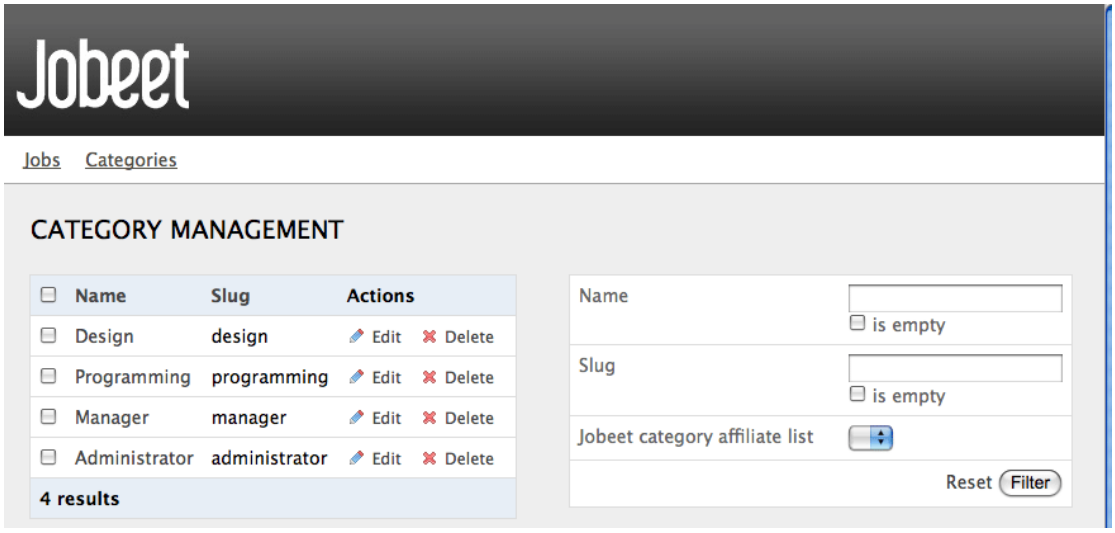

Figura 12.4. La tabla del listado

A continuación se realiza la misma configuración en el módulo `job` para hacerlo más fácil de leer:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    title:   Job Management
    display: [company, position, location, url, is_activated, email]
```

12.8.2. La opción layout

Los listados se pueden mostrar con diferentes layouts. El layout por defecto es `tabular`, que muestra el valor de cada columna en su propia columna de la tabla. No obstante, en el módulo `job` sería mejor utilizar el layout `stacked`, que es el otro layout que incluye Symfony:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    title:   Job Management
    layout:  stacked
    display: [company, position, location, url, is_activated, email]
    params:  |
      %%is_activated%% <small>%%category_id%%</small> - %%company%%
       (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
```

En el layout `stacked`, cada objeto se representa en una sola cadena de texto, cuyo formato se define en la opción `params`.

Nota

En el ejemplo anterior, la opción `display` sigue siendo necesaria porque define las columnas por las que el usuario puede reordenar los resultados.

12.8.3. Columnas virtuales

Si se utiliza la configuración anterior, el fragmento `%%category_id%%` se reemplaza por el valor de la clave primaria de la categoría. Sin embargo, en este caso sería más útil mostrar el nombre de la categoría.

Cuando se hace uso de la notación `%%`, la variable indicada no tiene que ser obligatoriamente una columna real de la base de datos. Para mostrar el valor de una variable, lo único que necesita el generador de la parte de administración es un método *getter* en la clase del modelo.

Si queremos mostrar el nombre de una categoría, podemos crear un método llamado `getCategoryName()` en la clase `JobeetJob` y reemplazar el fragmento `%%category_id%%` por `%%category_name%%`.

Por otra parte, la clase `JobeetJob` ya dispone de un método llamado `getJobeetCategory()` y que devuelve el objeto de la categoría relacionada. Por tanto, si utilizas `%%jobeet_category%%`, ya se va a mostrar el nombre de la categoría, ya que la clase `JobeetCategory` incluye un método mágico `__toString()` que convierte un objeto en una cadena de texto.

```
# apps/backend/modules/job/config/generator.yml
%%is_activated%% <small>%%jobeet_category%%</small> - %%company%%
 (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
```

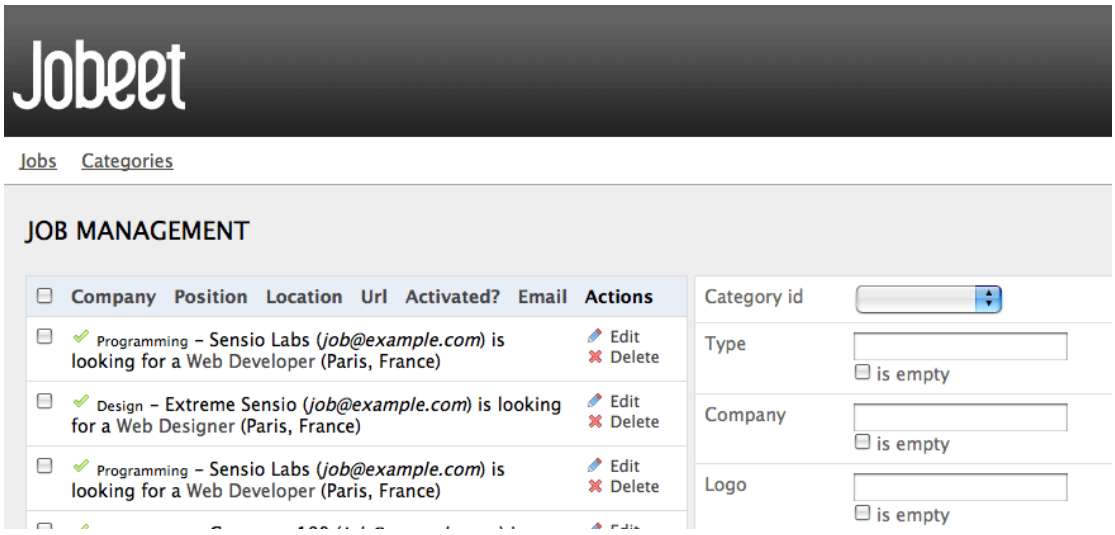

Figura 12.5. El layout stacked

12.8.4. La opción sort

Si eres un administrador, seguramente querrás ver las últimas ofertas de trabajo publicadas. Para configurar la columna por la que se ordenan los datos por defecto, incluye la opción `sort` indicando el nombre de la columna y el tipo de ordenación:

```
# apps/backend/modules/job/config/generator.yml
config:
```

```
  list:
    sort: [expires_at, desc]
```

12.8.5. La opción max_per_page

El listado incluye por defecto una paginación que muestra 20 elementos en cada página. Este valor se puede modificar con la opción `max_per_page`:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    max_per_page: 10
```

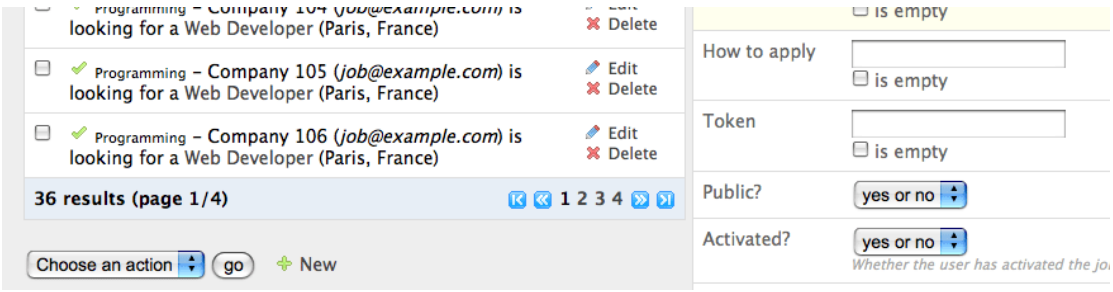

Figura 12.6. Modificando el máximo número de elementos por página

12.8.6. La opción batch_actions

En un listado se puede ejecutar una misma acción sobre varios objetos a la vez. Estas acciones por lotes no se necesitan en el módulo `category`, por lo que podemos eliminarlas:

```
# apps/backend/modules/category/config/generator.yml
config:
  list:
    batch_actions: {}
```

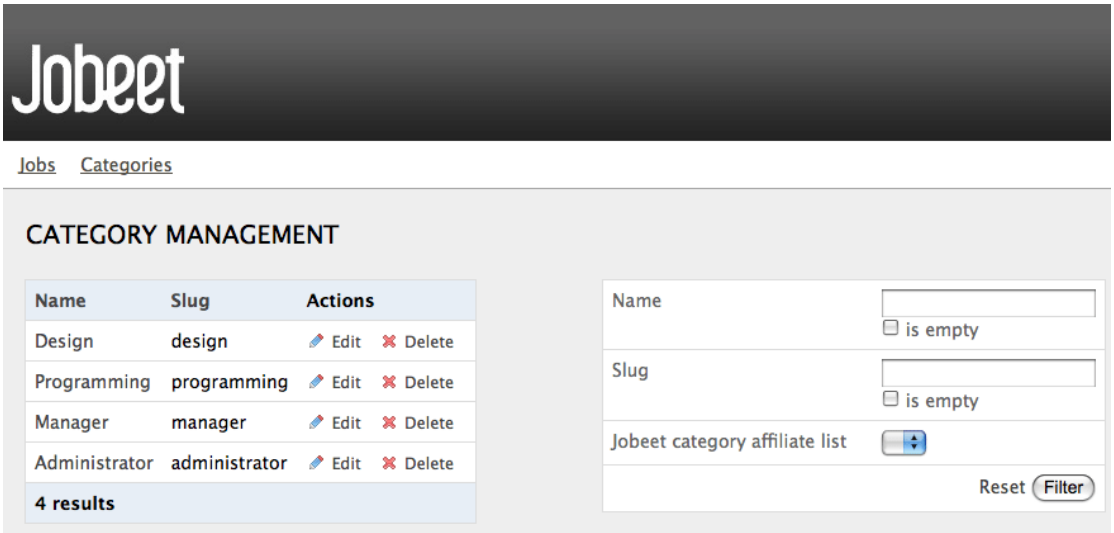

Figura 12.7. Eliminando las acciones por lotes

La opción `batch_actions` define la lista de acciones que se pueden realizar por lotes. Para eliminar esta opción, simplemente se indica un array vacío.

Por defecto cada módulo dispone de una acción de borrado por lotes llamada `delete` y que define el propio framework. Vamos a suponer que para el módulo `job` necesitamos además una acción por lotes que permita extender la validez de varias ofertas de trabajo por otros 30 días:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    batch_actions:
      _delete:    ~
      extend:     ~
```

Las acciones cuyo nombre comienza por `_` son acciones que incluye el propio framework. Si refrescas la página en el navegador y seleccionas la acción *Extend*, Symfony lanza una excepción que indica que debes crear un método llamado `executeBatchExtend()`:

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeBatchExtend(sfWebRequest $request)
  {
    $ids = $request->getParameter('ids');

    $jobs = JobeetJobPeer::retrieveByPks($ids);

    foreach ($jobs as $job)
    {
      $job->extend(true);
    }

    $this->getUser()->setFlash('notice', 'The selected jobs have been extended
successfully.');

    $this->redirect('@jobeet_job');
  }
}
```

Las claves primarias de los elementos seleccionados se almacenan en el parámetro `ids` de la petición. Una vez obtenidas las claves primarias, se ejecuta para cada oferta de trabajo seleccionada el método `JobeetJob::extend()` con un argumento adicional que permite saltarse la comprobación de la fecha de expiración que realiza ese método.

Actualiza el método `extend()` pra que tenga en cuenta este nuevo parámetro:

```
// lib/model/JobeetJob.php
class JobeetJob extends BaseJobeetJob
{
  public function extend($force = false)
  {
    if (!$force && !$this->expiresSoon())
    {
      return false;
    }
```

```
    $this->setExpiresAt(time() + 86400 * sfConfig::get('app_active_days'));
    $this->save();

    return true;
  }

  // ...
}
```

Una vez aumentada la validez de todas las ofertas de trabajo, se redirige al usuario a la portada del módulo `job`:

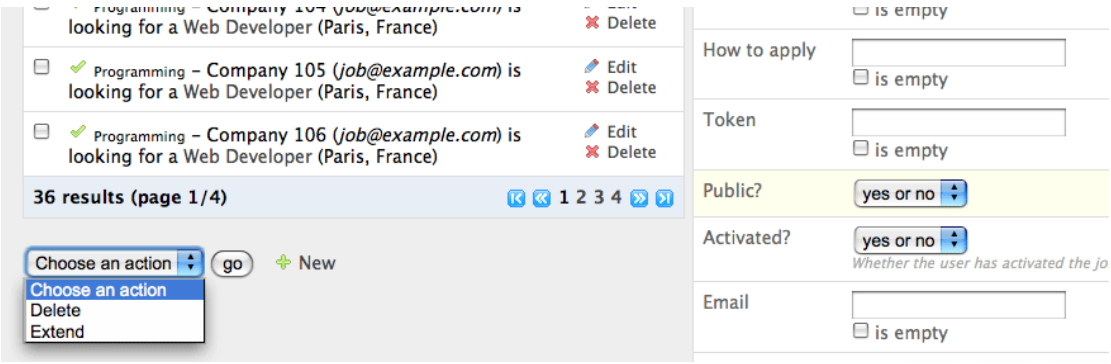

Figura 12.8. Acciones por lotes propias

12.8.7. La opción object_actions

En el listado de elementos siempre se muestra una columna adicional que contiene las acciones que se pueden realizar sobre un objeto individual. En el módulo `category` no necesitamos estas acciones porque ya disponemos del nombre de la categoría que es un enlace a la página de modificación de datos y porque tampoco necesitamos borrar una categoría directamente desde el listado:

```
# apps/backend/modules/category/config/generator.yml
config:
  list:
    object_actions: {}
```

En el módulo `job` vamos a dejar todas las acciones existentes y vamos a añadir una nueva acción llamada `extend` que es similar a la que acabamos de crear como acción por lotes:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    object_actions:
      extend:     ~
      _edit:      ~
      _delete:    ~
```

Como sucede para las acciones por lotes, las acciones `_delete` y `_edit` son acciones que define el propio framework, ya que su nombre empieza por _. Para que la acción `extend` se pueda utilizar, debemos definir la acción `listExtend()`:

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeListExtend(sfWebRequest $request)
  {
    $job = $this->getRoute()->getObject();
    $job->extend(true);

    $this->getUser()->setFlash('notice', 'The selected jobs have been extended
successfully.');

    $this->redirect('@jobeet_job');
  }

  // ...
}
```

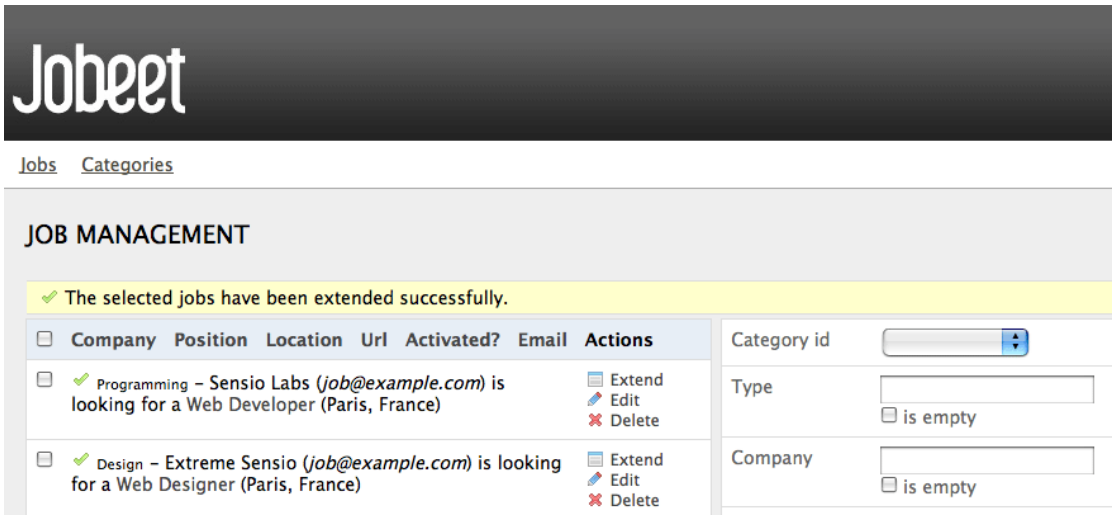

Figura 12.9. Creando una acción propia para los objetos

12.8.8. La opción actions

En las secciones anteriores se ha mostrado cómo añadir acciones por lotes y acciones que afectan a un solo objeto. Por su parte, la opción `actions` define las acciones que no utilizan ningún objeto, como la acción para crear un nuevo objeto. A continuación vamos a eliminar la opción `new` incluida por defecto y vamos a añadir una acción que borre todas las ofertas de trabajo que llevan más de 60 días sin ser activadas por parte del usuario que las insertó:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    actions:
      deleteNeverActivated: { label: Delete never activated jobs }
```

Hasta ahora, todas las acciones las hemos definido mediante ~, lo que significa que Symfony configura automáticamente esas acciones. Cada acción se puede personalizar pasándole un array de parámetros. La opción `label` redefine la etiqueta generada por defecto por Symfony.

Por defecto, la acción que se ejecuta cuando pinchas el enlace es el nombre de la acción prefijado con `list`.

Crea la acción `listDeleteNeverActivated` en el módulo `job`:

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeListDeleteNeverActivated(sfWebRequest $request)
  {
    $nb = JobeetJobPeer::cleanup(60);

    if ($nb)
    {
      $this->getUser()->setFlash('notice', sprintf('%d never activated jobs
have been deleted successfully.', $nb));
    }
    else
    {
      $this->getUser()->setFlash('notice', 'No job to delete.');
    }

    $this->redirect('@jobeet_job');
  }

  // ...
}
```

Como ya te habrás dado cuenta, hemos reutilizado el método `JobeetJobPeer::cleanup()` que definimos ayer. Este es otro ejemplo de las posibilidades de reutilización de código que nos brinda el patrón de diseño MVC.

Nota

También puedes modificar la acción que se ejecuta mediante el parámetro `action`:

```
deleteNeverActivated: { label: Delete never activated jobs, action: foo }
```

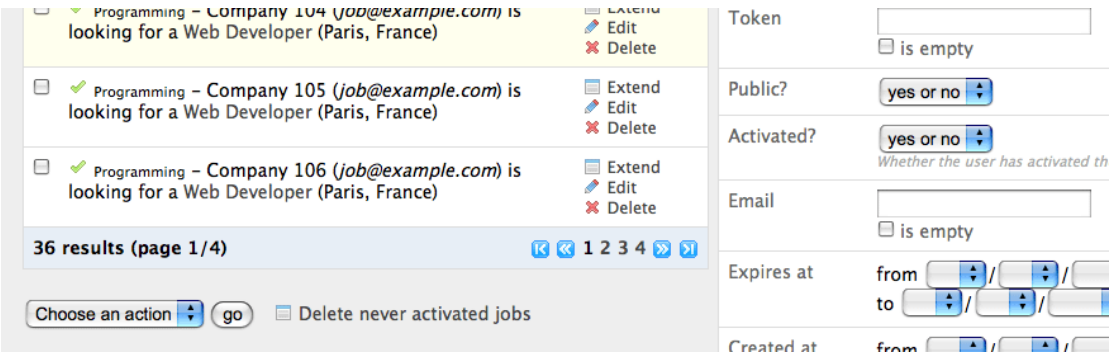

Figura 12.10. Acciones propias

12.8.9. La opción peer_method

Como muestra la barra de depuración web, se necesitan 14 consultas a la base de datos para mostrar el listado de ofertas de trabajo:

Si pinchas sobre ese número, verás que la mayoría de consultas se utilizan para obtener el nombre de la categoría de cada oferta de trabajo:

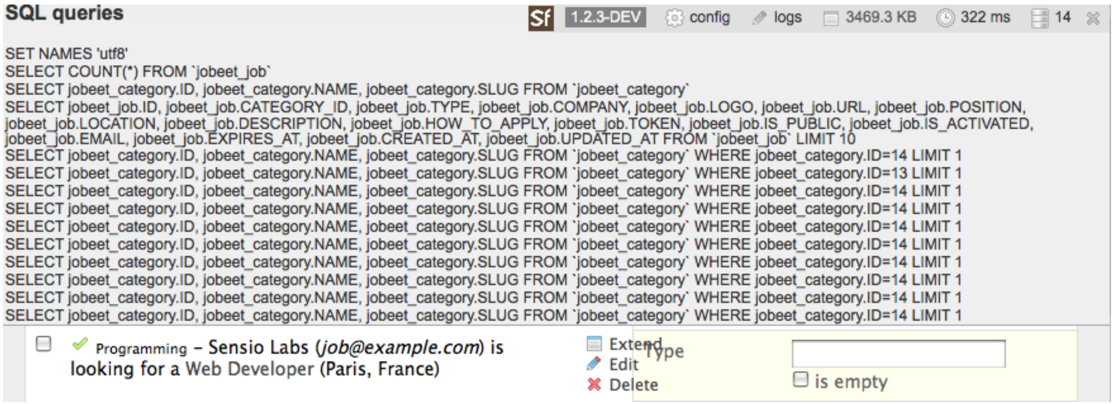

Figura 12.11. Número inicial de consultas

Si quieres reducir el número de consultas, en la opción `peer_method` puedes modificar el método por defecto que se emplea para obtener las ofertas de trabajo:

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    peer_method: doSelectJoinJobeetCategory
```

El método `doSelectJoinJobeetCategory()` añade un `JOIN` entre las tablas `job` y `category` para crear de forma automática el objeto de tipo categoría relacionado con cada oferta de trabajo.

Ahora el número de consultas se ha reducido a sólo cuatro:

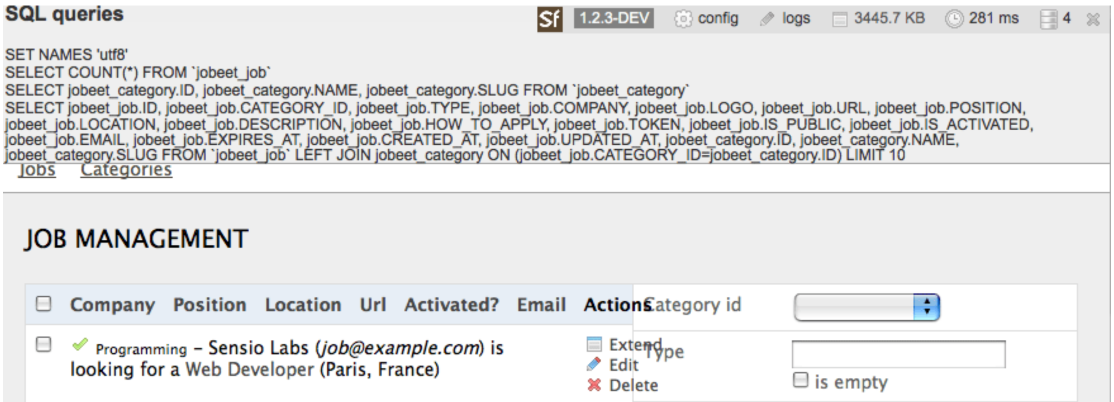

Figura 12.12. Número final de consultas

12.9. Configuración de la página de formularios

La configuración de las páginas de los formularios se realiza en tres secciones: `form`, `edit` y `new`. Todas tienen las mismas opciones de configuración y la sección `form` sólo existe por si no existen las secciones `edit` y `new`.

12.9.1. La opción display

Al igual que en el listado, si quieres modificar el orden en el que se muestran los campos, puedes utilizar la opción `display`. No obstante, como el formulario que se muestra está definido en una clase, no intentes quitar un campo porque podrían producirse errores de validación inesperados.

La opción `display` de las páginas de formularios también se puede utilizar para agrupar los campos:

```
# apps/backend/modules/job/config/generator.yml
config:
  form:
    display:
      Content: [category_id, type, company, logo, url, position, location,
description, how_to_apply, is_public, email]
      Admin:   [_generated_token, is_activated, expires_at]
```

La configuración anterior define dos grupos (`Content` y `Admin`), cada uno de los cuales contiene un subconjunto de campos de formulario.

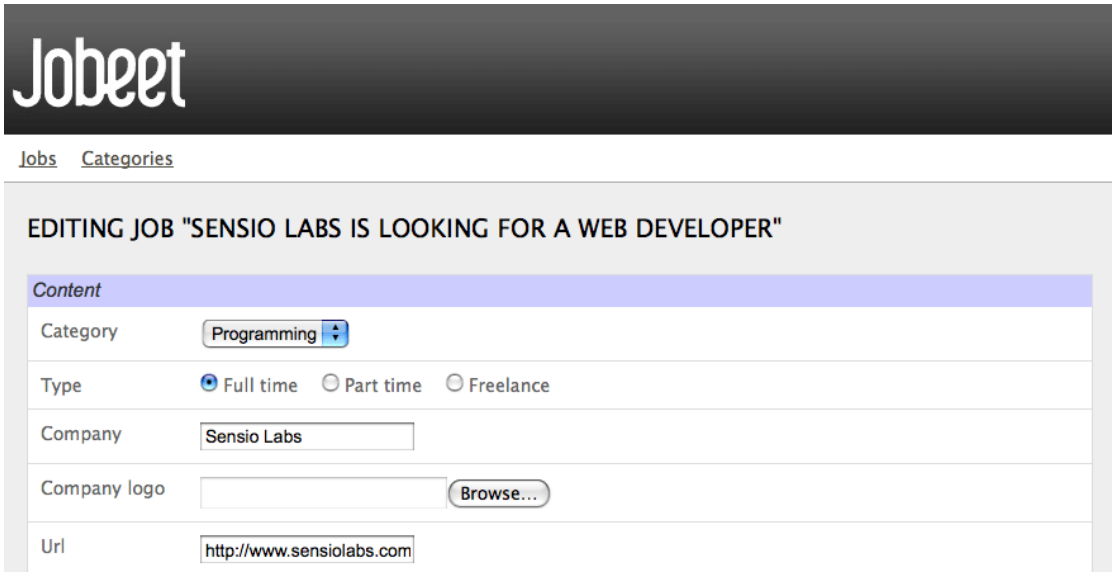

Figura 12.13. Agrupación de campos

Nota

Las columnas del grupo `Admin` todavía no se muestran en el navegador porque han sido eliminadas en la definición del formulario. Estas columnas aparecerán en algunas secciones cuando definamos una clase propia para el formulario `job` de la aplicación de administración.

El generador de la parte de administración incluye soporte para las relaciones muchos-a-muchos entre tablas de la base de datos. En el formulario para categorías, se muestra un cuadro de texto para el nombre, otro para el *slug* y una lista desplegable para los afiliados relacionados. Como no tiene sentido modificar esta relación en esta página, vamos a eliminarla:

```
// lib/form/JobeetCategoryForm.class.php
class JobeetCategoryForm extends BaseJobeetCategoryForm
{
  public function configure()
  {
    unset($this['jobeet_category_affiliate_list']);
  }
}
```

12.9.2. Columnas virtuales

En la opción `display` del formulario, el nombre del campo `_generated_token` comienza por un guión bajo (_). Esto significa que la forma en la que se muestra por pantalla este campo se controla mediante un elemento parcial llamado `_generated_token.php`.

Crea este elemento parcial con el siguiente contenido:

```
// apps/backend/modules/job/templates/_generated_token.php
<div class="sf_admin_form_row">
  <label>Token</label>
  <?php echo $form->getObject()->getToken() ?>
</div>
```

En este elemento parcial se puede acceder al formulario actual mediante la variable `$form` y el objeto relacionado se puede obtener mediante el método `getObject()`.

Nota

Si quieres utilizar un componente en vez de un elemento parcial para mostrar ese campo, puedes prefijar el nombre del campo con el símbolo ~

12.9.3. La opción class

Como este formulario lo van a utilizar los administradores, hemos mostrado más información que la que incluye el formulario que utilizan los usuarios normales. Sin embargo, por el momento el formulario no muestra parte de la información porque se ha eliminado en la clase `JobeetJobForm`.

Para utilizar diferentes formularios en la aplicación `frontend` y en la aplicación `backend`, tenemos que crear dos clases para ese formulario. Vamos a crear una clase `BackendJobeetJobForm` que herede de la clase `JobeetJobForm`. Como no vamos a tener los mismos campos ocultos, tenemos que refactorizar un poco la clase `JobeetJobForm` para mover la instrucción `unset()` a un método que sea redefinido en la clase `BackendJobeetJobForm`:

```
// lib/form/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    $this->removeFields();

    $this->validatorSchema['email'] = new sfValidatorEmail();

    // ...
  }

  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated'],
      $this['token']
    );
  }
}
// lib/form/BackendJobeetJobForm.class.php
class BackendJobeetJobForm extends JobeetJobForm
{
  public function configure()
  {
    parent::configure();
  }

  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['token']
    );
  }
}
```

La opción `class` permite redefinir la clase de formulario utilizada por el generador de la parte de administración:

```
# apps/backend/modules/job/config/generator.yml
config:
  form:
    class: BackendJobeetJobForm
```

Nota

Como acabamos de añadir una nueva clase, no te olvides de borrar la cache.

El formulario `edit` todavía tiene un pequeño inconveniente. El logotipo que se ha subido no se muestra en ninguna parte y tampoco se puede eliminar. El widget `sfWidgetFormInputFileEditable` añade estas opciones de modificación a cualquier campo simple que permita adjuntar archivos:

```
// lib/form/BackendJobeetJobForm.class.php
class BackendJobeetJobForm extends JobeetJobForm
{
  public function configure()
  {
    parent::configure();

    $this->widgetSchema['logo'] = new sfWidgetFormInputFileEditable(array(
      'label'     => 'Company logo',
      'file_src'  => '/uploads/jobs/'.$this->getObject()->getLogo(),
      'is_image'  => true,
      'edit_mode' => !$this->isNew(),
      'template'  => '<div>%file%<br />%input%<br />%delete%
%delete_label%</div>',
    ));

    $this->validatorSchema['logo_delete'] = new sfValidatorPass();
  }

  // ...
}
```

El widget `sfWidgetFormInputFileEditable` utiliza diversas opciones para configurar sus características y la forma en la que se muestra:

- `file_src`: la ruta web del archivo subido
- `is_image`: si vale `true`, el archivo se muestra como una imagen
- `edit_mode`: indica si el formulario se encuentra o no en el modo de edición
- `with_delete`: indica si se muestra el *checkbox* que permite borrar el archivo
- `template`: define la plantilla utilizada para mostrar el widget

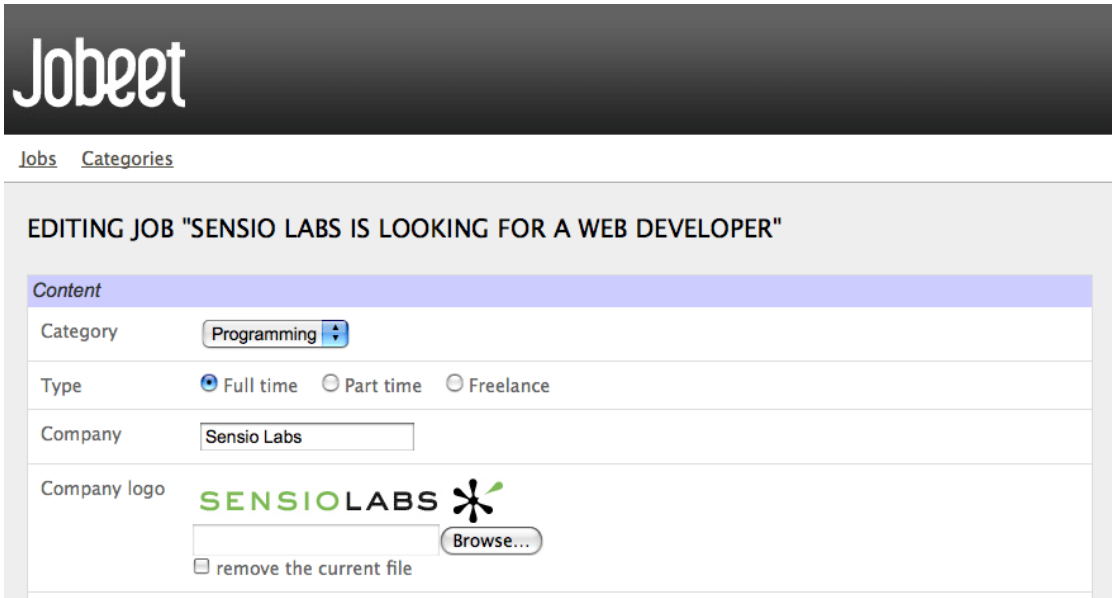

Figura 12.14. Subiendo un archivo

Sugerencia

El aspecto del generador de la parte de administración se puede configurar fácilmente porque las plantillas generadas incluyen muchos atributos `class` e `id`. El campo `logo` por ejemplo se puede modificar utilizando la clase `sf_admin_form_field_logo`. Cada campo también tiene un atributo `class` dependiente del tipo de campo, como por ejemplo `sf_admin_text` o `sf_admin_boolean`.

La opción `edit_mode` utiliza el método `sfPropel::isNew()`, que devuelve `true` si el objeto del formulario es nuevo y `false` en cualquier otro caso. Este método es muy útil cuando tienes diferentes widgets y validadores dependiendo del estado del objeto incluido.

12.10. Configuración de los filtros

Configurar los filtros es muy parecido a configurar las páginas de los formularios. De hecho, los filtros son simplemente formularios. Al igual que los formularios, las clases de los filtros se generan mediante la tarea `propel:build-all`. Si quieres volver a generar sólo los filtros, puedes utilizar la tarea `propel:build-filters`.

Las clases de los filtros de los formularios se encuentran en el directorio `lib/filter/` y cada clase del modelo dispone de una clase de filtros asociada (por ejemplo, `JobeetJobFormFilter` para el formulario `JobeetJobForm`).

Para el módulo `category` vamos a eliminar completamente los filtros:

```
# apps/backend/modules/category/config/generator.yml
config:
  filter:
    class: false
```

Para el módulo `job`, vamos a eliminar sólo algunos de ellos:

```
# apps/backend/modules/job/config/generator.yml
filter:
  display: [category_id, company, position, description, is_activated,
is_public, email, expires_at]
```

Como los filtros siempre son opcionales, no es necesario redefinir la clase de los filtros del formulario para configurar los campos que se muestran.

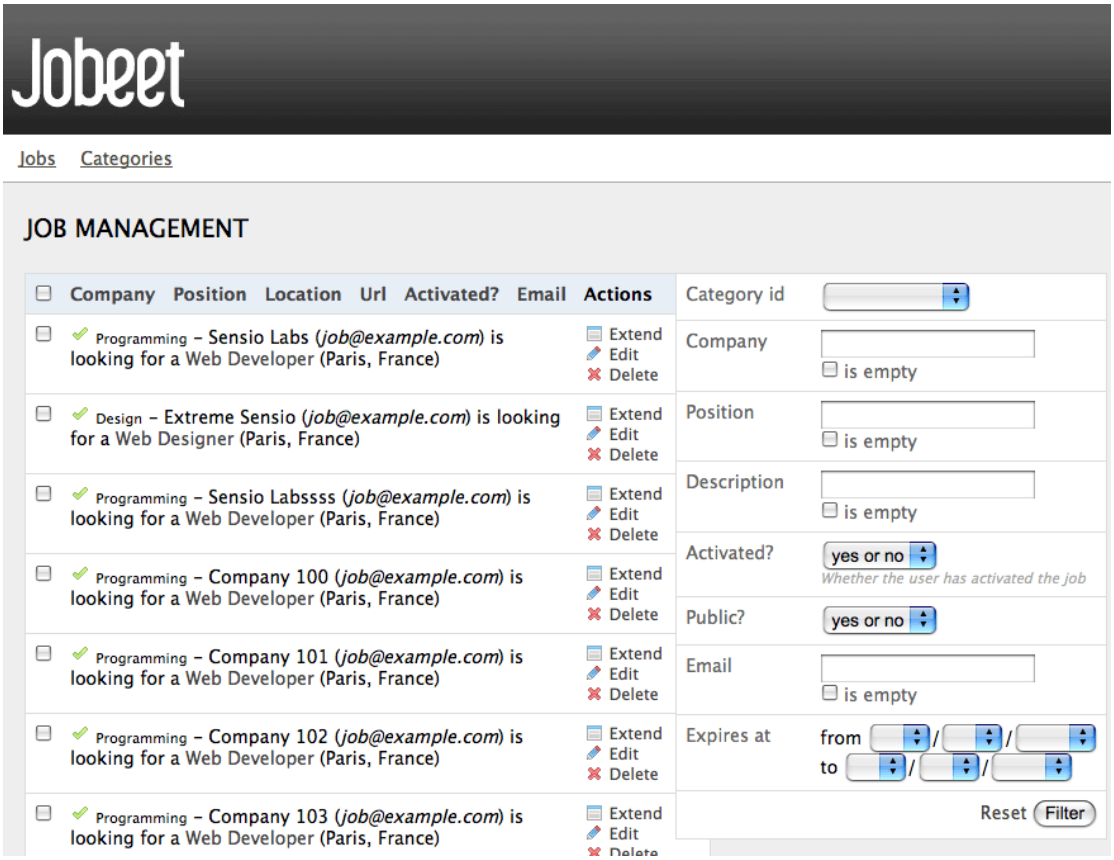

Figura 12.15. Los filtros

12.11. Modificando las acciones

Cuando configurar los módulos de administración no es suficiente, puedes añadir nuevos métodos a la clase de la acción tal y como hemos visto anteriormente al añadir la funcionalidad `extend`. Además, también puedes redefinir los métodos generados automáticamente en las acciones:

Método	Descripción
`executeIndex()`	La acción de la página `list`
`executeFilter()`	Actualiza los filtros
`executeNew()`	La acción de la página `new`
`executeCreate()`	Crea una nueva oferta de trabajo
`executeEdit()`	La acción de la página `edit`
`executeUpdate()`	Actualiza una oferta de trabajo
`executeDelete()`	Borra una oferta de trabajo
`executeBatch()`	Ejecuta una acción por lotes
`executeBatchDelete()`	Ejecuta la acción por lotes `_delete`
`processForm()`	Procesa el formulario de las ofertas de trabajo
`getFilters()`	Devuelve los filtros actuales

`setFilters()`	Establece los filtros
`getPager()`	Devuelve el paginador del listado
`getPage()`	Obtiene la página actual del listado
`setPage()`	Establece la página actual del listado
`buildCriteria()`	Define el objeto `Criteria` utilizado en el listado
`addSortCriteria()`	Añade el objeto `Criteria` utilizado para ordenar el listado
`getSort()`	Devuelve la columna utilizada para la ordenación actual
`setSort()`	Establece la columna utilizada para la ordenación actual

Como cada método generado automáticamente sólo realiza una tarea sencilla, es muy fácil modificar su comportamiento sin tener que copiar y pegar mucho código.

12.12. Personalizando las plantillas

Hemos visto en las secciones anteriores cómo modificar las plantillas generadas gracias a los atributos `class` e `id` que añade el generador de la parte de administración en el código HTML.

Además, las plantillas originales también se pueden redefinir completamente. Como las plantillas son archivos PHP y no clases PHP, una plantilla se puede redefinir simplemente creando en el módulo una plantilla con ese mismo nombre (por ejemplo en el directorio `apps/backend/modules/job/templates/` para el módulo `job`):

Plantilla	**Descripción**
`_assets.php`	Incluye los archivos CSS y JavaScript que se utilizan en las plantillas
`_filters.php`	Muestra la caje con los filtros
`_filters_field.php`	Muestra un campo de un filtro
`_flashes.php`	Muestra los mensajes flash
`_form.php`	Muestra el formulario
`_form_actions.php`	Muestra las acciones del formulario
`_form_field.php`	Muestra un campo de formulario
`_form_fieldset.php`	Muestra un *fieldset* de formulario
`_form_footer.php`	Muestra el pie de página de un formulario
`_form_header.php`	Muestra la cabecera de un formulario
`_list.php`	Muestra un listado
`_list_actions.php`	Muestra las acciones del listado
`_list_batch_actions.php`	Muestra las acciones por lotes del listado
`_list_field_boolean.php`	Muestra un campo de tipo *booleano* en el listado

`_list_footer.php`	Muestra el pie de página del listado
`_list_header.php`	Muestra la cabecera del listado
`_list_td_actions.php`	Muestra las acciones del objeto en una fila del listado
`_list_td_batch_actions.php`	Muestra el *checkbox* de una fila del listado
`_list_td_stacked.php`	Muestra el layout `stacked` para una fila del listado
`_list_td_tabular.php`	Muestra un campo del listado
`_list_th_stacked.php`	Muestra el nombre de una columna en la cabecera
`_list_th_tabular.php`	Muestra el nombre de una columna en la cabecera
`_pagination.php`	Muestra la paginación del listado
`editSuccess.php`	Muestra la página `edit`
`indexSuccess.php`	Muestra la página `list`
`newSuccess.php`	Muestra la página `new`

12.13. Configuración final

A continuación se muestra completa la configuración final de la parte de administración del proyecto Jobeet:

```
# apps/backend/modules/job/config/generator.yml
generator:
  class: sfPropelGenerator
  param:
    model_class:           JobeetJob
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_job
    with_propel_route:     1

    config:
      actions: ~
      fields:
        is_activated: { label: Activated?, help: Whether the user has activated
the job, or not }
        is_public:    { label: Public? }
      list:
        title:          Job Management
        layout:         stacked
        display:        [company, position, location, url, is_activated, email]
        params:  |
          %%is_activated%% <small>%%jobeet_category%%</small> - %%company%%
           (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
        max_per_page:  10
        sort:          [expires_at, desc]
        batch_actions:
```

```
          _delete:    ~
          extend:     ~
        object_actions:
          extend:     ~
          _edit:      ~
          _delete:    ~
        actions:
          deleteNeverActivated: { label: Delete never activated jobs }
        peer_method:  doSelectJoinJobeetCategory
      filter:
        display: [category_id, company, position, description, is_activated,
is_public, email, expires_at]
      form:
        class:     BackendJobeetJobForm
        display:
          Content: [category_id, type, company, logo, url, position, location,
description, how_to_apply, is_public, email]
          Admin:   [_generated_token, is_activated, expires_at]
      edit:
        title: Editing Job "%%company%% is looking for a %%position%%"
      new:
        title: Job Creation
# apps/backend/modules/category/config/generator.yml
generator:
  class: sfPropelGenerator
  param:
    model_class:           JobeetCategory
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_category
    with_propel_route:     1

    config:
      actions: ~
      fields:  ~
      list:
        title:   Category Management
        display: [=name, slug]
        batch_actions: {}
        object_actions: {}
      filter:
        class: false
      form:
        actions:
          _delete: ~
          _list:   ~
          _save:   ~
      edit:
        title: Editing Category "%%name%%"
      new:
        title: New Category
```

Con sólo estos dos archivos de configuración y en pocos minutos, hemos podido crear una interfaz de administración completa para Jobeet.

Sugerencia

Como ya sabrás, siempre que puedes configurar algo en un archivo de configuración YAML, también puedes hacerlo mediante código PHP. Para el generador de la parte de administración puedes editar el archivo `apps/backend/modules/job/lib/jobGeneratorConfiguration.class.php`. Esta clase permite utilizar las mismas opciones que las del archivo YAML pero mediante código PHP. Para aprender los nombres de cada método, puedes echar un vistazo a la clase base generada en `cache/backend/dev/modules/autoJob/lib/BaseJobGeneratorConfiguration.class.php`.

12.14. Nos vemos mañana

En sólo una hora hemos construido una completa interfaz de administración para el proyecto Jobeet. Además, hemos escrito menos de 50 líneas de código PHP, lo que no está nada mal teniendo en cuenta la cantidad de funcionalidades que contiene la interfaz.

Mañana aprenderemos a restringir la seguridad de la aplicación de administración mediante un nombre de usuario y una contraseña. Por ello también hablaremos sobre las clases de Symfony relacionadas con los usuarios.

Capítulo 13. El usuario

Ayer fue un día muy intenso y lleno de información. El generador de la parte de administración de Symfony nos permitió crear interfaces de administración completas en muy pocos minutos y con sólo unas pocas líneas de código PHP.

Hoy vamos a ver cómo gestiona Symfony la información que debe ser persistente entre las diferentes peticiones HTTP. Como ya sabes, HTTP es un protocolo sin estado, lo que significa que cada petición HTTP se considera independiente de cualquier otra petición. Por otra parte, los sitios web modernos requieren de un mecanismo para almacenar información persistente entre peticiones de forma que se pueda mejorar la experiencia de usuario.

Las sesiones de usuario se pueden identificar de forma única gracias a las cookies. En Symfony no es necesario que los programadores manipulen directamente las sesiones, ya que se puede utilizar el objeto `sfUser` que representa al usuario final de la aplicación.

13.1. Mensajes flash

En los tutoriales de los días anteriores ya hemos visto el uso del objeto `sfUser` en las acciones para establecer mensajes flash. Un mensaje flash es un mensaje temporal que se almacena en la sesión del usuario y que se borra automáticamente después de la siguiente petición.

Estos mensajes son muy útiles para mostrar información al usuario después de una redirección. El propio generador de la parte de administración utiliza mucho los mensajes flash para mostrar al usuario información sobre el resultado de las acciones, como por ejemplo cuando se crea, borra o guarda una oferta de trabajo.

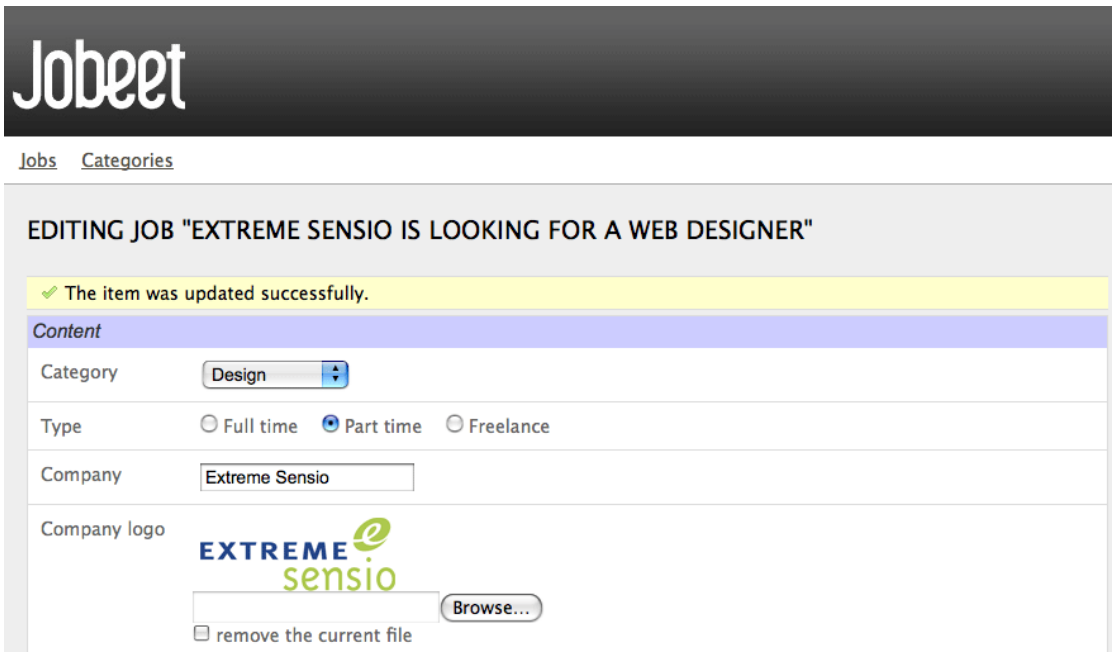

Figura 13.1. Ejemplo de mensajes flash

Los mensajes flash se crean con el método `setFlash()` del objeto `sfUser`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job validity has been
extend until %s.', $job->getExpiresAt('m/d/Y')));

  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

El primer argumento de `setFlash()` es el identificador del mensaje y el segundo argumento es el contenido del mensaje flash. Puedes definir cualquier tipo de mensaje flash, pero los tipos `notice` y `error` son los más comunes (y son los que utiliza continuamente el generador de la parte de administración).

La acción sólo crea los mensajes flash, por lo que si se quieren mostrar en la plantilla se deben incluir explícitamente. En la aplicación Jobeet, los mensajes flash se muestran en `layout.php`:

```
// apps/frontend/templates/layout.php
<?php if ($sf_user->hasFlash('notice')): ?>
  <div class="flash_notice"><?php echo $sf_user->getFlash('notice') ?></div>
<?php endif; ?>

<?php if ($sf_user->hasFlash('error')): ?>
  <div class="flash_error"><?php echo $sf_user->getFlash('error') ?></div>
<?php endif; ?>
```

La plantilla puede acceder a la información del usuario directamente a través de una variable especial llamada `sf_user`.

Nota

Algunos objetos propios de Symfony siempre están disponibles en las plantillas, sin necesidad de pasarlos de forma explícita desde la acción: `sf_request`, `sf_user` y `sf_response`.

13.2. Atributos del usuario

En los escenarios que describimos en el turorial del segundo día no incluimos ningún requisito para almacenar información en la sesión de usuario. Por tanto, a continuación vamos a definir un nuevo requerimiento: *"para facilitar la navegación por las ofertas de trabajo, en el menú se muestran los enlaces a las tres últimas ofertas de trabajo vistas por el usuario"*.

Cuando el usuario visita la página de una oferta de trabajo, debemos incluir en el historial del usuario el objeto que representa a esa oferta y debemos guardar el historial en la sesión del usuario:

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    // fetch jobs already stored in the job history
    $jobs = $this->getUser()->getAttribute('job_history', array());

    // add the current job at the beginning of the array
    array_unshift($jobs, $this->job->getId());

    // store the new job history back into the session
    $this->getUser()->setAttribute('job_history', $jobs);
  }

  // ...
}
```

Nota

En el código anterior podríamos haber guardado directamente los objetos `JobeetJob` en la sesión. No te aconsejamos que lo hagas porque las variables de sesión se serializan entre una petición y otra. Si guardáramos los objetos, al cargar la sesión se deserializarían los objetos `JobeetJob` y se podrían producir problemas si los objetos se han modificado o borrado desde que se guardaron en la sesión.

13.2.1. Los métodos getAttribute() y setAttribute()

El método `sfUser::getAttribute()` devuelve los valores de la sesión asociados al identificador que se indica. De la misma forma, el método `setAttribute()` guarda cualquier variable de PHP en la sesión del usuario y la asocia con el identificador proporcionado.

El método `getAttribute()` también permite indicar un segundo argumento opcional que es el valor que devuelve el método cuando el identificador proporcionado no está definido en la sesión del usuario.

Nota

El valor por defecto que se puede indicar en el método `getAttribute()` es simplemente un atajo de:

```
if (!$value = $this->getAttribute('job_history'))
{
  $value = array();
}
```

13.2.2. La clase myUser

Para mantener la separación del código en capas, vamos a mover el código a la clase `myUser`. La clase `myUser` redefine la clase sfUser (http://www.symfony-project.org/api/1_2/sfUser) que incluye por defecto de Symfony y permite añadir características propias de la aplicación:

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    $this->getUser()->addJobToHistory($this->job);
  }

  // ...
}
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function addJobToHistory(JobeetJob $job)
  {
    $ids = $this->getAttribute('job_history', array());

    if (!in_array($job->getId(), $ids))
    {
      array_unshift($ids, $job->getId());

      $this->setAttribute('job_history', array_slice($ids, 0, 3));
    }
  }
}
```

El código anterior también se ha modificado para tener en cuenta todos los requerimientos definidos:

- `!in_array($job->getId(), $ids)`: una misma oferta de trabajo no se puede guardar dos veces en el historial.
- `array_slice($ids, 0, 3)`: sólo se muestran las tres últimas ofertas de trabajo vistas por el usuario.

En el layout, añade el siguiente código antes de la instrucción que muestra el contenido de la variable `$sf_content`:

```
// apps/frontend/templates/layout.php
<div id="job_history">
  Recent viewed jobs:
  <ul>
    <?php foreach ($sf_user->getJobHistory() as $job): ?>
      <li>
        <?php echo link_to($job->getPosition().' - '.$job->getCompany(),
```

```
'job_show_user', $job) ?>
      </li>
    <?php endforeach; ?>
  </ul>
</div>

<div class="content">
  <?php echo $sf_content ?>
</div>
```

El layout anterior utiliza un nuevo método llamado `getJobHistory()` para obtener el historial de ofertas de trabajo visitadas:

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function getJobHistory()
  {
    $ids = $this->getAttribute('job_history', array());

    return JobeetJobPeer::retrieveByPKs($ids);
  }

  // ...
}
```

El método `getJobHistory()` utiliza el método `retrieveByPKs()` de Propel para obtener varios objetos de tipo `JobeetJob` mediante una única llamada.

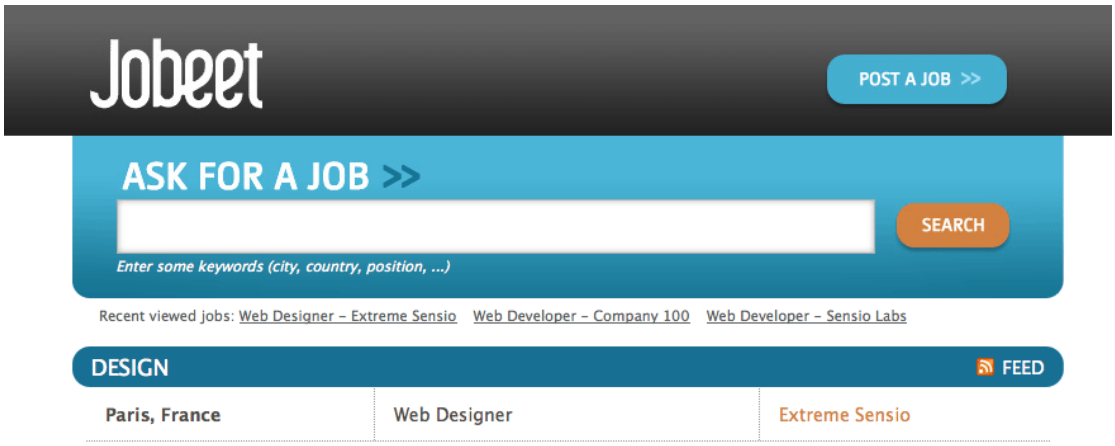

Figura 13.2. Historial de ofertas de trabajo visitadas

13.2.3. La clase sfParameterHolder

Para completar la nueva funcionalidad del historial de ofertas de trabajo, añade el siguiente método para borrar el historial:

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function resetJobHistory()
  {
```

```
    $this->getAttributeHolder()->remove('job_history');
  }

  // ...
}
```

Los atributos del usuario se gestionan a través de un objeto de la clase `sfParameterHolder`. Los métodos `getAttribute()` y `setAttribute()` de `sfUser` son en realidad atajos de los métodos `getParameterHolder()->get()` y `getParameterHolder()->set()`. Como el método `remove()` no dispone de un atajo en la clase `sfUser`, tenemos que utilizar directamente el objeto que representa al contenedor de parámetros.

Nota

La clase `sfRequest` también guarda sus parámetros en un objeto de la clase sfParameterHolder (http://www.symfony-project.org/api/1_2/sfParameterHolder) .

13.3. La seguridad de la aplicación

13.3.1. Autenticación

La seguridad de las aplicaciones Symfony se controla mediante un archivo en formato YAML llamado `security.yml`. Si quieres ver la configuración por defecto de la seguridad de la aplicación `backend`, puedes acceder al archivo `config/security.yml` de la aplicación:

```
# apps/backend/config/security.yml
default:
  is_secure: off
```

Si cambias el valor de la opción `is_secure` a `on`, la aplicación `backend` requerirá a partir de ese momento que los usuarios estén autenticados.

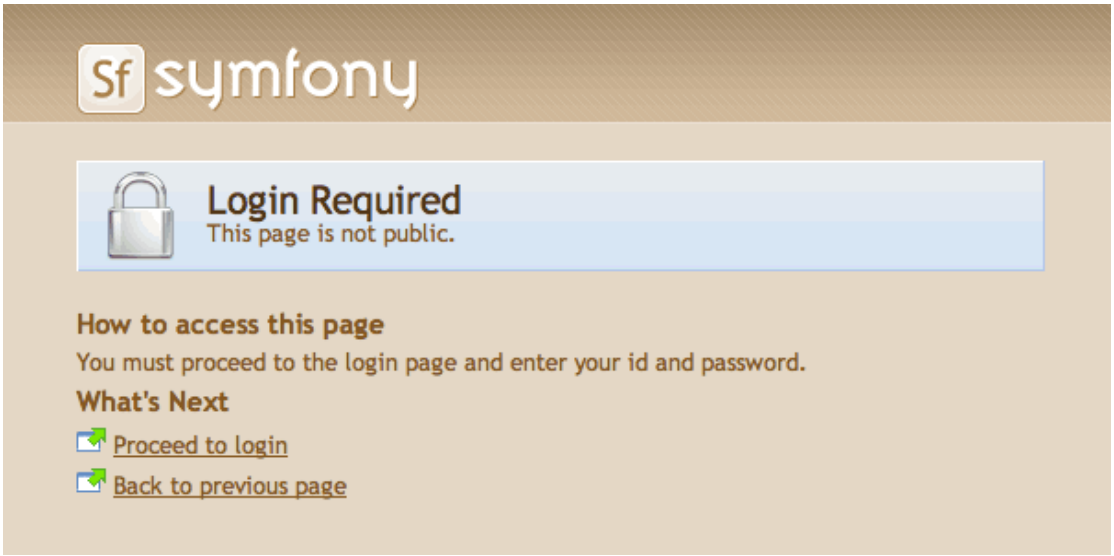

Figura 13.3. Pantalla que muestra que el usuario debe estar autenticado

Sugerencia

En los archivos YAML, los valores *booleanos* se pueden indicar con las cadenas de texto `true` y `false` o con los valores `on` y `off`.

Si echas un vistazo a los mensajes de log de la barra de depuración web, verás que cada vez que intentas acceder a una página de la aplicación `backend` se ejecuta el método `executeLogin()` de la clase `defaultActions`.

Figura 13.4. Mensajes de la barra de depuración web relacionados con el login

Cuando un usuario que no ha sido autenticado intenta acceder a una acción restringida, Symfony reenvía la petición a la acción de `login` configurada en el archivo `settings.yml`:

```
all:
  .actions:
    login_module: default
    login_action: login
```

Nota

No es posible restrigir la seguridad de la acción `login` para evitar recursiones infinitas.

Sugerencia

Como vimos en el tutorial del día 4, un mismo archivo de configuración se puede definir en diferentes directorios. Este también es el caso del archivo `security.yml`. Si sólo quieres restringir o permitir el acceso a una acción o a un módulo, crea un archivo llamado `security.yml` en el directorio `config/` de ese módulo:

```
index:
  is_secure: off
all:
  is_secure: on
```

La clase `myUser` hereda por defecto de sfBasicSecurityUser (http://www.symfony-project.org/api/1_2/sfBasicSecurityUser) y no de `sfUser`. La clase `sfBasicSecurityUser` incluye métodos adicionales para gestionar la autenticación y autorización de usuarios.

Si quieres controlar la autenticación de los usuarios, puedes utilizar los métodos `isAuthenticated()` y `setAuthenticated()`:

```
if (!$this->getUser()->isAuthenticated())
{
  $this->getUser()->setAuthenticated(true);
}
```

13.3.2. Autorización

Además de la autenticación de los usuarios, se puede restringir todavía más el acceso a algunas acciones mediante la definición de **credenciales**. Para acceder a una página determinada, el usuario debe contar con ciertas credenciales:

```
default:
  is_secure:   off
  credentials: admin
```

El sistema de credenciales de Symfony es bastante sencillo pero muy poderoso. Cada credencial puede representar cualquier cosa que requiera el modelo de seguridad de tu aplicación (como por ejemplo grupos o permisos).

Credenciales avanzadas

La opción `credentials` del archivo de configuración `security.yml` permite el uso de operaciones *booleanas* para describir los requerimientos de un sistema avanzado de credenciales.

Si un usuario debe disponer de dos credenciales, se indican entre corchetes. En el siguiente ejemplo, el usuario debe disponer tanto de la credencial `A` como de la credencial `B`:

```
index:
  credentials: [A, B]
```

Si un usuario debe disponer de al menos una de las dos credenciales, se indican con dos pares de corchetes. En el siguiente ejemplo, el usuario debe disponer o de la credencial `A` o de la credencial `B`:

```
index:
  credentials: [[A, B]]
```

También puedes combinar varios corchetes entre sí para describir cualquier tipo de expresión *booleana* compleja que utilice cualquier número de credenciales.

La clase `sfBasicSecurityUser` incluye varios métodos para gestionar las credenciales de los usuarios:

```
// Add one or more credentials
$user->addCredential('foo');
$user->addCredentials('foo', 'bar');

// Check if the user has a credential
echo $user->hasCredential('foo');                      =>   true

// Check if the user has both credentials
echo $user->hasCredential(array('foo', 'bar'));        =>   true

// Check if the user has one of the credentials
echo $user->hasCredential(array('foo', 'bar'), false); =>   true

// Remove a credential
$user->removeCredential('foo');
echo $user->hasCredential('foo');                      =>   false
```

```
// Remove all credentials (useful in the logout process)
$user->clearCredentials();
echo $user->hasCredential('bar');                    =>   false
```

En la parte de administración de Jobeet no vamos a utilizar credenciales porque sólo tenemos un perfil de usuario: el administrador.

13.4. Plugins

Como no nos gusta reinventar la rueda cada vez que tenemos que añadir una funcionalidad en la aplicación, no vamos a desarrollar un completo sistema de login, sino que vamos a instalar un **plugin de Symfony**.

Uno de los puntos fuertes del framework Symfony es su *ecosistema de plugins* (http://www.symfony-project.org/plugins/) . Como veremos en los próximos días, es muy sencillo crear un plugin. Además, los plugins son muy poderosos, ya que pueden contener desde configuración hasta módulos enteros y archivos.

Hoy vamos a instalar el plugin sfGuardPlugin (http://www.symfony-project.org/plugins/sfGuardPlugin) para restringir el acceso a la aplicación `backend`:

```
$ php symfony plugin:install sfGuardPlugin
```

La tarea `plugin:install` instala el plugin cuyo nombre se pasa como parámetro. Todos los plugins se guardan en el directorio `plugins/` y cada plugin dispone de su propio directorio llamado igual que el plugin.

> **Nota**
>
> Debes tener PEAR correctamente instalado y configurado en tu sistema para que funcione la tarea `plugin:install`.

Cuando se instalar un plugin con la tarea `plugin:install`, Symfony siempre instala su última versión estable. Para instalar una versión específica del plugin, puedes utilizar la opción `--release`. La página de cada plugin, como por ejemplo la página del plugin sfGuardPlugin (http://www.symfony-project.org/plugins/sfGuardPlugin?tab=plugin_all_releases) , muestra un listado de todas las versiones disponibles para cada versión de Symfony.

Como cada plugin se instala en su propio directorio, también puedes descargar sfGuardPlugin como archivo comprimido (http://www.symfony-project.org/plugins/sfGuardPlugin?tab=plugin_installation) y descomprimirlo en el directorio correspondiente. También puedes establecer un enlace con `svn:externals` al repositorio Subversion de sfGuardPlugin (http://svn.symfony-project.com/plugins/sfGuardPlugin) .

13.5. La seguridad de la aplicación backend

Cada plugin dispone de su propio archivo README (http://www.symfony-project.org/plugins/sfGuardPlugin?tab=plugin_readme) donde se explica cómo se configura. A continuación se muestra cómo configurar el plugin `sfGuardPlugin`. Como se trata de un plugin que incluye varias clases de su propio modelo de datos para gestionar usuarios, grupos y permisos, lo primero que debemos hacer es volver a generar todas las clases del modelo:

```
$ php symfony propel:build-all-load --no-confirmation
```

Sugerencia

Recuerda que la tarea `propel:build-all-load` borra todas las tablas de la base de datos antes de volver a crearlas. Si no quieres borrar las tablas, puedes generar los modelos, formularios y filtros y después, puedes crear las nuevas tablas ejecutando las sentencias SQL generadas en el directorio `data/sql`.

Como siempre que se crean nuevas clases, no te olvides de borrar la cache de Symfony:

```
$ php symfony cc
```

Como el plugin `sfGuardPlugin` añade varios métodos a la clase del usuario, tienes que modificar la clase de la que hereda `myUser` a `sfGuardSecurityUser`:

```
// apps/backend/lib/myUser.class.php
class myUser extends sfGuardSecurityUser
{
}
```

El plugin `sfGuardPlugin` incluye una acción llamada `signin` en el módulo `sfGuardAuth` para autenticar a los usuarios:

Modifica el archivo `settings.yml` para cambiar la acción utilizada por defecto en la página de login:

```
# apps/backend/config/settings.yml
all:
  .settings:
    enabled_modules: [default, sfGuardAuth]

    # ...

  .actions:
    login_module:    sfGuardAuth
    login_action:    signin

    # ...
```

Como los plugins están disponibles en todas las aplicaciones del proyecto, tienes que activar de forma explícita los módulos que quieres utilizar mediante la opción `enabled_modules`.

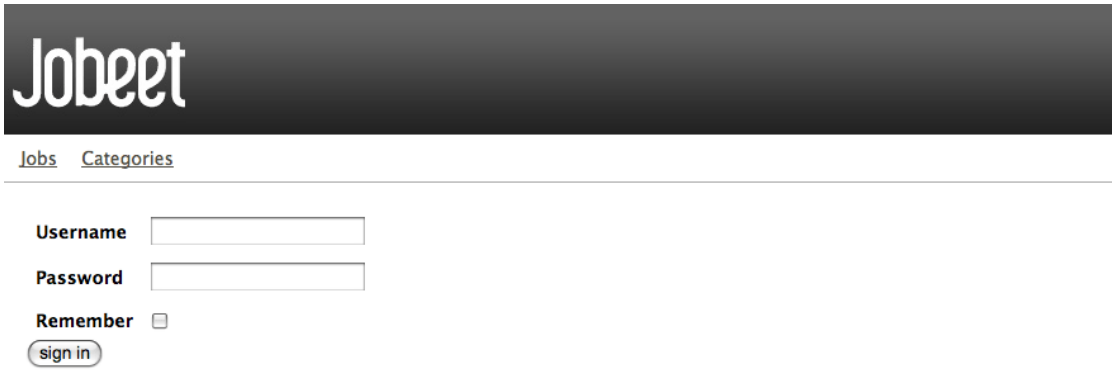

Figura 13.5. Pantalla de login del plugin sfGuardPlugin

Por último, crea el usuario de tipo administrador:

```
$ php symfony guard:create-user fabien ConTraSenA
$ php symfony guard:promote fabien
```

Sugerencia

El plugin `sfGuardPlugin` incluye tareas para gestionar usuarios, grupos y permisos directamente desde la línea de comandos. Si quieres ver todas las tareas disponibles para el *namespace* `guard`, puedes utilizar la tarea `list`:

```
$ php symfony list guard
```

El siguiente paso consiste en no mostrar la barra del menú si el usuario no está autenticado:

```
// apps/backend/templates/layout.php
<?php if ($sf_user->isAuthenticated()): ?>
  <div id="menu">
    <ul>
      <li><?php echo link_to('Jobs', '@jobeet_job') ?></li>
      <li><?php echo link_to('Categories', '@jobeet_category') ?></li>
    </ul>
  </div>
<?php endif; ?>
```

Por otra parte, cuando el usuario está autenticado, tenemos que mostrar un enlace para la acción de desconectar que incluye el plugin `sfGuardPlugin`:

```
// apps/backend/templates/layout.php
<li><?php echo link_to('Logout', '@sf_guard_signout') ?></li>
```

Sugerencia

Si quieres ver todas las rutas que define `sfGuardPlugin`, utiliza la tarea `app:routes`.

Para completar la parte de administración de Jobeet, vamos a añadir un módulo para gestionar los usuarios de tipo administrador. Afortunadamente, el plugin `sfGuardPlugin` ya incluye un módulo de este tipo. Para utilizarlo, debes activar el módulo llamado `sfGuardAuth` en el archivo de configuración `settings.yml`:

```
# apps/backend/config/settings.yml
all:
  .settings:
    enabled_modules: [default, sfGuardAuth, sfGuardUser]
```

Y por último, añade un enlace en el menú:

```
// apps/backend/templates/layout.php
<li><?php echo link_to('Users', '@sf_guard_user') ?></li>
```

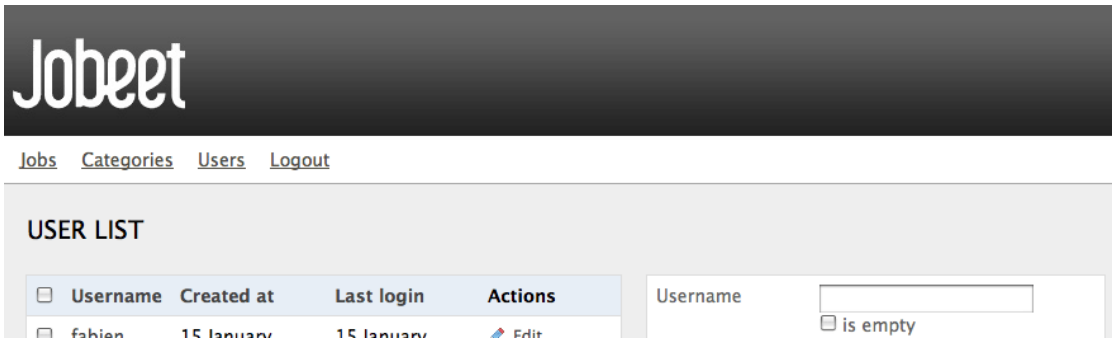

Figura 13.6. Menú de la parte de administración

Y eso es todo lo que tenemos que hacer para disponer de una completa gestión de usuarios, grupos y permisos.

13.6. Probando a los usuarios

El tutorial de hoy todavía no se ha acabado porque todavía no hemos hablado de cómo probar la parte de los usuarios. Como el navegador que incluye Symfony también simula el comportamiento de las cookies, es muy sencillo crear pruebas para la parte de los usuarios utilizando el *tester* sfTesterUser (http://symfony-project.org/api/1_2/sfTesterUser) .

A continuación vamos a actualizar las pruebas funcionales para las opciones del menú que hemos añadido durante el día de hoy. Añade el siguiente código al final de las pruebas funcionales del módulo `job`:

```
// test/functional/frontend/jobActionsTest.php
$browser->
  info('4 - User job history')->

  loadData()->
  restart()->

  info('  4.1 - When the user access a job, it is added to its history')->
  get('/')->
  click('Web Developer', array(), array('position' => 1))->
  get('/')->
  with('user')->begin()->
    isAttribute('job_history',
array($browser->getMostRecentProgrammingJob()->getId()))->
  end()->

  info('  4.2 - A job is not added twice in the history')->
```

```
  click('Web Developer', array(), array('position' => 1))->
  get('/')->
  with('user')->begin()->
    isAttribute('job_history',
array($browser->getMostRecentProgrammingJob()->getId()))->
  end()
;
```

Para que las pruebas sean más sencillas, en primer lugar volvemos a cargar los datos de prueba y reiniciamos el navegador para comenzar con una sesión de usuario limpia.

El método `isAttribute()` comprueba el atributo de usuario que se indica.

Nota

El *tester* `sfTesterUser` también incluye los métodos `isAuthenticated()` y `hasCredential()` para poder probar respectivamente la autenticación y la autorización del usuario.

13.7. Nos vemos mañana

Las clases de usuario de Symfony son una buena forma de abstraerse de la gestión de sesiones de PHP. Si a ello unimos el sistema de plugins de Symfony y sobre todo, el plugin `sfGuardPlugin`, podemos restringir la seguridad de la parte de administración de Jobeet en pocos minutos. Además, gracias a los módulos que incluye el plugin, hemos podido añadir un gestor de usuarios de tipo administrador.

Capítulo 14. El día de descanso

Después de la explicación ayer de las clases relacionadas con los usuarios, ya hemos completado el recorrido por todas las características fundamentales de Symfony. Aunque todavía te quedan muchas cosas por aprender, ya deberías ser capaz de crear por tu cuenta proyectos Symfony sencillos.

Para celebrar este hito, hoy vamos a hacer un descanso. En realidad, sólo vamos a descansar nosotros, porque hoy no vamos a publicar ningún tutorial. No obstante, vamos a darte unas pistas sobre lo que podrías hacer hoy para mejorar tus habilidades con Symfony.

14.1. Aprendiendo con la práctica

El framework Symfony, como cualquier otra aplicación, tiene su propia curva de aprendizaje. El primer paso en el proceso de aprendizaje consiste en utilizar ejemplos prácticos, tutoriales o libros como el que estás leyendo. El segundo paso consiste en **practicar**, que es algo que jamás se podrá reemplazar.

Esto es precisamente lo que puedes empezar a hacer hoy mismo. Piensa en cualquier proyecto web sencillo que pueda aportar valor: una lista de tareas, un blog sencillo, un conversor de divisas, etc. Selecciona un proyecto y empieza a desarrollarlo con todo lo que ya sabes.

Haz uso de los mensajes de ayuda de las tareas para aprender cada una de sus opciones, investiga el código generado automáticamente por Symfony, utiliza un editor de textos que tenga autocompletado de PHP como Eclipse (http://www.eclipse.org/) , lee la documentación de la API (http://www.symfony-project.org/api/1_2/) para descubrir nuevos métodos, pregunta todas las dudas que tengas en el grupo de usuarios de Google (http://groups.google.com/group/symfony-es/) , conéctate al chat en el canal #symfony del IRC de freenode (irc://irc.freenode.net/symfony) .

Y sobre todo, disfruta de la gran cantidad de material gratuito relacionado con Symfony que tienes a tu disposición.

Capítulo 15. Canales Atom

Si seguiste nuestra recomendación, ayer empezaste a desarrollar tu propio proyecto de Symfony. No abandones ese proyecto y continúa añadiendo características a tu aplicación a medida que aprendas más conceptos avanzados de Symfony. Además, puedes colgar tu proyecto en cualquier servidor público de Internet para compartirlo con la comunidad.

Sin embargo, nosotros hoy nos vamos a dedicar a algo completamente diferente.

Si estás buscando trabajo, seguramente te interesa enterarte lo antes posible de las ofertas de trabajo que se publican. Como evidentemente no es lógico estar entrando en el sitio web cada poco tiempo para comprobar si se han publicado nuevas ofertas, hoy vamos a añadir varios canales Atom a la aplicación para mantener actualizados a los usuarios de Jobeet.

15.1. Formatos

El framework Symfony incluye soporte de muchos formatos y tipos MIME. Esto significa que la parte del modelo y del controlador pueden utilizar diferentes plantillas en función del formato en el que se realiza la petición. El formato por defecto es HTML, pero Symfony soporta muchos otros formatos como `txt`, `js`, `css`, `json`, `xml`, `rdf` y `atom`.

El formato se puede establecer con el método `setRequestFormat()` del objeto de la petición:

```
$request->setRequestFormat('xml');
```

No obstante, el formato se incluye casi siempre en la propia URL. En este caso, Symfony establece automáticamente el formato si en la ruta correspondiente se utiliza una variable especial llamada `sf_format`. La URL del listado de ofertas de trabajo es:

```
http://jobeet.localhost/frontend_dev.php/job
```

La URL anterior es equivalente a:

```
http://jobeet.localhost/frontend_dev.php/job.html
```

Las dos URL anteriores son equivalentes porque las rutas generadas por la clase `sfPropelRouteCollection` incluyen la variable `sf_format` como extensión del archivo y porque `html` es el valor por defecto. Si quieres comprobarlo tu mismo, puedes utilizar la tarea `app:routes` que muestra todas las rutas de la aplicación:

```
~/work/jobeet $ ./symfony app:routes frontend
>> app       Current routes for application "frontend"
Name          Method Pattern
category      ANY    /category/:slug
job           GET    /job.:sf_format
job_new       GET    /job/new.:sf_format
job_create    POST   /job.:sf_format
job_edit      GET    /job/:token/edit.:sf_format
job_update    PUT    /job/:token.:sf_format
job_delete    DELETE /job/:token.:sf_format
job_show      GET    /job/:token.:sf_format
job_publish   PUT    /job/:token/publish.:sf_format
job_extend    PUT    /job/:token/extend.:sf_format
job_show_user GET    /job/:company_slug/:location_slug/:id/:position_slug
homepage      ANY    /
~/work/jobeet $
```

Figura 15.1. Resultado de ejecutar la tarea app:routes

15.2. Canales Atom

15.2.1. Canal de las últimas ofertas de trabajo

Soportar diferentes formatos es tan sencillo como crear diferentes plantillas. Si quieres crear un canal en formato Atom (http://es.wikipedia.org/wiki/Atom_(formato_de_redifusión)) que incluya las últimas ofertas de trabajo publicadas, crea un plantilla llamada `indexSuccess.atom.php`:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="" rel="self"/>
  <link href=""/>
  <updated></updated>
  <author><name>Jobeet</name></author>
  <id>Unique Id</id>

  <entry>
    <title>Job title</title>
    <link href="" />
    <id>Unique id</id>
    <updated></updated>
    <summary>Job description</summary>
    <author><name>Company</name></author>
  </entry>
</feed>
```

El nombre de las plantillas

Como `html` es el formato más utilizado en las aplicaciones web, puedes omitirlo en el nombre de la plantilla. Tanto `indexSuccess.php` como `indexSuccess.html.php` son nombres equivalentes y Symfony siempre utiliza el primero que encuentra.

¿Por qué añadimos el sufijo `Success` al nombre de todas las plantillas? Las acciones de Symfony pueden devolver un valor que indica la plantilla que se debe utilizar. Si la acción no devuelve nada, se considera que es equivalente al siguiente código:

```
return sfView::SUCCESS; // == 'Success'
```

Si quieres modificar el sufijo del nombre de la plantilla, simplemente devuelve cualquier otro valor:

```
return sfView::ERROR; // == 'Error'

return 'Foo';
```

También puedes modificar el nombre de la plantilla utilizando el método `setTemplate()`:

```
$this->setTemplate('foo');
```

Symfony modifica el valor del `Content-Type` de la respuesta en función del formato utilizado y además, deshabilita el layout para cualquier formato que no sea HTML. En el caso del canal Atom, Symfony cambia el valor del `Content-Type` a `application/atom+xml; charset=utf-8`

A continuación, actualiza en el pie de página del layout de Jobeet el enlace al nuevo canal:

```
<!-- apps/frontend/templates/layout.php -->
<li class="feed">
  <a href="<?php echo url_for('@job?sf_format=atom') ?>">Full feed</a>
</li>
```

La URI interna del canal Atom es la misma que la del listado `job` pero con `sf_format` añadido en forma de variable.

No te olvides de incluir también la etiqueta `<link>` de HTML en la cabecera del layout para que los navegadores puedan descubrir automáticamente la presencia de los canales:

```
<!-- apps/frontend/templates/layout.php -->
<link rel="alternate" type="application/atom+xml" title="Latest Jobs"
href="<?php echo url_for('@job?sf_format=atom', true) ?>" />
```

En este caso, el atributo `href` incluye la URL absoluta del canal Atom, porque se ha utilizado el segundo argumento del helper `url_for()`.

Para crear el canal Atom, en primer lugar reemplaza la cabecera de la plantilla de Atom por el siguiente código:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<title>Jobeet</title>
<subtitle>Latest Jobs</subtitle>
<link href="<?php echo url_for('@job?sf_format=atom', true) ?>" rel="self"/>
<link href="<?php echo url_for('@homepage', true) ?>"/>
<updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
JobeetJobPeer::getLatestPost()->getCreatedAt('U')) ?></updated>
```

```
<author>
  <name>Jobeet</name>
</author>
<id><?php echo sha1(url_for('@job?sf_format=atom', true)) ?></id>
```

Si te fijas en el código anterior, verás que hemos utilizado la letra `U` como argumento del método `getCreatedAt()` para obtener la fecha en forma de timestamp. Si quieres obtener la fecha de la última oferta de trabajo, crea un método llamado `getLatestPost()`:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public function getLatestPost()
  {
    $criteria = new Criteria();
    self::addActiveJobsCriteria($criteria);

    return JobeetJobPeer::doSelectOne($criteria);
  }

  // ...
}
```

Una vez terminada la cabecera, el cuerpo del canal Atom se puede generar con el siguiente código:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?php use_helper('Text') ?>
<?php foreach ($categories as $category): ?>
  <?php foreach
($category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')) as $job):
?>
    <entry>
      <title>
        <?php echo $job->getPosition() ?> (<?php echo $job->getLocation() ?>)
      </title>
      <link href="<?php echo url_for('job_show_user', $job, true) ?>" />
      <id><?php echo sha1($job->getId()) ?></id>
      <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
$job->getCreatedAt('U')) ?></updated>
      <summary type="xhtml">
       <div xmlns="http://www.w3.org/1999/xhtml">
         <?php if ($job->getLogo()): ?>
           <div>
             <a href="<?php echo $job->getUrl() ?>">
               <img src="http://<?php echo $sf_request->getHost().'/uploads/
jobs/'.$job->getLogo() ?>"
                    alt="<?php echo $job->getCompany() ?> logo" />
             </a>
           </div>
         <?php endif; ?>

         <div>
           <?php echo simple_format_text($job->getDescription()) ?>
```

```
        </div>

        <h4>How to apply?</h4>

        <p><?php echo $job->getHowToApply() ?></p>
      </div>
    </summary>
    <author>
      <name><?php echo $job->getCompany() ?></name>
    </author>
  </entry>
<?php endforeach; ?>
<?php endforeach; ?>
```

El método `getHost()` del objeto de la petición (`$sf_request`) devuelve el *host* o servidor actual, lo que resulta muy útil para crear el enlace absoluto de la imagen del logotipo de la empresa.

Figura 15.2. Canal Atom tal y como se muestra en el navegador

Sugerencia

Cuando desarrollas canales RSS o Atom, es mucho más fácil depurarlos si utilizas herramientas de la línea de comandos como curl (http://curl.haxx.se/) o wget (http://www.gnu.org/software/wget/) , ya que te permiten ver directamente el contenido real del canal.

15.2.2. Canal de las últimas ofertas de trabajo de una categoría

Uno de los objetivos de Jobeet es ayudar a la gente a encontrar puestos de trabajo muy específicos. Por tanto, es imprescindible que incluyamos canales en cada categoría.

En primer lugar, actualiza la ruta `category` para añadir el soporte de varios formatos:

```
# apps/frontend/config/routing.yml
category:
  url:     /category/:slug.:sf_format
  class:   sfPropelRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object }
  requirements:
    sf_format: (?:html|atom)
```

Ahora la ruta `category` ya es capaz de reconocer los formatos `html` y `atom`. El siguiente paso consiste en actualizar en la plantilla los enlaces a los canales de cada categoría:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<div class="feed">
  <a href="<?php echo url_for('category', array('sf_subject' => $category,
'sf_format' => 'atom')) ?>">Feed</a>
</div>
<!-- apps/frontend/modules/category/templates/showSuccess.php -->
<div class="feed">
  <a href="<?php echo url_for('category', array('sf_subject' => $category,
'sf_format' => 'atom')) ?>">Feed</a>
</div>
```

Por último, crea una plantilla llamada `showSuccess.atom.php`. Como esta plantilla también incluye un listado de ofertas de trabajo, vamos a refactorizar el código que genera los elementos del canal Atom mediante un elemento parcial llamado `_list.atom.php`. Al igual que para el formato `html`, los elementos parciales son dependientes del formato:

```
<!-- apps/frontend/job/templates/_list.atom.php -->
<?php use_helper('Text') ?>

<?php foreach ($jobs as $job): ?>
  <entry>
    <title><?php echo $job->getPosition() ?> (<?php echo $job->getLocation()
?>)</title>
    <link href="<?php echo url_for('job_show_user', $job, true) ?>" />
    <id><?php echo sha1($job->getId()) ?></id>
    <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
$job->getCreatedAt('U')) ?></updated>
    <summary type="xhtml">
     <div xmlns="http://www.w3.org/1999/xhtml">
       <?php if ($job->getLogo()): ?>
         <div>
           <a href="<?php echo $job->getUrl() ?>">
             <img src="http://<?php echo $sf_request->getHost().'/uploads/jobs/
'.$job->getLogo() ?>" alt="<?php echo $job->getCompany() ?> logo" />
           </a>
         </div>
       <?php endif; ?>

       <div>
         <?php echo simple_format_text($job->getDescription()) ?>
       </div>
```

```
      <h4>How to apply?</h4>

      <p><?php echo $job->getHowToApply() ?></p>
    </div>
   </summary>
   <author>
     <name><?php echo $job->getCompany() ?></name>
   </author>
  </entry>
<?php endforeach; ?>
```

Utilizando este elemento parcial `_list.atom.php` se puede simplificar mucho la plantilla del canal que hemos creado en la sección anterior y que muestra las últimas ofertas de trabajo de todo el sitio:

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="<?php echo url_for('@job?sf_format=atom', true) ?>" rel="self"/>
  <link href="<?php echo url_for('@homepage', true) ?>"/>
  <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
JobeetJobPeer::getLatestPost()->getCreatedAt('U')) ?></updated>
  <author>
    <name>Jobeet</name>
  </author>
  <id><?php echo sha1(url_for('@job?sf_format=atom', true)) ?></id>

<?php foreach ($categories as $category): ?>
  <?php include_partial('job/list', array('jobs' =>
$category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')))) ?>
<?php endforeach; ?>
</feed>
```

Por último, crea la plantilla `showSuccess.atom.php` haciendo uso del elemento parcial `_list.atom.php`:

```
<!-- apps/frontend/modules/category/templates/showSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet (<?php echo $category ?>)</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="<?php echo url_for('category', array('sf_subject' => $category,
'sf_format' => 'atom'), true) ?>" rel="self" />
  <link href="<?php echo url_for('category', array('sf_subject' => $category),
true) ?>" />
  <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
$category->getLatestPost()->getCreatedAt('U')) ?></updated>
  <author>
    <name>Jobeet</name>
  </author>
  <id><?php echo sha1(url_for('category', array('sf_subject' => $category),
true)) ?></id>
```

```
  <?php include_partial('job/list', array('jobs' => $pager->getResults())) ?>
</feed>
```

Al igual que para el canal principal del sitio, tenemos que calcular la fecha de la última oferta de trabajo de cada categoría:

```
// lib/model/JobeetCategory.php
class JobeetCategory extends BaseJobeetCategory
{
  public function getLatestPost()
  {
    $jobs = $this->getActiveJobs(1);

    return $jobs[0];
  }

  // ...
}
```

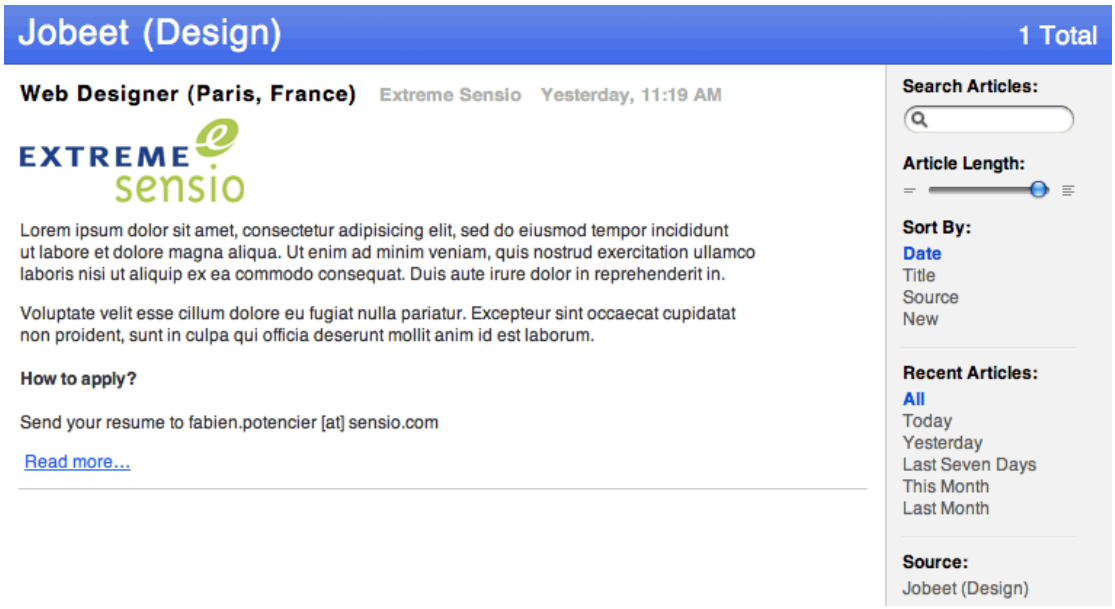

Figura 15.3. Canal Atom de cada categoría

15.3. Nos vemos mañana

Como sucede con otras muchas características de Symfony, el soporte nativo de formatos y tipos MIME permite crear canales Atom de forma sencilla y sin esfuerzo.

Hoy hemos mejorado la experiencia de usuario de los que buscan trabajo. Mañana mejoraremos la experiencia de usuario de los que publican las ofertas de trabajo mediante la creación de servicios web.

Capítulo 16. Servicios web

Ayer añadimos canales Atom a la aplicación, de forma que los usuarios que buscan trabajo con Jobeet pueden estar informados casi en tiempo real de las nuevas ofertas que se publican.

Si se considera el otro lado del proceso, cuando un usuario publica una oferta de trabajo, seguramente quiere que esa oferta sea vista por la mayor cantidad de personas. Si la oferta de trabajo se publica de forma simultánea en muchos sitios web, es más probable que puedas encontrar a la persona adecuada para el puesto. Este fenómeno se conoce como el *long tail* (http://es.wikipedia.org/wiki/Larga_Cola) . Hoy vamos a desarrollar los servicios web que van a permitir a los afiliados publicar las últimas ofertas de trabajo en sus propios sitios web.

16.1. Los afiliados

En los escenarios del tutorial del día 2 establecimos que *"un usuario afiliado obtiene la lista de ofertas de trabajo activas"*.

16.1.1. Los archivos de datos

A continuación vamos a crear un nuevo archivo de datos para la información de los afiliados:

```
# data/fixtures/030_affiliates.yml
JobeetAffiliate:
  sensio_labs:
    url:       http://www.sensio-labs.com/
    email:     fabien.potencier@example.com
    is_active: true
    token:     sensio_labs
    jobeet_category_affiliates: [programming]

  symfony:
    url:       http://www.symfony-project.org/
    email:     fabien.potencier@example.org
    is_active: false
    token:     symfony
    jobeet_category_affiliates: [design, programming]
```

Cuando se establecen relaciones muchos-a-muchos, crear los registros de la tabla intermedia es tan sencillo como definir un array cuya clave sea el nombre de la tabla intermedia seguido de una letra s. El contenido del array está formado por los nombres de los objetos que se han definido en los archivos de datos. Puedes utilizar objetos definidos en otros archivos de datos, pero con la condición de que los objetos hayan sido definidos antes de utilizarlos (el orden en el que se cargan los archivos YAML es importante).

El archivo de datos anterior ya incluye el valor del token de cada afiliado para que las pruebas sean más fáciles. En cualquier caso, cuando un usuario real solicita una cuenta, el token se debe generar automáticamente:

```
// lib/model/JobeetAffiliate.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function save(PropelPDO $con = null)
  {
    if (!$this->getToken())
    {
      $this->setToken(sha1($this->getEmail().rand(11111, 99999)));
    }

    return parent::save($con);
  }

  // ...
}
```

Después de crear el archivo de datos, ya puedes volver a cargar todos los datos de prueba:

```
$ php symfony propel:data-load
```

16.1.2. El servicio web de las ofertas de trabajo

Como ya hemos explicado varias veces, siempre que vayas a añadir alguna nueva funcionalidad a la aplicación, es mejor pensar primero en su URL:

```
# apps/frontend/config/routing.yml
api_jobs:
  url:     /api/:token/jobs.:sf_format
  class:   sfPropelRoute
  param:   { module: api, action: list }
  options: { model: JobeetJob, type: list, method: getForToken }
  requirements:
    sf_format: (?:xml|json|yaml)
```

En la ruta anterior, la variable especial `sf_format` es el último elemento que forma la URL y sus posibles valores son `xml`, `json` o `yaml`.

El método `getForToken()` se invoca cuando la acción obtiene la colección de objetos relacionados con la ruta. Como es necesario comprobar que el afiliado se encuentra activado, debemos redefinir el comportamiento por defecto de la ruta:

```
// lib/model/JobeetJobPeer.php
class JobeetJobPeer extends BaseJobeetJobPeer
{
  static public function getForToken(array $parameters)
  {
    $affiliate = JobeetAffiliatePeer::getByToken($parameters['token']);
    if (!$affiliate || !$affiliate->getIsActive())
    {
```

```
      throw new sfError404Exception(sprintf('Affiliate with token "%s" does not
exist or is not activated.', $parameters['token']));
    }

    return $affiliate->getActiveJobs();
  }

  // ...
}
```

Si el token no existe en la base de datos, se lanza una excepción de tipo `sfError404Exception`. Después, esta clase se convierte automáticamente en una respuesta de error de tipo `404`. Esta es por tanto la forma más sencilla de generar una página de error `404` desde una clase del modelo.

El método `getForToken()` utiliza, a su vez, otros dos nuevos métodos que vamos a crear a continuación.

En primer lugar tenemos que crear el método `getByToken()` para obtener los datos de un afiliado a partir del token que se indica:

```
// lib/model/JobeetAffiliatePeer.php
class JobeetAffiliatePeer extends BaseJobeetAffiliatePeer
{
  static public function getByToken($token)
  {
    $criteria = new Criteria();
    $criteria->add(self::TOKEN, $token);

    return self::doSelectOne($criteria);
  }
}
```

En segundo lugar, el método `getActiveJobs()` devuelve el listado de las actuales ofertas de trabajo activas para las categorías seleccionadas por el afiliado:

```
// lib/model/JobeetAffiliate.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function getActiveJobs()
  {
    $cas = $this->getJobeetCategoryAffiliates();
    $categories = array();
    foreach ($cas as $ca)
    {
      $categories[] = $ca->getCategoryId();
    }

    $criteria = new Criteria();
    $criteria->add(JobeetJobPeer::CATEGORY_ID, $categories, Criteria::IN);
    JobeetJobPeer::addActiveJobsCriteria($criteria);

    return JobeetJobPeer::doSelect($criteria);
  }
```

```
  // ...
}
```

El último paso consiste en crear la acción y las plantillas relacionadas con la API. Para ello, crea un módulo vacío llamado `api` utilizando la tarea `generate:module`:

```
$ php symfony generate:module frontend api
```

Nota

Como no vamos a hacer uso de la acción `index` generada por defecto, la puedes borrar de la clase de las acciones y también puedes borrar su plantilla asociada `indexSucess.php`

16.1.3. La acción

La misma acción `list` que se muestra a continuación se utiliza para todos los formatos en los que se pueden obtener los datos de la API:

```
// apps/frontend/modules/api/actions/actions.class.php
public function executeList(sfWebRequest $request)
{
  $this->jobs = array();
  foreach ($this->getRoute()->getObjects() as $job)
  {
    $this->jobs[$this->generateUrl('job_show_user', $job, true)] =
$job->asArray($request->getHost());
  }
}
```

En vez de pasar un array de objetos `JobeetJob` a las plantillas, les pasamos simplemente un array de cadenas de texto. Además, como tenemos tres plantillas diferentes para la misma acción, hemos creado un método llamado `JobeetJob::asArray()` que contiene la lógica que procesa los valores:

```
// lib/model/JobeetJob.php
class JobeetJob extends BaseJobeetJob
{
  public function asArray($host)
  {
    return array(
      'category'     => $this->getJobeetCategory()->getName(),
      'type'         => $this->getType(),
      'company'      => $this->getCompany(),
      'logo'         => $this->getLogo() ? 'http://'.$host.'/uploads/
jobs/'.$this->getLogo() : null,
      'url'          => $this->getUrl(),
      'position'     => $this->getPosition(),
      'location'     => $this->getLocation(),
      'description'  => $this->getDescription(),
      'how_to_apply' => $this->getHowToApply(),
      'expires_at'   => $this->getCreatedAt('c'),
    );
  }
```

```
  // ...
}
```

16.1.4. El formato XML

Si recuerdas el tutorial de ayer, añadir el soporte del formato `xml` es tan sencillo como crear una nueva plantilla:

```
<!-- apps/frontend/modules/api/templates/listSuccess.xml.php -->
<?xml version="1.0" encoding="utf-8"?>
<jobs>
<?php foreach ($jobs as $url => $job): ?>
  <job url="<?php echo $url ?>">
<?php foreach ($job as $key => $value): ?>
    <<?php echo $key ?>><?php echo $value ?></<?php echo $key ?>>
<?php endforeach; ?>
  </job>
<?php endforeach; ?>
</jobs>
```

16.1.5. El formato JSON

De la misma forma, añadir el soporte del formato JSON (http://json.org/) es muy similar:

```
<!-- apps/frontend/modules/api/templates/listSuccess.json.php -->
[
<?php $nb = count($jobs); $i = 0; foreach ($jobs as $url => $job): ++$i ?>
{
  "url": "<?php echo $url ?>",
<?php $nb1 = count($job); $j = 0; foreach ($job as $key => $value): ++$j ?>
  "<?php echo $key ?>": <?php echo json_encode($value).($nb1 == $j ? '' : ',')
?>

<?php endforeach; ?>
}<?php echo $nb == $i ? '' : ',' ?>

<?php endforeach; ?>
]
```

16.1.6. El formato YAML

Cuando el formato que utilizas es uno de los que incluye Symfony por defecto, el framework se encarga de realizar automáticamente algunas tareas como por ejemplo cambiar el `Content-Type` de la respuesta o deshabilitar el layout.

Como el formato YAML no está incluido entre los formatos que soporta Symfony para la peticiones de los usuarios, debemos modificar el `Content-Type` de la respuesta y debemos deshabilitar el layout desde la acción:

```
class apiActions extends sfActions
{
```

```
  public function executeList(sfWebRequest $request)
  {
    $this->jobs = array();
    foreach ($this->getRoute()->getObjects() as $job)
    {
      $this->jobs[$this->generateUrl('job_show_user', $job, true)] =
$job->asArray($request->getHost());
    }

    switch ($request->getRequestFormat())
    {
      case 'yaml':
        $this->setLayout(false);
        $this->getResponse()->setContentType('text/yaml');
        break;
    }
  }
}
```

En una acción, el método `setLayout()` modifica el layout utilizado por defecto y también permite deshabilitarlo si utilizas el valor `false`.

A continuación se muestra la plantilla resultante para el formato YAML:

```
<!-- apps/frontend/modules/api/templates/listSuccess.yaml.php -->
<?php foreach ($jobs as $url => $job): ?>
-
  url: <?php echo $url ?>

<?php foreach ($job as $key => $value): ?>
  <?php echo $key ?>: <?php echo sfYaml::dump($value) ?>

<?php endforeach; ?>
<?php endforeach; ?>
```

Si realizas una llamada a este servicio web con un token inválido, verás una página de error `404` en formato XML si la petición la realizas en XML y una página de error `404` en formato JSON si tu petición estaba en el formato JSON. Sin embargo, si se produce un error con una petición en formato YAML, symfony no sabe lo que debe mostrar.

Cada vez que creas un nuevo formato, debes crear una plantilla de error asociada. Esta plantilla se utiliza para las páginas del error `404` pero también para todas las demás excepciones.

Como las excepciones deben ser diferentes en el entorno de producción y en el de desarrollo, debes crear dos archivos diferentes: `config/error/exception.yaml.php` para el entorno de desarrollo y `config/error/error.yaml.php` para el de producción:

```
// config/error/exception.yaml.php
<?php echo sfYaml::dump(array(
  'error'       => array(
    'code'      => $code,
    'message'   => $message,
    'debug'     => array(
```

```
      'name'    => $name,
      'message' => $message,
      'traces'  => $traces,
    ),
)), 4) ?>
// config/error/error.yaml.php
<?php echo sfYaml::dump(array(
  'error'       => array(
    'code'      => $code,
    'message'   => $message,
))) ?>
```

Por último, antes de probar estas páginas no te olvides de crear un layout para el formato YAML:

```
// apps/frontend/templates/layout.yaml.php
<?php echo $sf_content ?>
```

```
~/work/jobeet $ curl http://jobeet.localhost/frontend_dev.php/api/sensio_lab/jobs.yaml
error:
  code: 404
  message: 'Affiliate with token "sensio_lab" does not exist or is not activated.'
  debug:
    name: sfError404Exception
    message: 'Affiliate with token "sensio_lab" does not exist or is not activated.'
    traces:
      - 'at () in SF_ROOT_DIR/lib/model/JobeetJobPeer.php line 12'
      - 'at JobeetJobPeer::getForToken(array(''token'' =&gt; ''sensio_lab'', ''sf_format''
      - 'at call_user_func(array(''JobeetJobPeer'', ''getForToken''), array(''token'' =&gt
      - 'at sfObjectRoute->getObjectForParameters(array(''module'' =&gt; ''api'', ''action
sfPropelPlugin/lib/routing/sfPropelRoute.class.php line 100'
      - 'at sfPropelRoute->getObjectsForParameters(array(''module'' =&gt; ''api'', ''actio
/sfObjectRoute.class.php line 141'
      - 'at sfObjectRoute->getObjects() in SF_ROOT_DIR/apps/frontend/modules/api/actions/a
```

Figura 16.1. Página de error 404

Sugerencia

Si quieres redefinir las plantillas que incluye Symfony por defecto para el error `404` y las excepciones, tan sólo debes crear los archivos correspondientes en el directorio `config/error/`.

16.2. Probando los servicios web

Si quieres probar el nuevo servicio web que acabamos de crear, copia el archivo de datos de los afiliados del directorio `data/fixtures/` al directorio `test/fixtures/` y reemplaza el contenido del archivo `apiActionsTest.php` generado automáticamente por el siguiente código:

```
// test/functional/frontend/apiActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->
  info('1 - Web service security')->
```

```
  info('  1.1 - A token is needed to access the service')->
  get('/api/foo/jobs.xml')->
  with('response')->isStatusCode(404)->

  info('  1.2 - An inactive account cannot access the web service')->
  get('/api/symfony/jobs.xml')->
  with('response')->isStatusCode(404)->

  info('2 - The jobs returned are limited to the categories configured for the
affiliate')->
  get('/api/sensio_labs/jobs.xml')->
  with('request')->isFormat('xml')->
  with('response')->checkElement('job', 32)->

  info('3 - The web service supports the JSON format')->
  get('/api/sensio_labs/jobs.json')->
  with('request')->isFormat('json')->
  with('response')->contains('"category": "Programming"')->

  info('4 - The web service supports the YAML format')->
  get('/api/sensio_labs/jobs.yaml')->
  with('response')->begin()->
    isHeader('content-type', 'text/yaml; charset=utf-8')->
    contains('category: Programming')->
  end()
;
```

En el código anterior se utilizan por primera vez dos métodos que te pueden resultar útiles:

- `isFormat()`: comprueba el formato de la respuesta
- `contains()`: para el contenido que no sea HTML comprueba si la respuesta contiene el trozo de texto que se indica

16.3. El formulario para darse de alta como afiliado

Después de haber preparado el servicio web, el siguiente paso consiste en crear el formulario con el que los afiliados se van a dar de alta. Una vez más, vamos a describir paso a paso cómo añadir una nueva característica a la aplicación.

16.3.1. Sistema de enrutamiento

Como ya habrás adivinado, lo primero que hacemos es pensar en la URL de la nueva funcionalidad:

```
# apps/frontend/config/routing.yml
affiliate:
  class:   sfPropelRouteCollection
  options:
    model: JobeetAffiliate
    actions: [new, create]
    object_actions: { wait: get }
```

La ruta anterior es una colección de rutas de Propel que utiliza una nueva opción llamada `actions`. Como en este caso no necesitamos las siete acciones que define este tipo de ruta, la opción `actions` permite indicar las acciones para las que esta ruta debe funcionar (en el ejemplo anterior, sólo las acciones `new` y `create`). La ruta `wait` adicional se va a emplear para informar al afiliado sobre el estado de su cuenta.

16.3.2. Inicialización

A continuación, se genera automáticamente el módulo llamado `affiliate`:

```
$ php symfony propel:generate-module frontend affiliate JobeetAffiliate
--non-verbose-templates
```

16.3.3. Plantillas

La tarea `propel:generate-module` genera las acciones y plantillas de las siete acciones clásicas de las colecciones de rutas de Propel. Por tanto, entra en el directorio `templates/` del módulo y elimina todos los archivos salvo `_form.php` y `newSuccess.php`. En estos dos archivos, reemplaza su contenido por el siguiente código:

```
<!-- apps/frontend/modules/affiliate/templates/newSuccess.php -->
<?php use_stylesheet('job.css') ?>

<h1>Become an Affiliate</h1>

<?php include_partial('form', array('form' => $form)) ?>
<!-- apps/frontend/modules/affiliate/templates/_form.php -->
<?php include_stylesheets_for_form($form) ?>
<?php include_javascripts_for_form($form) ?>

<?php echo form_tag_for($form, 'affiliate') ?>
  <table id="job_form">
    <tfoot>
      <tr>
        <td colspan="2">
          <input type="submit" value="Submit" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

A continuación, crea la plantilla `waitSuccess.php` para la acción `wait` adicional:

```
<!-- apps/frontend/modules/affiliate/templates/waitSuccess.php -->
<h1>Your affiliate account has been created</h1>

<div style="padding: 20px">
  Thank you!
  You will receive an email with your affiliate token
```

```
  as soon as your account will be activated.
</div>
```

Por último, modifica el enlace del pie de página para que apunte al nuevo módulo `affiliate`:

```
// apps/frontend/templates/layout.php
<li class="last">
  <a href="<?php echo url_for('@affiliate_new') ?>">Become an affiliate</a>
</li>
```

16.3.4. Acciones

De nuevo, como sólo vamos a utilizar el formulario para crear nuevos afiliados, abre el archivo `actions.class.php` y elimina todos los métodos salvo `executeNew()`, `executeCreate()` y `processForm()`.

En la acción `processForm()`, modifica la URL de la redirección para que apunte a la acción `wait`:

```
// apps/frontend/modules/affiliate/actions/actions.class.php
$this->redirect($this->generateUrl('affiliate_wait', $jobeet_affiliate));
```

La propia acción `wait` es muy sencilla porque no tenemos que pasar ninguna variable a la plantilla:

```
// apps/frontend/modules/affiliate/actions/actions.class.php
public function executeWait()
{
}
```

Ahora mismo el usuario afiliado no puede ni elegir su token ni activar su cuenta. Por tanto, abre el archivo `JobeetAffiliateForm` para personalizar el formulario:

```
// lib/form/JobeetAffiliateForm.class.php
class JobeetAffiliateForm extends BaseJobeetAffiliateForm
{
  public function configure()
  {
    unset($this['is_active'], $this['token'], $this['created_at']);


$this->widgetSchema['jobeet_category_affiliate_list']->setOption('expanded',
true);

$this->widgetSchema['jobeet_category_affiliate_list']->setLabel('Categories');


$this->validatorSchema['jobeet_category_affiliate_list']->setOption('required',
true);

    $this->widgetSchema['url']->setLabel('Your website URL');
    $this->widgetSchema['url']->setAttribute('size', 50);

    $this->widgetSchema['email']->setAttribute('size', 50);
```

```
    $this->validatorSchema['email'] = new sfValidatorEmail(array('required' =>
true));
  }
}
```

El framework de formularios soporta las relaciones muchos-a-muchos. Por defecto, este tipo de relaciones se muestran en forma de lista desplegable mediante el widget `sfWidgetFormChoice`. Como ya vimos durante el tutorial del día 10, hemos cambiado la forma en la que se muestra este widget mediante la opción `expanded`.

Como los campos en los que se escriben emails y URL suelen ser más largos que el tamaño por defecto de la etiqueta `<input>`, hemos establecido nuevos atributos HTML con el método `setAttribute()`.

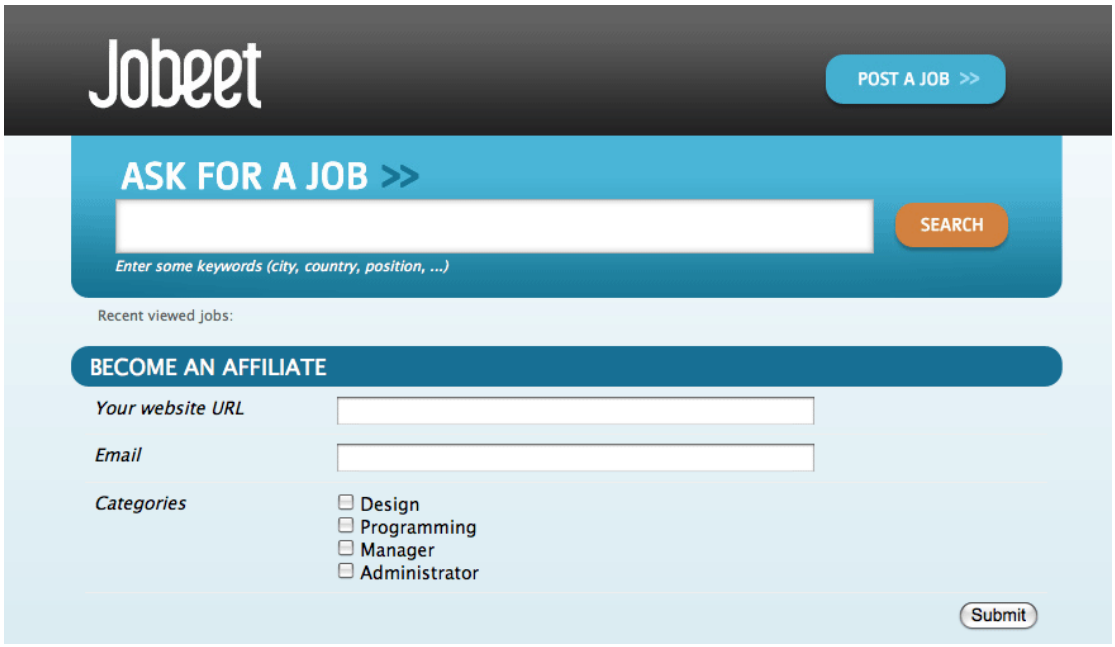

Figura 16.2. El formulario de los afiliados

16.3.5. Pruebas

Como siempre que añadimos una nueva característica a la aplicación, no te olvides de crear las pruebas funcionales correspondientes.

Reemplaza el contenido de las pruebas generadas automáticamente para el módulo `affiliate` por el siguiente código:

```
// test/functional/frontend/affiliateActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->
  info('1 - An affiliate can create an account')->
```

```
  get('/affiliate/new')->
  click('Submit', array('jobeet_affiliate' => array(
    'url'                           => 'http://www.example.com/',
    'email'                         => 'foo@example.com',
    'jobeet_category_affiliate_list' =>
array($browser->getProgrammingCategory()->getId()),
  )))->
  isRedirected()->
  followRedirect()->
  with('response')->checkElement('#content h1', 'Your affiliate account has
been created')->

  info('2 - An affiliate must at leat select one category')->

  get('/affiliate/new')->
  click('Submit', array('jobeet_affiliate' => array(
    'url'   => 'http://www.example.com/',
    'email' => 'foo@example.com',
  )))->
  with('form')->isError('jobeet_category_affiliate_list')
;
```

Para simular la selección de elementos de tipo *checkbox*, se pasa un array con los identificadores de los elementos que se quieren seleccionar. Para simplificar un poco más la tarea, hemos creado un método llamado `getProgrammingCategory()` en la clase `JobeetTestFunctional`:

```
// lib/model/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function getProgrammingCategory()
  {
    $criteria = new Criteria();
    $criteria->add(JobeetCategoryPeer::SLUG, 'programming');

    return JobeetCategoryPeer::doSelectOne($criteria);
  }

  // ...
}
```

No obstante, quizás recuerdes que ya tenemos este mismo código en el método `getMostRecentProgrammingJob()`, por lo que vamos a refactorizar ese código en un nuevo método llamado `getForSlug()` en la clase `JobeetCategoryPeer`:

```
// lib/model/JobeetCategoryPeer.php
static public function getForSlug($slug)
{
  $criteria = new Criteria();
  $criteria->add(self::SLUG, $slug);

  return self::doSelectOne($criteria);
}
```

No te olvides de modificar en la clase `JobeetTestFunctional` las dos veces que aparece el código anterior.

16.4. Administrando los afiliados

Como el administrador debe activar a cada afiliado, tenemos que crear en la aplicación `backend` un nuevo módulo llamado `affiliate`:

```
$ php symfony propel:generate-admin backend JobeetAffiliate --module=affiliate
```

Para que el administrador pueda acceder al nuevo módulo, añade un enlace en el menú principal que indique el número de afiliados que están pendientes de activar:

```
<!-- apps/backend/templates/layout.php -->
<li>
  <a href="<?php echo url_for('@jobeet_affiliate') ?>">
    Affiliates - <strong><?php echo JobeetAffiliatePeer::countToBeActivated()
?></strong>
  </a>
</li>
// lib/model/JobeetAffiliatePeer.php
class JobeetAffiliatePeer extends BaseJobeetAffiliatePeer
{
  static public function countToBeActivated()
  {
    $criteria = new Criteria();
    $criteria->add(self::IS_ACTIVE, 0);

    return self::doCount($criteria);
  }

  // ...

}
```

La única acción que necesitamos en el `backend` es la de activar o desactivar cuentas de afiliados, así que puedes modificar la sección `config` creada automáticamente por la tarea `propel:generate-admin` para simplificar un poco la interfaz y para añadir al listado un enlace que permita activar cuentas directamente:

```
# apps/backend/modules/affiliate/config/generator.yml
config:
  fields:
    is_active: { label: Active? }
  list:
    title:   Affiliate Management
    display: [is_active, url, email, token]
    sort:    [is_active]
    object_actions:
      activate:   ~
      deactivate: ~
    batch_actions:
      activate:   ~
      deactivate: ~
```

```
    actions: {}
  filter:
    display: [url, email, is_active]
```

Si quieres mejorar la productividad de los administradores, modifica los filtros por defecto para que muestren sólo los afiliados pendientes de activar:

```
// apps/backend/modules/affiliate/lib/affiliateGeneratorConfiguration.class.php
class affiliateGeneratorConfiguration extends
BaseAffiliateGeneratorConfiguration
{
  public function getFilterDefaults()
  {
    return array('is_active' => '0');
  }
}
```

El único código que tienes que escribir es el correspondiente a las acciones `activate` y `deactivate`:

```
// apps/backend/modules/affiliate/actions/actions.class.php
class affiliateActions extends autoAffiliateActions
{
  public function executeListActivate()
  {
    $this->getRoute()->getObject()->activate();

    $this->redirect('@jobeet_affiliate');
  }

  public function executeListDeactivate()
  {
    $this->getRoute()->getObject()->deactivate();

    $this->redirect('@jobeet_affiliate');
  }

  public function executeBatchActivate(sfWebRequest $request)
  {
    $affiliates =
JobeetAffiliatePeer::retrieveByPks($request->getParameter('ids'));

    foreach ($affiliates as $affiliate)
    {
      $affiliate->activate();
    }

    $this->redirect('@jobeet_affiliate');
  }

  public function executeBatchDeactivate(sfWebRequest $request)
  {
    $affiliates =
JobeetAffiliatePeer::retrieveByPks($request->getParameter('ids'));
```

```
    foreach ($affiliates as $affiliate)
    {
      $affiliate->deactivate();
    }

    $this->redirect('@jobeet_affiliate');
  }
}
// lib/model/JobeetAffiliate.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function activate()
  {
    $this->setIsActive(true);

    return $this->save();
  }

  public function deactivate()
  {
    $this->setIsActive(false);

    return $this->save();
  }

  // ...
}
```

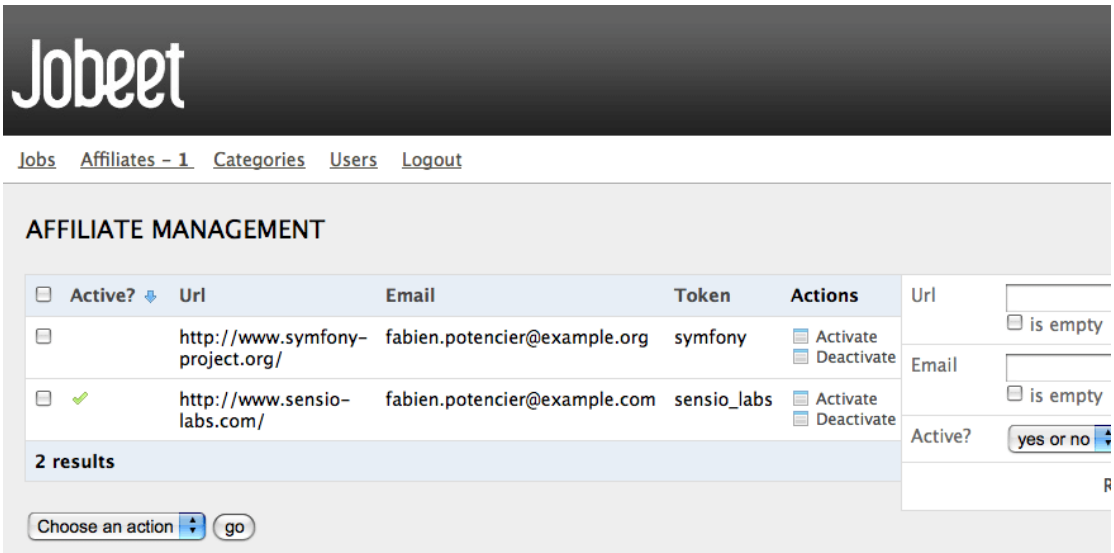

Figura 16.3. La parte de administración de los afiliados

16.5. Enviando emails

Cuando el administrador activa la cuenta de un afiliado, se debe mandar un email a ese usuario confirmándole su suscripción e indicándole cuál es su token.

PHP dispone de muchas librerías buenas para mandar emails, como por ejemplo SwiftMailer (http://www.swiftmailer.org/) , Zend_Mail (http://framework.zend.com/)

y ezcMail (http://ezcomponents.org/docs/tutorials/Mail) . Como en los tutoriales de los próximos días haremos uso de algunos componentes del Zend Framework, vamos a utilizar `Zend_Mail` para enviar los emails.

16.5.1. Instalación y configuración del Zend Framework

La librería `Zend_Mail` forma parte del Zend Framework. Como no queremos utilizar todos los componentes de este framework, vamos a instalar solamente los componentes necesarios en el directorio `lib/vendor/`, el mismo en el que instalamos Symfony.

En primer lugar, descarga el Zend Framework (http://framework.zend.com/download/overview) y descomprime sus archivos en el directorio `lib/vendor/Zend/`. A continuación, elimina todos los archivos y directorios salvo los siguientes, que son los que vamos a utilizar para enviar emails:

- `Exception.php`
- `Loader/`
- `Loader.php`
- `Mail/`
- `Mail.php`
- `Mime/`
- `Mime.php`
- `Search/`

Nota

El directorio `Search/` no lo necesitamos para enviar emails pero sí para el tutorial de mañana.

Después, añade el siguiente código en la clase `ProjectConfiguration` de tu proyecto para registrar el cargador automático de clases de Zend:

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  static protected $zendLoaded = false;

  static public function registerZend()
  {
    if (self::$zendLoaded)
    {
      return;
    }


set_include_path(sfConfig::get('sf_lib_dir').'/vendor'.PATH_SEPARATOR.get_include_path());
    require_once sfConfig::get('sf_lib_dir').'/vendor/Zend/Loader.php';
    Zend_Loader::registerAutoload();
```

```
    self::$zendLoaded = true;
  }

  // ...
}
```

16.5.2. Enviando emails

Modifica la acción `activate` para enviar un email cuando el administrador valida un afiliado:

```
// apps/backend/modules/affiliate/actions/actions.class.php
class affiliateActions extends autoAffiliateActions
{
  public function executeListActivate()
  {
    $affiliate = $this->getRoute()->getObject();
    $affiliate->activate();

    // send an email to the affiliate
    ProjectConfiguration::registerZend();
    $mail = new Zend_Mail();
    $mail->setBodyText(<<<EOF
Your Jobeet affiliate account has been activated.

Your token is {$affiliate->getToken()}.

The Jobeet Bot.
EOF
);
    $mail->setFrom('jobeet@example.com', 'Jobeet Bot');
    $mail->addTo($affiliate->getEmail());
    $mail->setSubject('Jobeet affiliate token');
    $mail->send();

    $this->redirect('@jobeet_affiliate');
  }

  // ...
}
```

Para que el código anterior funcione correctamente, modifica `jobeet@example.com` por una dirección de email válida.

Nota

El sitio web del Zend Framework incluye un completo tutorial sobre la librería Zend_Mail (http://framework.zend.com/manual/en/zend.mail.html) .

16.6. Nos vemos mañana

Gracias a la arquitectura REST de Symfony, es muy sencillo incluir servicios web en tus proyectos. Aunque en este tutorial sólo hemos creado un servicio web de consulta de

datos, ya tienes suficientes conocimientos de Symfony como para crear un servicio web de consulta y/o modificación de datos.

Como ya conoces el proceso de añadir nuevas funcionalidades en un proyecto, hoy ha sido realmente sencillo crear el formulario para que los afiliados se den de alta y el correspondiente gestor de usuarios afiliados.

Si recuerdas los requisitos que establecimos durante el día 2: *"los afiliados también pueden limitar el número de ofertas de trabajo del listado y pueden especificar una categoría para refinar la búsqueda"*.

Como este requisito es realmente sencillo, vamos a dejar que seas tu mismo el que lo implemente.

En el tutorial de mañana añadiremos un buscador, que será la última funcionalidad del sitio web de Jobeet.

Capítulo 17. El buscador

Hace dos días añadimos canales Atom para que los usuarios de Jobeet pudieran estar permanentemente informados de las últimas ofertas de trabajo publicadas. Hoy seguimos mejorando la experiencia de usuario añadiendo la última gran característica de Jobeet: el buscador.

17.1. La tecnología

Antes de ponernos manos a la obra, vamos a hablar brevemente de la historia de Symfony. Los creadores de Symfony somos partidarios de aplicar siempre las mejores prácticas, como pruebas y refactorización, y también intentamos incorporar estas buenas prácticas al desarrollo del propio framework.

Uno de los lemas que más nos gusta es el de *"No reinventes la rueda"*. De hecho, el framework Symfony inició su andadura hace cuatro años a partir de la unión de dos aplicaciones de software libre: Mojavi y Propel. De la misma forma, cada vez que nos enfrentamos a un problema, en vez de intentar resolverlo nosotros mismos, siempre buscamos en primer lugar alguna librería que ya exista y que resuelva correctamente ese problema.

Hoy queremos añadir un buscador a Jobeet y el Zend Framework incluye una librería fantástica llamada Zend Lucene (http://framework.zend.com/manual/en/zend.search.lucene.html) , que es una versión del conocido proyecto Lucene para Java. Como crear un buen buscador es realmente complicado, vamos a utilizar Zend Lucene en vez de intentar crear un buscador desde cero.

La propia documentación de Zend Lucene describe la librería de la siguiente forma:

"...un buscador genérico de texto escrito completamente con PHP 5. Como guarda sus índices en archivos y no requiere de un servidor de bases de datos, permite incluir un buscador en cualquier sitio web construido con PHP."

`Zend_Search_Lucene` incluye las siguientes características

- Búsqueda por ranking, que muestra primero los mejores resultados
- Soporta consultas mediante frases, consultas *booleanas*, consultas con comodines, consultas de proximidad, consultas basadas en rangos y muchos otros tipos de consultas
- Búsqueda por un campo específico, como por ejemplo título, autor o contenidos

Nota

Este capítulo no es un tutorial sobre la librería Zend Lucene, sino un tutorial sobre cómo integrar Zend Lucene en el sitio web de Jobeet y en general, un tutorial sobre cómo integrar librerías externas en proyectos Symfony. Si quieres conocer más sobre la tecnología de esta librería,

puedes consultar la documentación sobre Zend Lucene (http://framework.zend.com/manual/en/zend.search.lucene.html) disponible en el sitio web del Zend Framework.

Si seguiste el tutorial de ayer, ya tienes instalada la librería Zend Lucene como parte de la instalación de Zend Framework que realizamos ayer para enviar emails.

17.2. Indices

El buscador de Jobeet debe encontrar todas las ofertas de trabajo que coincidan de alguna manera con las palabras clave introducidas por los usuarios. Por ello, antes de poder realizar cualquier búsqueda, es necesario crear los índices con la información de las ofertas de trabajo. En el caso de Jobeet, los índices generados los vamos a guardar en el directorio `data/`

Zend Lucene incluye dos métodos para obtener un índice dependiendo de si ese índice ya existe o no. Vamos a crear un helper en la clase `JobeetJobPeer` que devuelve o crea un índice en función de si ya existía o no:

```
// lib/model/JobeetJobPeer.php
static public function getLuceneIndex()
{
  ProjectConfiguration::registerZend();

  if (file_exists($index = self::getLuceneIndexFile()))
  {
    return Zend_Search_Lucene::open($index);
  }
  else
  {
    return Zend_Search_Lucene::create($index);
  }
}

static public function getLuceneIndexFile()
{
  return
sfConfig::get('sf_data_dir').'/job.'.sfConfig::get('sf_environment').'.index';
}
```

17.2.1. El método save()

Cada vez que creamos, modificamos o borramos una oferta de trabajo, debemos actualizar el índice. Modifica la clase `JobeetJob` para que se actualice el índice cada vez que guardamos una oferta de trabajo en la base de datos:

```
// lib/model/JobeetJob.php
public function save(PropelPDO $con = null)
{
  // ...

  $ret = parent::save($con);
```

```
  $this->updateLuceneIndex();

  return $ret;
}
```

A continuación, crea el método `updateLuceneIndex()` que es realmente el que actualiza el índice:

```
// lib/model/JobeetJob.php
public function updateLuceneIndex()
{
  $index = JobeetJobPeer::getLuceneIndex();

  // remove an existing entry
  if ($hit = $index->find('pk:'.$this->getId()))
  {
    $index->delete($hit->id);
  }

  // don't index expired and non-activated jobs
  if ($this->isExpired() || !$this->getIsActivated())
  {
    return;
  }

  $doc = new Zend_Search_Lucene_Document();

  // store job primary key URL to identify it in the search results
  $doc->addField(Zend_Search_Lucene_Field::UnIndexed('pk', $this->getId()));

  // index job fields
  $doc->addField(Zend_Search_Lucene_Field::UnStored('position',
$this->getPosition(), 'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('company',
$this->getCompany(), 'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('location',
$this->getLocation(), 'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('description',
$this->getDescription(), 'utf-8'));

  // add job to the index
  $index->addDocument($doc);
  $index->commit();
}
```

Como Zend Lucene no es capaz de actualizar un registro existente en el índice, primero comprobamos si ya existía esa oferta de trabajo en el índice y en caso afirmativo, la eliminamos antes de volver a añadirla.

Indexar la información de una oferta de trabajo es muy sencillo: guardamos la clave primaria para utilizarla posteriormente en las búsquedas e indexamos el contenido de las columnas de datos principales (`position`, `company`, `location` y `description`). El

contenido de estas columnas se indexa pero no se guarda porque al mostrar los resultados de búsqueda utilizaremos los objetos reales.

17.2.2. Transacciones Propel

¿Qué sucede si surge un problema al indexar una oferta de trabajo o si la oferta no se guarda correctamente en la base de datos? En este caso, tanto Propel como Zend Lucene lanzan una excepción. No obstante, puede suceder que hayamos guardado una oferta de trabajo en la base de datos pero su información no se encuentre en el índice. Para evitar que esto ocurra, vamos a encerrar las dos actualizaciones de datos en una transacción que podremos anular en caso de error:

```
// lib/model/JobeetJob.php
public function save(PropelPDO $con = null)
{
  // ...

  if (is_null($con))
  {
    $con = Propel::getConnection(JobeetJobPeer::DATABASE_NAME,
Propel::CONNECTION_WRITE);
  }

  $con->beginTransaction();
  try
  {
    $ret = parent::save($con);

    $this->updateLuceneIndex();

    $con->commit();

    return $ret;
  }
  catch (Exception $e)
  {
    $con->rollBack();
    throw $e;
  }
}
```

17.2.3. El método delete()

Además de modificar el método `save()`, también tenemos que redefinir el método `delete()` para eliminar del índice el registro de la oferta de trabajo borrada:

```
// lib/model/JobeetJob.php
public function delete(PropelPDO $con = null)
{
  $index = JobeetJobPeer::getLuceneIndex();

  if ($hit = $index->find('pk:'.$this->getId()))
  {
```

```
    $index->delete($hit->id);
  }

  return parent::delete($con);
}
```

17.2.4. Borrados masivos

Cada vez que utilizas la tarea `propel:data-load` para cargar la información de los archivos de datos, Symfony borra todos los registros de las ofertas de trabajo en la base de datos con el método `JobeetJobPeer::doDeleteAll()`. A continuación, redefinimos este comportamiento por defecto para que también borre todo el índice de ofertas de trabajo:

```
// lib/model/JobeetJobPeer.php
public static function doDeleteAll($con = null)
{
  if (file_exists($index = self::getLuceneIndexFile()))
  {
    sfToolkit::clearDirectory($index);
    rmdir($index);
  }

  return parent::doDeleteAll($con);
}
```

17.3. Búsquedas

Ahora que ya tenemos todo preparado, vuelve a cargar los archivos de datos para que se cree el índice:

```
$ php symfony propel:data-load --env=dev
```

En esta ocasión, la tarea `propel:data-load` la ejecutamos con la opción `--env` porque el índice depende del entorno de ejecución y el entorno por defecto de las tareas es `cli`.

Sugerencia

Si eres usuario de sistemas operativos tipo Unix, ten en cuenta que el índice se modifica tanto desde la línea de comandos como desde la web, por lo que debes establecer los permisos adecuados al directorio donde guardas el índice. Comprueba tu configuración para que tanto el usuario de la línea de comandos como el usuario con el que se ejecuta el servidor web tengan permisos de escritura en el directorio de los índices.

Nota

Si no has compilado la extensión `zip` para tu PHP, puede que se muestren algunos mensajes de aviso sobre la clase `ZipArchive`. Se trata de un error conocido de la clase `Zend_Loader`.

Después de crear los índices, añadir el buscador en la aplicación `frontend` es realmente sencillo. Como siempre, primero crea la ruta asociada:

```
job_search:
  url:   /search
  param: { module: job, action: search }
```

A continuación, crea la acción correspondiente:

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    if (!$query = $request->getParameter('query'))
    {
      return $this->forward('job', 'index');
    }

    $this->jobs = JobeetJobPeer::getForLuceneQuery($query);
  }

  // ...
}
```

La plantilla asociada a esta acción también es muy sencilla:

```
// apps/frontend/modules/job/templates/searchSuccess.php
<?php use_stylesheet('jobs.css') ?>

<div id="jobs">
  <?php include_partial('job/list', array('jobs' => $jobs)) ?>
</div>
```

En realidad, la búsqueda se delega al método `getForLuceneQuery()`:

```
// lib/model/JobeetJobPeer.php
static public function getForLuceneQuery($query)
{
  $hits = self::getLuceneIndex()->find($query);

  $pks = array();
  foreach ($hits as $hit)
  {
    $pks[] = $hit->pk;
  }

  $criteria = new Criteria();
  $criteria->add(self::ID, $pks, Criteria::IN);
  $criteria->setLimit(20);

  return self::doSelect(self::addActiveJobsCriteria($criteria));
}
```

Después de obtener todos los resultados del índice de Lucene, filtramos las ofertas de trabajo que no están activas y limitamos el número de resultados a un máximo de `20`.

Para que el buscador esté completo, actualiza el layout:

```
// apps/frontend/templates/layout.php
<h2>Ask for a job</h2>
<form action="<?php echo url_for('@job_search') ?>" method="get">
  <input type="text" name="query" value="<?php echo
$sf_request->getParameter('query') ?>" id="search_keywords" />
  <input type="submit" value="search" />
  <div class="help">
    Enter some keywords (city, country, position, ...)
  </div>
</form>
```

Nota

Zend Lucene define su propio lenguaje para realizar consultas avanzadas que permite incluir operadores *booleanos*, comodines, búsquedas difusas y muchas otras cosas. Todas estas opciones están perfectamente documentadas (http://framework.zend.com/manual/en/zend.search.lucene.query-api.html) en el manual del Zend Framework.

17.4. Pruebas unitarias

¿Qué pruebas unitarias son las más recomendables para nuestro buscador? Obviamente no vamos a probar la propia librería Zend Lucene, sino su integración con la clase `JobeetJob`.

Para ello, añade las siguientes pruebas al final del archivo `JobeetJobTest.php` y no te olvides de actualizar a `7` el número de pruebas al principio del archivo:

```
// test/unit/model/JobeetJobTest.php
$t->comment('->getForLuceneQuery()');
$job = create_job(array('position' => 'foobar', 'is_activated' => false));
$job->save();

$jobs = JobeetJobPeer::getForLuceneQuery('position:foobar');
$t->is(count($jobs), 0, '::getForLuceneQuery() does not return non activated
jobs');

$job = create_job(array('position' => 'foobar', 'is_activated' => true));
$job->save();

$jobs = JobeetJobPeer::getForLuceneQuery('position:foobar');
$t->is(count($jobs), 1, '::getForLuceneQuery() returns jobs matching the
criteria');
$t->is($jobs[0]->getId(), $job->getId(), '::getForLuceneQuery() returns jobs
matching the criteria');

$job->delete();

$jobs = JobeetJobPeer::getForLuceneQuery('position:foobar');
$t->is(count($jobs), 0, '::getForLuceneQuery() does not return delete jobs');
```

Las pruebas anteriores comprueban que el índice no contenga ni ofertas de trabajo inactivas ni ofertas borradas. También comprobamos que los resultados de búsqueda muestran las ofertas de trabajo que coinciden con los criterios de búsqueda indicados.

17.5. Tareas

Tarde o temprano tendremos que crear una tarea que se encargue de *limpiar* el índice borrando las ofertas de trabajo expiradas y optimizando periódicamente el índice. Como ya disponemos de una tarea que se encarga de la limpieza de la base de datos, podemos actualizarla para que también se encargue del mantenimiento del índice:

```
// lib/task/JobeetCleanupTask.class.php
protected function execute($arguments = array(), $options = array())
{
  $databaseManager = new sfDatabaseManager($this->configuration);

  // cleanup Lucene index
  $index = JobeetJobPeer::getLuceneIndex();

  $criteria = new Criteria();
  $criteria->add(JobeetJobPeer::EXPIRES_AT, time(), Criteria::LESS_THAN);
  $jobs = JobeetJobPeer::doSelect($criteria);
  foreach ($jobs as $job)
  {
    if ($hit = $index->find('pk:'.$job->getId()))
    {
      $hit->delete();
    }
  }

  $index->optimize();

  $this->logSection('lucene', 'Cleaned up and optimized the job index');

  // Remove stale jobs
  $nb = JobeetJobPeer::cleanup($options['days']);

  $this->logSection('propel', sprintf('Removed %d stale jobs', $nb));
}
```

La tarea anterior ahora también elimina del índice todas las ofertas de trabajo expiradas y optimiza el índice gracias al método `optimize()` incluido en Zend Lucene.

17.6. Nos vemos mañana

Hoy hemos creado un completo buscador con muchas funcionalidades en menos de una hora. El tutorial de hoy también nos ha servido para explicar que cada vez que quieres añadir una nueva característica a tu aplicación, deberías comprobar que otros no la hayan resuelto anteriormente. Primero deberías comprobar si esa nueva característica no es algo que ya está incluido en la API de Symfony 1.2 (http://www.symfony-project.org/api/1_2/) .

Después, deberías comprobar que la nueva funcionalidad tampoco la resuelve ninguno de los plugins de Symfony (http://www.symfony-project.org/plugins/) . Por último, no te olvides de comprobar las librerías del Zend Framework

(http://framework.zend.com/manual/en/) y las librerías de ezComponent (http://ezcomponents.org/docs) .

Mañana añadiremos código JavaScript no intrusivo para mejorar el tiempo de respuesta del buscador actualizando los resultados en tiempo real a medida que el usuario escribe en el cuadro de búsqueda. Por tanto, mañana también hablaremos de cómo utilizar AJAX con Symfony.

Capítulo 18. AJAX

Ayer implementamos un buscador completo para Jobeet gracias a la librería Zend Lucene. Hoy vamos a mejorar el tiempo de respuesta del buscador mediante AJAX (http://es.wikipedia.org/wiki/AJAX) para convertir un buscador normal en un buscador en tiempo real.

Como el formulario de búsqueda debe funcionar tanto si se activa como si se desactiva JavaScript, vamos a incluir el buscador en tiempo real mediante JavaScript no intrusivo (http://es.wikipedia.org/wiki/JavaScript_no_obstructivo) . Además, utilizar JavaScript no intrusivo garantiza una mejor separación entre el código HTML, CSS y JavaScript de la parte de cliente de la aplicación.

18.1. Instalando jQuery

Como no queremos reinventar la rueda y perder el tiempo intentando solucionar las diferencias de comportamientos de JavaScript en cada navegador, vamos a utilizar una librería de JavaScript llamada jQuery (http://jquery.com/) . El framework Symfony no te obliga a utilizar ninguna librería concreta, ya que funciona con cualquier librería de JavaScript.

Accede al sitio web de jQuery (http://jquery.com/) , descarga su última versión y guarda el archivo JavaScript descargado en el directorio `web/js/`

18.2. Incluyendo jQuery

Como vamos a hacer uso de jQuery en todas las páginas, actualiza el layout para enlazar el archivo JavaScipt en la sección `<head>`. Ten en cuenta que debes insertar la función `use_javascript()` antes que la llamada a `include_javascripts()`:

```
<!-- apps/frontend/templates/layout.php -->

  <?php use_javascript('jquery-1.2.6.min.js') ?>
  <?php include_javascripts() ?>
</head>
```

Aunque podríamos haber enlazado el archivo de jQuery directamente con una etiqueta `<script>`, el uso del helper `use_javascript()` nos asegura que no incluimos en la página dos veces el mismo archivo de JavaScript.

Nota

Si quieres mejorar el rendimiento, puedes colocar el helper `include_javascripts()` justo antes de la etiqueta `</body>`, tal y como explican las reglas sobre rendimiento de aplicaciones web (http://developer.yahoo.com/performance/rules.html#js_bottom) elaboradas por Yahoo.

18.3. Añadiendo los comportamientos

Crear un buscador en tiempo real significa que cada vez que el usuario escribe un carácter en el cuadro de búsqueda debemos realizar una llamada al servidor. Posteriormente, el servidor devuelve la información necesaria para poder actualizar las zonas de la página donde se muestran los resultados sin tener que recargar completamente la página.

Aunque tradicionalmente los comportamientos de JavaScript se han incluido mediante los atributos `on*()` de HTML, el principio básico de funcionamiento de jQuery consiste en añadir los comportamientos de cada elemento después de que la página se ha cargado por completo. De esta forma, si deshabilitas JavaScript en el navegador, no se añade ningún comportamiento y el formulario sigue funcionando como un formulario normal.

En primer lugar, creamos una función para responder al evento que se produce cada vez que el usuario pulsa una tecla en el cuadro de búsqueda:

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    // do something
  }
});
```

Nota

No añadas todavía el código de JavaScript porque lo vamos a modificar muchas veces. En la próxima sección vamos a incluir el código JavaScript definitivo en el layout.

Cada vez que el usuario pulsa una tecla, jQuery ejecuta la función anónima definida en el código anterior. En nuestro caso, sólo realizamos una consulta al servidor si el usuario ha escrito más de tres caracteres o si el usuario ha borrado completamente el contenido del cuadro de búsqueda.

Realizar la llamada al servidor mediante AJAX es tan sencillo como utilizar el método `load()` sobre el elemento DOM que queremos actualizar:

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    $('#jobs').load(
      $(this).parents('form').attr('action'), { query: this.value + '*' } }
    );
  }
});
```

La parte de servidor que se encarga de responder a la petición AJAX es la misma acción que se ejecuta cuando se realizan peticiones *normales*. En la siguiente sección mostraremos los cambios necesarios en esa acción.

Por último, si JavaScript se encuentra activado, ocultamos el botón del formulario de búsqueda:

```
$('.search input[type="submit"]').hide();
```

18.4. Informando al usuario

Cuando se realizan peticiones AJAX, las páginas no se actualizan instantáneamente. El navegador espera la respuesta del servidor antes de poder actualizar los contenidos de la página. Por tanto, durante ese periodo de tiempo debemos mostrar algún tipo de indicación visual para informar al usuario de que ya se ha realizado la petición.

Una práctica muy extendida consiste en mostrar durante la petición AJAX un pequeño icono en movimiento. Por tanto, añade en el layout la imagen del icono y ocultala por defecto:

```
<!-- apps/frontend/templates/layout.php -->
<div class="search">
  <h2>Ask for a job</h2>
  <form action="<?php echo url_for('@job_search') ?>" method="get">
    <input type="text" name="query" value="<?php echo
$sf_request->getParameter('query') ?>" id="search_keywords" />
    <input type="submit" value="search" />
    <img id="loader" src="/images/loader.gif" style="vertical-align: middle;
display: none" />
    <div class="help">
      Enter some keywords (city, country, position, ...)
    </div>
  </form>
</div>
```

Nota

El icono está preparado para que quede bien en el layout actual de Jobeet. Si quieres crear tu propio icono, existen muchos sitios web que permiten hacerlo, como por ejemplo http://www.ajaxload.info/

Ahora que ya disponemos del código HTML completo para que el buscador en tiempo real funcione, crea un archivo llamado `search.js` que contenga todo el código JavaScript que hemos creado hasta el momento:

```
// web/js/search.js
$(document).ready(function()
{
  $('.search input[type="submit"]').hide();

  $('#search_keywords').keyup(function(key)
  {
    if (this.value.length >= 3 || this.value == '')
```

```
    {
      $('#loader').show();
      $('#jobs').load(
        $(this).parents('form').attr('action'),
        { query: this.value + '*' },
        function() { $('#loader').hide(); }
      );
    }
  });
});
```

También debes actualizar el layout para incluir este nuevo archivo JavaScript:

```
<!-- apps/frontend/templates/layout.php -->
<?php use_javascript('search.js') ?>
```

JavaScript como acción

Aunque el código JavaScript que hemos utilizado para el buscador es estático, en ocasiones los archivos JavaScript deben ser dinámicos para poder incluir algo de código PHP (como por ejemplo para utilizar el helper `url_for()`).

JavaScript no es más que otro formato y, como vimos hace algunos días, Symfony te permite trabajar con los formatos de forma sencilla. Como el archivo JavaScript contiene el comportamiento dinámico de una página, puedes utilizar la misma URL tanto para la página como para el archivo JavaScript (utilizando en este último caso la extensión `.js`). Si por ejemplo quieres crear un archivo JavaScript para definir el comportamiento del buscador, puedes modificar la ruta `job_search` de la siguiente forma y puedes crear una plantilla llamada `searchSuccess.js.php`:

```
job_search:
  url:   /search.:sf_format
  param: { module: job, action: search, sf_format: html }
  requirements:
    sf_format: (?:html|js)
```

18.5. AJAX en las acciones

Cuando JavaScript está activado, jQuery intercepta todas las teclas pulsadas por el usuario en el cuadro de búsqueda y realiza la llamada a la acción `search`. Si JavaScript no se encuentra activado, se ejecuta la misma acción `search` cuando el usuario envía el formulario pulsando la tecla `ENTER` o pulsando el botón *Search*.

Por tanto, la acción `search` necesita conocer si la petición se realiza mediante AJAX o no. Cuando una petición se realiza con AJAX, el método `isXmlHttpRequest()` del objeto de la petición devuelve `true`.

Nota

El método `isXmlHttpRequest()` funciona con todas las principales librerías de JavaScript, como por ejemplo Prototype, Mootools y jQuery.

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeSearch(sfWebRequest $request)
{
  if (!$query = $request->getParameter('query'))
  {
    return $this->forward('job', 'index');
  }

  $this->jobs = JobeetJobPeer::getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    return $this->renderPartial('job/list', array('jobs' => $this->jobs));
  }
}
```

Como jQuery no recarga la página y sólo reemplaza el contenido del elemento `#jobs` del DOM con el contenido de la respuesta del servidor, la página devuelta no debería estar decorada por el layout. Como este caso es el habitual, Symfony deshabilita por defecto el layout cuando la petición se realiza con AJAX.

Además, en vez de devolver la plantilla completa, sólo tenemos que devolver el contenido del elemento parcial `job/list`. El método `renderPartial()` de la acción anterior devuelve como respuesta el contenido del elemento parcial y no la plantilla completa.

Si el usuario borra todos los caracteres del cuadro de búsqueda o si la búsqueda no devuelve ningún resultado, vamos a mostrar un mensaje adecuado en lugar de la pantalla vacía que se muestra actualmente. Para que la acción devuelva una simple cadena de texto, podemos utilizar el método `renderText()`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeSearch(sfWebRequest $request)
{
  if (!$query = $request->getParameter('query'))
  {
    return $this->forward('job', 'index');
  }

  $this->jobs = JobeetJobPeer::getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    if ('*' == $query || !$this->jobs)
    {
      return $this->renderText('No results.');
    }
    else
    {
      return $this->renderPartial('job/list', array('jobs' => $this->jobs));
    }
  }
}
```

Sugerencia

Si quieres devolver el contenido de un componente en una acción, puedes utilizar el método `renderComponent()`.

18.6. Probando AJAX

Como el navegador de Symfony no puede simular el código JavaScript, tienes que echarle una mano cuando quieres realizar pruebas con peticiones AJAX. En otras palabras, tienes que añadir a mano la cabecera que jQuery y todas las demás librerías importantes de JavaScript incluyen cuando realizan una petición:

```
// test/functional/frontend/jobActionsTest.php
$browser->setHttpHeader('X_REQUESTED_WITH', 'XMLHttpRequest');
$browser->
  info('5 - Live search')->

  get('/search?query=sens*')->
  with('response')->begin()->
    checkElement('table tr', 3)->
  end()
;
```

El método `setHttpHeader()` establece una cabecera HTTP en la siguiente petición realizada con el navegador de Symfony.

18.7. Nos vemos mañana

Ayer utilizamos la librería Zend Lucene para incluir un completo buscador. Hoy hemos utilizado jQuery para mejorar su tiempo de respuesta. El framework Symfony incluye todas las herramientas básicas para crear fácilmente aplicaciones que siguen la arquitectura MVC y también se integra perfectamente con otros frameworks y librerías. Como ya hemos comentado varias veces, siempre deberías utilizar la herramienta más adecuada para tu trabajo.

Mañana nos dedicaremos a internacionalizar el sitio web de Jobeet.

Capítulo 19. Internacionalización y localización

Ayer terminamos de incluir el buscador en nuestra aplicación haciéndolo más interesante gracias a AJAX. Hoy vamos a hablar sobre la internacionalización (palabra que se suele abreviar por `i18n`) y la localización (abreviada como `l10n`).

Según la definición de la Wikipedia (http://es.wikipedia.org/wiki/Internacionalización_(computación)) :

"La internacionalización es el proceso de diseñar aplicaciones de software que puedan ser adaptadas a distintos idiomas y regiones sin necesidad de realizar cambios en su ingeniería."

"La localización es el proceso de adaptar el software para una región o idioma específicos mediante la inclusión de componentes específicos de esa región y mediante la traducción del texto."

Como siempre, Symfony no trata de reinventar la rueda y el soporte de i18n y l10n se basa en el estándar ICU (http://www.icu-project.org/) .

19.1. El usuario

La internacionalización no tiene ningún sentido sin los usuarios. Cuando un sitio web está disponible en varios idiomas o adaptado a varias regiones del mundo, el usuario es el responsable de seleccionar el idioma o región que más le guste.

Nota

Durante el tutorial del día 13 ya hablamos en detalle sobre la clase `sfUser` de Symfony.

19.1.1. La cultura del usuario

Las características de i18n y l10n de Symfony se basan en **la cultura del usuario**. La cultura es la combinación del idioma y el país/región del usuario. La cultura de un usuario que por ejemplo habla francés es `fr`, mientras que la cultura de un usuario de Francia es `fr_FR`.

Si quieres gestionar la cultura del usuario, puedes utilizar los métodos `setCulture()` y `getCulture()` del objeto que representa al usuario:

```
// in an action
$this->getUser()->setCulture('fr_BE');
echo $this->getUser()->getCulture();
```

Sugerencia

El idioma siempre se representa con dos letras minúsculas correspondientes al estándar ISO 639-1 (http://es.wikipedia.org/wiki/ISO_639-1) y el país se indica con dos letras mayúsculas que corresponden al estándar ISO 3166-1 (http://es.wikipedia.org/wiki/ISO_3166-1) .

19.1.2. La cultura por defecto

La cultura de usuario por defecto se configura en el archivo `settings.yml`:

```
# apps/frontend/config/settings.yml
all:
  .settings:
    default_culture: it_IT
```

Sugerencia

Como la cultura se gestiona a través del objeto `sfUser`, su valor se guarda en la sesión del usuario. Por tanto, si modificas la cultura durante el desarrollo de la aplicación, tienes que borrar la cookie de la sesión para que el navegador tenga en cuenta los cambios.

Cuando un usuario inicia una sesión en el sitio web de Jobeet, podemos determinar la cultura que mejor se adapta al usuario en función del valor de la cabecera `Accept-Language` de HTTP.

El método `getLanguages()` del objeto de la petición devuelve un array con los idiomas que acepta el usuario ordenados por preferencia:

```
// in an action
$languages = $request->getLanguages();
```

Por otra parte, seguramente los sitios web que desarrollas no están disponibles en los 136 principales idiomas del mundo. En este caso, puedes utilizar el método `getPreferredCulture()`, que devuelve el mejor idioma comparando los idiomas preferidos por el usuario y los idiomas que soporta tu sitio web:

```
// in an action
$language = $request->getPreferredCulture(array('en', 'fr'));
```

En el código anterior, el idioma devuelto será o inglés o francés en función del idioma preferido por el usuario. Si ninguno de los idiomas indicados coincide con los idiomas preferidos por el usuario, se devuelve el primer idioma del array (en el ejemplo anterior, sería el inglés).

19.2. Incluyendo la cultura en la URL

El sitio web de Jobeet está disponible en inglés y francés. Como una misma URL sólo puede representar un único recurso, debemos incluir la cultura como parte de la URL. Para ello, abre el archivo `routing.yml` y añade la variable especial `:sf_culture` en todas las rutas salvo en `api_jobs` y `homepage`. En las rutas sencillas, añade `/:sf_culture` al principio de la URL. En las colecciones de rutas, añade `/:sf_culture` al principio del valor de la opción `prefix_path`.

```
# apps/frontend/config/routing.yml
affiliate:
  class: sfPropelRouteCollection
  options:
    model:          JobeetAffiliate
    actions:        [new, create]
    object_actions: { wait: get }
    prefix_path:    /:sf_culture/affiliate

category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfPropelRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object }
  requirements:
    sf_format: (?:html|atom)

job_search:
  url:   /:sf_culture/search
  param: { module: job, action: search }

job:
  class: sfPropelRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: put, extend: put }
    prefix_path:    /:sf_culture/job
  requirements:
    token: \w+

job_show_user:
  url:     /:sf_culture/job/:company_slug/:location_slug/:id/:position_slug
  class:   sfPropelRoute
  options:
    model: JobeetJob
    type:  object
    method_for_criteria: doSelectActive
  param:   { module: job, action: show }
  requirements:
    id:        \d+
    sf_method: get
```

Cuando se incluye la variable `sf_culture` en una ruta, Symfony utiliza su valor para modificar automáticamente la cultura del usuario.

Como tenemos tantas portadas como idiomas soportados por la aplicación (`/en/`, `/fr/`, ...), la portada por defecto (`/`) debe redirigir al usuario a la portada adecuada en función de su cultura. Sin embargo, si es la primera vez que el usuario entra en Jobeet, el usuario todavía no tiene definida su cultura, por lo que debemos elegir la cultura que mejor se adapte al usuario.

En primer lugar, añade el método `isFirstRequest()` en la clase `myUser`. Se trata de un método sencillo que devuelve `true` sólo para la primera petición realizada en cada sesión de usuario:

```
// apps/frontend/lib/myUser.class.php
public function isFirstRequest($boolean = null)
{
  if (is_null($boolean))
  {
    return $this->getAttribute('first_request', true);
  }
  else
  {
    $this->setAttribute('first_request', $boolean);
  }
}
```

Añade también una ruta llamada `localized_homepage`:

```
# apps/frontend/config/routing.yml
localized_homepage:
  url:   /:sf_culture/
  param: { module: job, action: index }
  requirements:
    sf_culture: (?:fr|en)
```

A continuación, modifica la acción `index` del módulo `job` para incluir la lógica que se encarga de redirigir al usuario a la mejor portada cuando realiza la primera petición de su sesión de usuario:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeIndex(sfWebRequest $request)
{
  if (!$request->getParameter('sf_culture'))
  {
    if ($this->getUser()->isFirstRequest())
    {
      $culture = $request->getPreferredCulture(array('en', 'fr'));
      $this->getUser()->setCulture($culture);
      $this->getUser()->isFirstRequest(false);
    }
    else
    {
      $culture = $this->getUser()->getCulture();
    }

    $this->redirect('@localized_homepage');
  }

  $this->categories = JobeetCategoryPeer::getWithJobs();
}
```

Si no existe la variable `sf_culture` en la petición, eso significa que el usuario ha entrado en la URL `/`. Si estamos en ese caso y la sesión es nueva, se utiliza la cultura preferida por el usuario. En otro caso, se sigue utilizando la cultura actual del usuario.

El último paso consiste en redirigir al usuario a la ruta `localized_homepage`. Si te fijas en el código anterior, en la redirección no hemos incluido el valor de la variable `sf_culture`, ya que Symfony se encarga de añadirla automáticamente.

Si ahora intentas acceder a la URL `/it/`, Symfony devuelve un error de tipo `404` porque hemos restringido los posibles valores de la variable `sf_culture` a `en` o `fr`. Por tanto, añade este requerimiento en todas las rutas que incluyen la cultura:

```
requirements:
  sf_culture: (?:fr|en)
```

19.3. Probando la cultura

Ha llegado la hora de probar lo que hemos añadido a la aplicación. Pero antes de añadir más pruebas, vamos a arreglar las que ya tenemos. Como hemos modificado las URL, tenemos que modificar los archivos con pruebas funcionales que se encuentran en el directorio `test/functional/frontend/` y tenemos que añadir `/en` al principio de todas las URL. No te olvides de cambiar también las URL del archivo `lib/test/JobeetTestFunctional.class.php`. Después de realizar los cambios, ejecuta todas las pruebas para asegurarte de que has hecho bien las modificaciones:

```
$ php symfony test:functional frontend
```

El *tester* de los usuarios incluye un método llamado `isCulture()` que permite probar la cultura del usuario. Abre el archivo `jobActionsTest` y añade las siguientes pruebas:

```
// test/functional/frontend/jobActionsTest.php
$browser->setHttpHeader('ACCEPT_LANGUAGE', 'fr_FR,fr,en;q=0.7');
$browser->
  info('6 - User culture')->

  restart()->

  info('  6.1 - For the first request, symfony guesses the best culture')->
  get('/')->
  isRedirected()->followRedirect()->
  with('user')->isCulture('fr')->

  info('  6.2 - Available cultures are en and fr')->
  get('/it/')->
  with('response')->isStatusCode(404)
;

$browser->setHttpHeader('ACCEPT_LANGUAGE', 'en,fr;q=0.7');
$browser->
  info('  6.3 - The culture guessing is only for the first request')->

  get('/')->
```

```
  isRedirected()->followRedirect()->
  with('user')->isCulture('fr')
;
```

19.4. Cambiando de idioma

Para que el usuario pueda modificar su cultura, debemos incluir en el layout un formulario para cambiar de idioma. El framework de formularios de Symfony no incluye por defecto un formulario de este tipo, pero como se trata de algo bastante común para los sitios web disponibles en varios idiomas, los propios creadores de Symfony mantienen un plugin llamado sfFormExtraPlugin (http://www.symfony-project.org/plugins/sfFormExtraPlugin?tab=plugin_readme) que contiene validadores, widgets y formularios que son útiles pero que no se incluyen por defecto en Symfony porque son demasiado específicos o contienen dependencias externas.

Instala el plugin mediante la tarea `plugin:install`:

```
$ php symfony plugin:install sfFormExtraPlugin
```

No te olvides de borrar la cache de Symfony porque este plugin define clases nuevas:

```
$ php symfony cc
```

Nota

El plugin `sfFormExtraPlugin` contiene widgets que incluyen dependencias externas con librerías de JavaScript. Entre otros, este plugin contiene un editor avanzado de fechas y un editor de textos WYSIWYG. Te recomendamos que leas la documentación del plugin para descubrir cosas muy interesantes.

El plugin `sfFormExtraPlugin` incluye `sfFormLanguage`, un tipo de formulario que permite seleccionar el idioma de la aplicación. A continuación se muestra cómo puedes añadir el formulario del idioma en el layout:

Nota

El código que se muestra a continuación no es la forma más adecuada de incluir el formulario. Incluimos este código para mostrar la forma equivocada de incluir este formulario. Más adelante se muestra cómo incluir bien el formulario en la aplicación Symfony.

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <!-- footer content -->

    <?php $form = new sfFormLanguage(
      $sf_user,
      array('languages' => array('en', 'fr'))
      )
    ?>
    <form action="<?php echo url_for('@change_language') ?>">
      <?php echo $form ?><input type="submit" value="ok" />
    </form>
```

```
  </div>
</div>
```

¿Te has dado cuenta del error? Efectivamente, crear el objeto del formulario no es algo propio de la capa de la vista. Este objeto se debe crear en la acción. Como el código se ha incluido en el layout, el formulario se crea en cada acción, algo que no es nada práctico. En estos casos, debes utilizar un **componente**. Los componentes son como los elementos parciales pero con código asociado. Se podría considerar que un componente es como una acción muy simplificada.

Los componentes definidos por las plantillas se incluyen en el layout mediante el helper `include_component()`:

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <!-- footer content -->

    <?php include_component('language', 'language') ?>
  </div>
</div>
```

Los argumentos del helper `include_component()` son el nombre del módulo y el nombre de la acción. Se puede utilizar un tercer argumento opcional para pasar parámetros al componente.

Crea un módulo llamado `language` para poder definir el componente y la acción que van a modificar el idioma del usuario:

```
$ php symfony generate:module frontend language
```

Los componentes se definen en el archivo `actions/components.class.php`. Crea ese archivo y añade lo siguiente:

```
// apps/frontend/modules/language/actions/components.class.php
class languageComponents extends sfComponents
{
  public function executeLanguage(sfWebRequest $request)
  {
    $this->form = new sfFormLanguage(
      $this->getUser(),
      array('languages' => array('en', 'fr'))
    );
  }
}
```

Como se puede observar en el código anterior, la clase de los componentes es muy similar a la clase de las acciones.

Además, el nombre de la plantilla de un componente sigue las mismas convenciones que en los elementos parciales: un guión bajo (_) seguido por el nombre del componente:

```
// apps/frontend/modules/language/templates/_language.php
<form action="<?php echo url_for('@change_language') ?>">
```

```
  <?php echo $form ?><input type="submit" value="ok" />
</form>
```

Como el plugin no incluye la acción que realmente cambia la cultura del usuario, modifica el archivo `routing.yml` para crear una nueva ruta llamada `change_language`:

```
# apps/frontend/config/routing.yml
change_language:
  url:   /change_language
  param: { module: language, action: changeLanguage }
```

Y después se crea la acción correspondiente:

```
// apps/frontend/modules/language/actions/actions.class.php
class languageActions extends sfActions
{
  public function executeChangeLanguage(sfWebRequest $request)
  {
    $form = new sfFormLanguage(
      $this->getUser(),
      array('languages' => array('en', 'fr'))
    );

    $form->process($request);

    return $this->redirect('@localized_homepage');
  }
}
```

El método `process()` del formulario `sfFormLanguage` se encarga de modificar la cultura del usuario en función de la información enviada por el usuario.

Figura 19.1. Pie de página internacionalizado

19.5. Internacionalización

19.5.1. Idiomas, codificaciones y conjuntos de caracteres

Cada idioma define su propio conjunto de caracteres. El idioma inglés es el más sencillo porque sólo utiliza los caracteres ASCII. Otros idiomas como el francés son más complicados porque utilizan por ejemplo caracteres acentuados como é. Por último, idiomas como el ruso, el chino o el árabe son mucho más complicados porque todos sus caracteres se encuentran fuera del conjunto de caracteres ASCII. Estos últimos idiomas definen conjuntos de caracteres completamente diferentes.

Cuando se trabaja con aplicaciones internacionalizadas, es mejor seguir la norma *unicode*. La idea del estándar unicode consiste en crear un conjunto universal de caracteres que incluya todos los caracteres de todos los idiomas de la humanidad. El problema de unicode es que, debido a este enorme conjunto de caracteres, cada carácter puede llegar a necesitar hasta 21 bits para ser representado. Por tanto, para las aplicaciones web utilizamos UTF-8, que transforma los caracteres de Unicode en secuencias de octetos de longitud variable. Empleando UTF-8, los caracteres de los idiomas más utilizados en el mundo se representan con menos de 3 bits cada uno.

UTF-8 es la codificación que utiliza por defecto Symfony, tal y como se establece en el archivo de configuración `settings.yml`:

```
# apps/frontend/config/settings.yml
all:
  .settings:
    charset: utf-8
```

Además, para activar la internacionalización en Symfony, debes establecer la opción `i18n` a un valor `on` en el archivo de configuración `settings.yml`:

```
# apps/frontend/config/settings.yml
all:
  .settings:
    i18n: on
```

19.5.2. Plantillas

Un sitio web internacionalizado es aquel cuya interfaz de usuario se traduce a varios idiomas.

En las plantillas, las cadenas de texto que dependen del idioma utilizado se deben encerrar con el helper `__()` (cuidado al escribir el helper porque son dos guiones bajos seguidos).

El helper `__()` es parte del grupo de helpers `I18N`, que contiene helpers que facilitan el trabajo con la internacionalización de las plantillas. Como este grupo de helpers no se carga por defecto, debes incluirlo manualmente en la plantilla mediante `use_helper('I18N')` (como ya hicimos en su día para el grupo de helpers `Text`) o puedes cargarlo de forma global en la aplicación utilizando la opción `standard_helpers`:

```
# apps/frontend/config/settings.yml
all:
  .settings:
    standard_helpers: [Partial, Cache, I18N]
```

El siguiente código muestra cómo utilizar el helper `__()` en el pie de página de Jobeet:

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <span class="symfony">
      <img src="/images/jobeet-mini.png" />
```

```
      powered by <a href="http://www.symfony-project.org/">
      <img src="/images/symfony.gif" alt="symfony framework" /></a>
    </span>
    <ul>
      <li>
        <a href=""><?php echo __('About Jobeet') ?></a>
      </li>
      <li class="feed">
        <?php echo link_to(__('Full feed'), '@job?sf_format=atom') ?>
      </li>
      <li>
        <a href=""><?php echo __('Jobeet API') ?></a>
      </li>
      <li class="last">
        <?php echo link_to(__('Become an affiliate'), '@affiliate_new') ?>
      </li>
    </ul>
    <?php include_component('language', 'language') ?>
  </div>
</div>
```

Nota

Al helper `__()` se le puede pasar como argumento la cadena de texto mostrada para el idioma por defecto o también se le puede pasar el identificador único de cada cadena. Elegir una u otra opción es simplemente una cuestión de gusto personal. En Jobeet vamos a utilizar la primera forma porque así las plantillas son mucho más fáciles de leer.

Cuando Symfony procesa la plantilla para mostrarla, cada vez que encuentra una llamada al helper `__()`, Symfony busca la traducción de la cadena de texto para la cultura actual del usuario. Si se encuentra la traducción, se muestra directamente en la plantilla. Si no se encuentra la traducción, se devuelve el primer argumento del helper `__()`.

Las traducciones se guardan en catálogos. El framework de internacionalización de Symfony incluye muchas formas de guardar las traducciones. En este caso vamos a utilizar el formato XLIFF (http://es.wikipedia.org/wiki/XLIFF) , que es un estándar internacional y también es el más flexible. Además, XLIFF es el formato utilizado por el generador de la parte de administración y por la mayoría de plugins de Symfony.

Nota

Las otras formas de guardar los catálogos son `gettext`, `MySQL` y `SQLite`. Como siempre, no te olvides de echar un vistazo a la API de i18n (http://www.symfony-project.org/api/1_2/i18n) para descubrir todos los detalles.

19.5.3. La tarea i18n:extract

Si no quieres crear el catálogo a mano, puedes utilizar la tarea `i18n:extract`:

```
$ php symfony i18n:extract frontend fr --auto-save
```

La tarea `i18n:extract` del ejemplo anterior busca todas las cadenas de texto que deben traducirse al idioma `fr` en la aplicación `frontend` y crea o actualiza el catálogo correspondiente. La opción `--auto-save` hace que se guarden en el catálogo las nuevas cadenas de texto. También puedes hacer uso de la opción `--auto-delete` para eliminar automáticamente todas las cadenas de texto que ya no existen.

En nuestro caso, la tarea anterior añade todas las cadenas de texto al archivo que hemos creado:

```
<!-- apps/frontend/i18n/fr/messages.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN"
  "http://www.oasis-open.org/committees/xliff/documents/xliff.dtd">
<xliff version="1.0">
  <file source-language="EN" target-language="fr" datatype="plaintext"
      original="messages" date="2008-12-14T12:11:22Z"
      product-name="messages">
    <header/>
    <body>
      <trans-unit id="1">
        <source>About Jobeet</source>
        <target/>
      </trans-unit>
      <trans-unit id="2">
        <source>Feed</source>
        <target/>
      </trans-unit>
      <trans-unit id="3">
        <source>Jobeet API</source>
        <target/>
      </trans-unit>
      <trans-unit id="4">
        <source>Become an affiliate</source>
        <target/>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Cada traducción se define mediante una etiqueta `trans-unit` que tiene un identificador único en forma de atributo `id`. Ahora ya puedes modificar ese archivo para añadir las traducciones al francés:

```
<!-- apps/frontend/i18n/fr/messages.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN"
  "http://www.oasis-open.org/committees/xliff/documents/xliff.dtd">
<xliff version="1.0">
  <file source-language="EN" target-language="fr" datatype="plaintext"
      original="messages" date="2008-12-14T12:11:22Z"
      product-name="messages">
    <header/>
    <body>
      <trans-unit id="1">
```

```
        <source>About Jobeet</source>
        <target>A propos de Jobeet</target>
      </trans-unit>
      <trans-unit id="2">
        <source>Feed</source>
        <target>Fil RSS</target>
      </trans-unit>
      <trans-unit id="3">
        <source>Jobeet API</source>
        <target>API Jobeet</target>
      </trans-unit>
      <trans-unit id="4">
        <source>Become an affiliate</source>
        <target>Devenir un affilié</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Sugerencia

Como XLIFF es un formato estándar, existen muchas herramientas que facilitan el proceso de traducción. Open Language Tools (https://open-language-tools.dev.java.net/) es un proyecto de software libre creado con Java que incluye un editor de archivos en formato XLIFF.

Sugerencia

Como XLIFF es un formato basado en archivos de texto, se le aplican las mismas reglas de la configuración en cascada que se utiliza para los archivos de configuración de Symfony. Se pueden definir archivos i18n a nivel de proyecto, aplicación y módulo, aplicándose siempre la traducción del archivo más específico.

19.5.4. Traducciones con variables

El principal objetivo de la internacionalización consiste en traducir frases enteras. No obstante, algunas frases incluyen partes variables. En Jobeet, este caso se produce con los enlaces *"and X more..."* de la portada, donde `X` es el número de ofertas de trabajo disponibles:

```
// apps/frontend/modules/job/templates/indexSuccess.php
<div class="more_jobs">
  and <?php echo link_to($count, 'category', $category) ?> more...
</div>
```

Como el número de ofertas de trabajo es variable, en la traducción tenemos que sustituirlo por una variable:

```
// apps/frontend/modules/job/templates/indexSuccess.php
<div class="more_jobs">
  <?php echo __('and %count% more...', array('%count%' => link_to($count,
'category', $category))) ?>
</div>
```

Ahora la cadena de texto que tenemos que traducir es `and %count% more...`, siendo `%count%` la variable que se va a sustituir por el número de ofertas de trabajo indicado como segundo argumento del helper `__()`.

Añade la nueva cadena de texto en una etiqueta `trans-unit` del archivo `messages.xml`, o utiliza la tarea `i18n:extract` para actualizar el archivo automáticamente:

```
$ php symfony i18n:extract frontend fr --auto-save
```

Después de ejecutar la tarea, abre el archivo XLIFF y añade la correspondiente traducción al francés:

```
<trans-unit id="5">
  <source>and %count% more...</source>
  <target>et %count% autres...</target>
</trans-unit>
```

El único requisito de la traduccón es que debes utilizar en algún sitio la variable `%count%`.

Traducir otras cadenas de texto puede llegar a ser muy complicado por el uso de los plurales. Estas cadenas de texto cambian en función del valor de algunos números. Además, el comportamiento de los plurales no es idéntico en todos los idiomas, ya que idiomas como el ruso o el polaco tienen reglas gramaticales muy complejas para los plurales.

En la página de cada categoría, se muestra el número de ofertas de trabajo disponibles para esa categoría:

```
// apps/frontend/modules/category/templates/showSuccess.php
<strong><?php echo $pager->getNbResults() ?></strong> jobs in this category
```

Cuando la traducción de una cadena de texto es diferente en función del valor de un número, debes utilizar el helper `format_number_choice()`:

```
<?php echo format_number_choice(
    '[0]No job in this category|[1]One job in this category|(1,+Inf]%count%
jobs in this category',
    array('%count%' => '<strong>'.$pager->getNbResults().'</strong>'),
    $pager->getNbResults()
  )
?>
```

El helper `format_number_choice()` requiere tres argumentos:

- La cadena de texto que se utiliza en función del número
- Un array con las sustituciones de la parte variable
- El número empleado para determinar la traducción que se utiliza

La cadena que establece las diferentes traducciones a utilizar en función del valor del número emplea el siguiente formato:

- Cada posible traducción se separa de las demás mediante una barra vertical (|)

- Cada cadena de texto está formada por un rango seguido de una traducción

El rango puede describir cualquier tipo de rango numérico:

- `[1,2]`: acepta todos los valores entre 1 y 2, incluyendo 1 y 2
- `(1,2)`: acepta todos los valores entre 1 y 2, salvo 1 y 2
- `{1,2,3,4}`: sólo acepta los números indicados en ese conjunto de valores
- `[-Inf,0)`: acepta valores mayores o iguales que -infinito y estrictamente inferiores a `0`
- `{n: n % 10 > 1 && n % 10 < 5}`: acepta números como 2, 3, 4, 22, 23, 24, etc.

Traducir esta cadena de texto es similar a traducir cualquier otra cadena:

```
<trans-unit id="6">
  <source>[0]No job in this category|[1]One job in this
category|(1,+Inf]%count% jobs in this category</source>
  <target>[0]Aucune annonce dans cette catégorie|[1]Une annonce dans cette
catégorie|(1,+Inf]%count% annonces dans cette catégorie</target>
</trans-unit>
```

Ahora que ya sabes cómo traducir cualquier tipo de cadena de texto, dedica un tiempo a añadir llamadas al helper `__()` en todas las plantillas de la aplicación `frontend`. Por el momento no vamos a traducir la aplicación `backend`.

19.5.5. Formularios

Las clases de los formularios incluyen muchas cadenas de texto que tenemos que traducir, como etiquetas, mensajes de error y mensajes de ayuda. Symfony se encarga de internacionalizar automáticamente todas estas cadenas de texto, por lo que sólo es necesario que definas la traducción en los archivos XLIFF.

> **Nota**
>
> Desafortunadamente, la tarea `i18n:extract` no es capaz por el momento de procesar las clases de los formularios en busca de cadenas de texto sin traducir.

19.5.6. Objetos Propel

En el sitio web de Jobeet no vamos a traducir el contenido de todas las tablas porque no tiene sentido que los usuarios que publican ofertas de trabajo tengan que traducir sus ofertas a todos los idiomas disponibles. No obstante, sí que vamos a traducir el contenido de la tabla `category`.

El plugin de Propel ya incluye el soporte de tablas internacionalizadas. Por cada tabla que vamos a traducir, tenemos que crear dos tablas: una para las columnas que son independientes de la internacionalización y otra para todas las columnas cuyos valores se van a traducir. Las dos tablas están relacionadas mediante una relación de tipo uno-a-muchos.

Por lo tanto, actualiza el archivo `schema.yml` para crear las dos tablas relacionadas con las categorías:

```
# config/schema.yml
jobeet_category:
  _attributes:  { isI18N: true, i18nTable: jobeet_category_i18n }
  id:           ~

jobeet_category_i18n:
  id:           { type: integer, required: true, primaryKey: true,
foreignTable: jobeet_category, foreignReference: id }
  culture:      { isCulture: true, type: varchar, size: 7, required: true,
primaryKey: true }
  name:         { type: varchar(255), required: true }
  slug:         { type: varchar(255), required: true }
```

La opción `_attributes` define las opciones de la tabla. Después de modificar el esquema, actualiza la parte de las categorías en los archivos de datos:

```
# data/fixtures/010_categories.yml
JobeetCategory:
  design:        { }
  programming:   { }
  manager:       { }
  administrator: { }

JobeetCategoryI18n:
  design_en:        { id: design, culture: en, name: Design }
  programming_en:   { id: programming, culture: en, name: Programming }
  manager_en:       { id: manager, culture: en, name: Manager }
  administrator_en: { id: administrator, culture: en, name: Administrator }

  design_fr:        { id: design, culture: fr, name: Design }
  programming_fr:   { id: programming, culture: fr, name: Programmation }
  manager_fr:       { id: manager, culture: fr, name: Manager }
  administrator_fr: { id: administrator, culture: fr, name: Administrateur }
```

A continuación, vuelve a generar las clases del modelo para que se creen las clases relacionadas con la internacionalización:

```
$ php symfony propel:build-all --no-confirmation
$ php symfony cc
```

Como las columnas `name` y `slug` se han movido a la tabla internacionalizada, mueve el método `setName()` de `JobeetCategory` a `JobeetCategoryI18n`:

```
// lib/model/JobeetCategoryI18n.php
public function setName($name)
{
  parent::setName($name);

  $this->setSlug(Jobeet::slugify($name));
}
```

También debemos arreglar el método `getForSlug()` de la clase `JobeetCategoryPeer`:

```
// lib/model/JobeetCategoryPeer.php
static public function getForSlug($slug)
{
  $criteria = new Criteria();
  $criteria->addJoin(JobeetCategoryI18nPeer::ID, self::ID);
  $criteria->add(JobeetCategoryI18nPeer::CULTURE, 'en');
  $criteria->add(JobeetCategoryI18nPeer::SLUG, $slug);

  return self::doSelectOne($criteria);
}
```

Sugerencia

Como la tarea `propel:build-all` borra todas las tablas y toda la información de la base de datos, no te olvides de volver a crear un usuario para acceder a la parte de administración de Jobeet mediante la tarea `guard:create-user`. Si lo prefieres, puedes crear un archivo de datos para añadir este usuario de forma automática.

Después de construir el modelo, verás que Symfony crea métodos en el objeto `JobeetCategory` principal para acceder a las columnas internacionalizadas definidas en la clase `JobeetCategoryI18n`:

```
$category = new JobeetCategory();

$category->setName('foo');       // sets the name for the current culture
$category->setName('foo', 'fr'); // sets the name for French

echo $category->getName();     // gets the name for the current culture
echo $category->getName('fr'); // gets the name for French
```

Sugerencia

Si quieres reducir el número de consultas a la base de datos, utiliza el método `doSelectWithI18n()` en vez del tradicional método `doSelect()`. Este nuevo método obtiene en una sola consulta el objeto principal y el objeto internacionalizado asociado.

```
$categories = JobeetCategoryPeer::doSelectWithI18n($c, $culture);
```

Como la ruta `category` está asociada a la clase `JobeetCategory` del modelo y como `slug` ahora es parte de `JobeetCategoryI18n`, la ruta no es capaz de obtener el objeto `Category` automáticamente. Vamos a crear un método para ayudar al sistema de enrutamiento a obtener el objeto:

```
// lib/model/JobeetCategoryPeer.php
class JobeetCategoryPeer extends BaseJobeetCategoryPeer
{
  static public function doSelectForSlug($parameters)
  {
    $criteria = new Criteria();
    $criteria->addJoin(JobeetCategoryI18nPeer::ID, JobeetCategoryPeer::ID);
    $criteria->add(JobeetCategoryI18nPeer::CULTURE, $parameters['sf_culture']);
    $criteria->add(JobeetCategoryI18nPeer::SLUG, $parameters['slug']);

    return self::doSelectOne($criteria);
```

```
  }

  // ...
}
```

Después, utiliza la opción `method` en la ruta `category` para indicar que `doSelectForSlug()` es el método que se debe utilizar para obtener el objeto:

```
# apps/frontend/config/routing.yml
category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfPropelRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object, method: doSelectForSlug }
  requirements:
    sf_format: (?:html|atom)
```

Por último, volvemos a cargar los archivos de datos para que se generen los slugs adecuados para cada categoría:

```
$ php symfony propel:data-load
```

Después de todos estos cambios, la ruta `category` ya está internacionalizada y la URL de una categoría incluye la traducción del `slug` correspondiente:

```
/frontend_dev.php/fr/category/programmation
/frontend_dev.php/en/category/programming
```

19.5.7. El generador de la parte de administración

Debido a un error en la versión 1.2.1 de Symfony, comenta la opción `title` en la sección `edit`:

```
# apps/backend/modules/category/config/generator.yml
edit:
  #title: Editing Category "%%name%%" (#%%id%%)
```

En la aplicación `backend`, queremos utilizar el mismo formulario para modificar las categorías tanto en inglés como en francés:

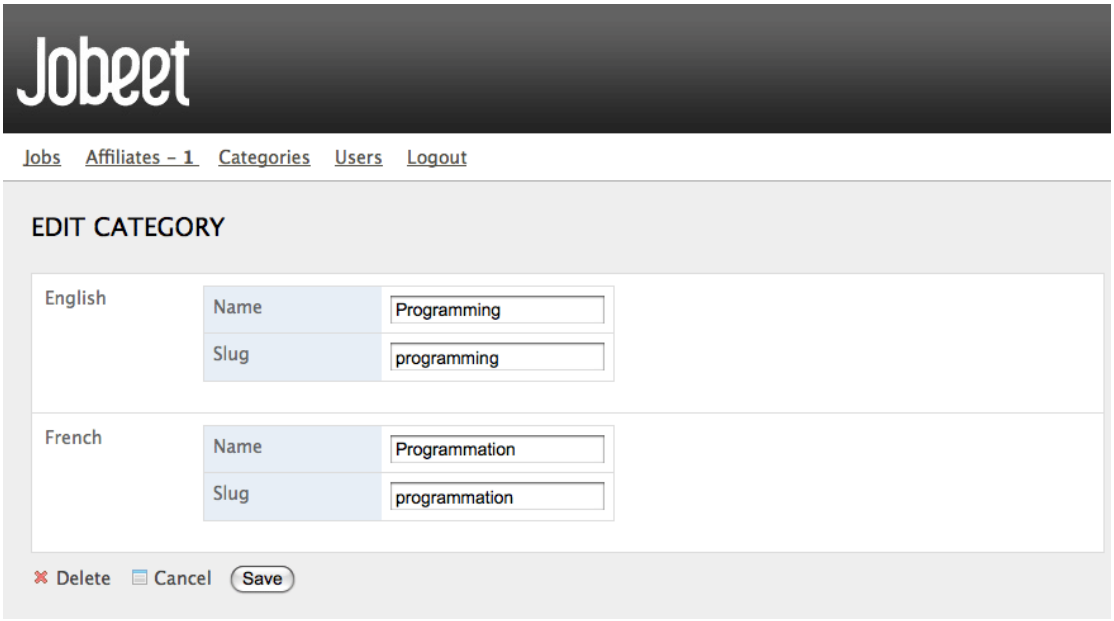

Figura 19.2. Modificando las categorías en dos idiomas a la vez

Utiliza el método `embedI18N()` para incluir un formulario internacionalizado:

```
// lib/form/JobeetCategoryForm.class.php
class JobeetCategoryForm extends BaseJobeetCategoryForm
{
  public function configure()
  {
    unset($this['jobeet_category_affiliate_list']);

    $this->embedI18n(array('en', 'fr'));
    $this->widgetSchema->setLabel('en', 'English');
    $this->widgetSchema->setLabel('fr', 'French');
  }
}
```

La interfaz del generador de la parte de administración incluye soporte para su internacionalización. Por defecto incluye las traducciones en 20 idiomas y es realmente sencillo añadir una nueva traducción o modificar una traducción existente. Copia en el directorio `i18n` de la aplicación el archivo del idioma que vas a modificar (las traducciones de la parte de administración se encuentran en `lib/vendor/symfony/lib/plugins/sfPropelPlugin/i18n/`). Como el archivo de tu aplicación se fusiona después con el de Symfony, puedes borrar todas las cadenas de texto cuya traducción no vas a modificar.

Como ya habrás visto, los archivos con las traducciones del administrador se llaman `sf_admin.fr.xml` en vez de `fr/messages.xml`. De hecho, el valor `messages` es el nombre del catálogo y puedes utilizar cualquier nombre que quieras para permitir una mejor separación entre las diferentes partes de la aplicación. No obstante, si utilizas cualquier catálogo diferente al de por defecto, tienes que indicarlo explícitamente en cada llamada al helper `__()`:

```
<?php echo __('About Jobeet', array(), 'jobeet') ?>
```

En el ejemplo anterior, Symfony busca la traducción de la cadena *"About Jobeet"* en el catálogo llamado `jobeet`.

19.5.8. Pruebas

Para completar la migración a una aplicación internacionalizada, no te olvides de arreglar las pruebas. En primer lugar, actualiza la información de las categorías en los archivos de datos copiando en el archivo `test/fixtures/010_categories.yml` los datos utilizados en las secciones anteriores. Después, vuelve a generar las clases del modelo para el entorno `test`:

```
$ php symfony propel:build-all-load --no-confirmation --env=test
```

Por último, ejecuta todas las pruebas para asegurar que no has cometido ningún error:

```
$ php symfony test:all
```

Nota

Cuando creamos la aplicación `backend` de Jobeet, no añadimos ninguna prueba funcional. Sin embargo, siempre que creas un módulo mediante la línea de comandos de Symfony se crean unas pruebas funcionales de ejemplo. Si quieres, puedes borrar todos estos archivos de prueba.

19.6. Localización

19.6.1. Plantillas

Soportar diferentes culturas también implica soportar diferentes formas de mostrar las fechas y los números. Symfony incluye numerosos métodos para que las plantillas puedan tener en consideración todas estas diferencias dependientes de la cultura del usuario:

El grupo de helpers Date (http://www.symfony-project.org/api/1_2/DateHelper) incluye los siguientes helpers:

Helper	Descripción
`format_date()`	Muestra una fecha con el formato indicado
`format_datetime()`	Muestra una fecha y hora con el formato indicado

El grupo de helpers Number (http://www.symfony-project.org/api/1_2/NumberHelper) incluye los siguientes helpers:

Helper	Descripción
`format_number()`	Muestra un número con el formato indicado
`format_currency()`	Muestra el valor de una divisa con el formato indicado

El grupo de helpers I18N (http://www.symfony-project.org/api/1_2/I18NHelper) incluye los siguientes helpers:

Helper	Descripción
`format_country()`	Muestra el nombre de un país en el idioma indicado
`format_language()`	Muestra el nombre de un idioma en el idioma indicado

19.6.2. Formularios

El framework de formularios incluye varios widgets y validadores para la información internacionalizada:

- sfWidgetFormI18nDate (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nDate)
- sfWidgetFormI18nDateTime (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nDateTime)
- sfWidgetFormI18nTime (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nTime)
- sfWidgetFormI18nSelectCountry (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nSelectCountry)
- sfWidgetFormI18nSelectCurrency (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nSelectCurrency)
- sfWidgetFormI18nSelectLanguage (http://www.symfony-project.org/api/1_2/sfWidgetFormI18nSelectLanguage)
- sfValidatorI18nChoiceCountry (http://www.symfony-project.org/api/1_2/sfValidatorI18nChoiceCountry)
- sfValidatorI18nChoiceCountry (http://www.symfony-project.org/api/1_2/sfValidatorI18nChoiceCountry)

19.7. Nos vemos mañana

Symfony incluye soporte completo para la internacionalización y la localización. De esta forma, traducir un sitio web para tus usuarios es muy sencillo porque Symfony ya incluye todas las utilidades básicas e incluso dispone de tareas de la línea de comandos para mejorar tu productividad.

El tutorial de mañana será muy especial porque vamos a mover un montón de archivos de un sitio a otro y vamos a mostrar otra forma de organizar los proyectos de Symfony.

Capítulo 20. Plugins

Ayer aprendimos a internacionalizar y localizar las aplicaciones Symfony. Una vez más, gracias al uso de estándares como ICU y la ayuda de los helpers, Symfony simplifica al máximo el proceso de internacionalización.

Hoy vamos a explicar los plugins: qué son, qué puedes incluir en un plugin y para qué se pueden utilizar.

20.1. Plugins

20.1.1. Los plugins de Symfony

Un plugin de Symfony es una forma de agrupar y distribuir un subconjunto de archivos de tu proyecto. Al igual que los proyectos, los plugins pueden contener clases, helpers, archivos de configuración, tareas, esquemas de datos e incluso archivos web como CSS y JavaScript.

20.1.2. Plugins privados

El uso más habitual de los plugins es la posibilidad de compartir código entre tus diferentes aplicaciones o incluso entre diferentes proyectos. ¿Recuerdas que las aplicaciones Symfony sólo comparten el modelo? Gracias a los plugins, las aplicaciones pueden compartir muchos otros componentes.

Si quieres reutilizar un mismo esquema de datos en diferentes proyectos o incluso un módulo entero, crea un plugin que contenga esos archivos. Como un plugin simplemente es un directorio, puedes moverlo fácilmente de un sitio a otro creando un repositorio de Subversion y empleando la propiedad `svn:externals` o simplemente copiando y pegando los archivos de un proyecto a otro.

Denominamos a estos plugins *"privados"* porque su uso se restringe a un programador o una empresa concreta, ya que no están disponibles de forma pública.

> **Sugerencia**
>
> También puedes crear paquetes para tus plugins privados y después crear tu propio canal de plugins Symfony para poder instalarlos mediante la tarea `plugin:install`.

20.1.3. Plugins públicos

Los plugins públicos son aquellos que están disponibles para que cualquier usuario de la comunidad de Symfony los pueda descargar e instalar en sus proyectos. A lo largo de este tutorial ya hemos utilizado un par de plugins públicos: `sfGuardPlugin` y `sfFormExtraPlugin`.

Aunque técnicamente son iguales que los plugins privados, la diferencia reside en que cualquiera puede instalarlos y utilizarlos en sus proyectos. Más adelante explicaremos cómo publicar un plugin público en el sitio web de Symfony.

20.1.4. Otra forma de organizar el código

Existe otra forma de utilizar los plugins muy diferente a la reutilización de código. Los plugins permiten organizar el código del proyecto de forma completamente distinta. En vez de organizar los archivos por capas (las clases del modelo en el directorio `lib/model/`, las plantillas en el directorio `templates/`, etc.) puedes organizar los archivos según su funcionalidad: guardar juntos todos los archivos relacionados con las ofertas de trabajo (modelos, módulos y plantillas), guardar juntos todos los archivos relacionados con el CMS, etc.

20.2. Estructura de archivos de los plugins

Un plugin de Symfony consiste simplemente en un conjunto de directorios que organiza los archivos según una estructura predefinida de acuerdo a la naturaleza de cada archivo. Hoy vamos a mover la mayoría del código que hemos escrito para la aplicación Jobeet a un plugin llamado `sfJobeetPlugin`. La estructura de archivos y directorios que vamos a utilizar es la siguiente:

```
sfJobeetPlugin/
  config/
    sfJobeetPluginConfiguration.class.php // Plugin initialization
    schema.yml                            // Database schema
    routing.yml                           // Routing
  lib/
    Jobeet.class.php                      // Classes
    helper/                               // Helpers
    filter/                               // Filter classes
    form/                                 // Form classes
    model/                                // Model classes
    task/                                 // Tasks
  modules/
    job/                                  // Modules
      actions/
      config/
      templates/
  web/                                    // Assets like JS, CSS, and images
```

20.3. El plugin Jobeet

Inicializar un plugin es tan sencillo como crear un nuevo directorio bajo el directorio `plugins/`. Para el plugin de Jobeet, crea un directorio llamado `sfJobeetPlugin`:

```
$ mkdir plugins/sfJobeetPlugin
```

Nota

El nombre de todos los plugins debe acabar con la palabra `Plugin`. También es recomendable utilizar el prefijo `sf`, aunque no es obligatorio.

20.3.1. El modelo

En primer lugar, mueve el archivo `config/schema.yml` a `plugins/sfJobeetPlugin/config/`:

```
$ mkdir plugins/sfJobeetPlugin/config/
$ mv config/schema.yml plugins/sfJobeetPlugin/config/schema.yml
```

Nota

Todos los comandos que mostramos en este tutorial son los apropiados para los entornos tipo Unix. Si utilizas Windows, puedes copiar y pegar los archivos utilizando el explorador de archivos. Si utilizas Subversion o cualquier otra herramienta para gestionar tu código, utiliza las herramientas que incluyen para mover código (como por ejemplo `svn mv` para mover los archivos).

A continuación, mueve todos los archivos del modelo, formularios y filtros al directorio `plugins/sfJobeetPlugin/lib/`:

```
$ mkdir plugins/sfJobeetPlugin/lib/
$ mv lib/model/ plugins/sfJobeetPlugin/lib/
$ mv lib/form/ plugins/sfJobeetPlugin/lib/
$ mv lib/filter/ plugins/sfJobeetPlugin/lib/
```

Si ahora ejecutas la tarea `propel:build-model`, Symfony sigue generando todos sus archivos en el directorio `lib/model/`, que es justo lo que no queremos. El directorio en el que Propel genera sus archivos se puede configurar mediante la opción `package`. Abre el archivo `schema.yml` y añade la siguiente configuración:

```
# plugins/sfJobeetPlugin/config/schema.yml
propel:
  _attributes:      { package: plugins.sfJobeetPlugin.lib.model }
```

Ahora Symfony genera sus archivos en el directorio `plugins/sfJobeetPlugin/lib/model/`. Los generadores de formularios y de filtros también tienen en consideración esta configuración cuando generan sus archivos.

La tarea `propel:build-sql` genera un archivo SQL para crear las tablas de la base de datos. Como el archivo se llama igual que el paquete, elimina el archivo actual:

```
$ rm data/sql/lib.model.schema.sql
```

Si ejecutas ahora la tarea `propel:build-all-load`, Symfony genera todos sus archivos en el directorio `lib/model/` del plugin:

```
$ php symfony propel:build-all-load --no-confirmation
```

Después de ejecutar la tarea anterior, asegúrate de que no se ha creado un directorio llamado `lib/model/`. Sin embargo, la tarea anterior si que ha creado los directorios `lib/`

`form/` y `lib/filter/`. Estos directorios incluyen las clases base de todos los formularios Propel del proyecto.

Como estos archivos son globales para un proyecto, puedes eliminarlos en el plugin:

```
$ rm plugins/sfJobeetPlugin/lib/form/BaseFormPropel.class.php
$ rm plugins/sfJobeetPlugin/lib/filter/BaseFormFilterPropel.class.php
```

Nota

Si utilizas Symfony 1.2.0 o 1.2.1, el archivo del formulario base de los filtros se encuentra en el directorio `plugins/sfJobeetPlugin/lib/filter/base/`.

También puedes mover el archivo `Jobeet.class.php` al plugin:

```
$ mv lib/Jobeet.class.php plugins/sfJobeetPlugin/lib/
```

Como hemos movido muchos archivos y clases, no te olvides de borrar la cache de Symfony:

```
$ php symfony cc
```

Sugerencia

Si utilizas un acelerador de PHP tipo APC, es posible que se produzcan algunos errores en este punto, por lo que te recomendamos que reinicies Apache.

Después de mover todos los archivos del modelo al plugin, ejecuta las pruebas automáticas para comprobar que todo sigue funcionando correctamente:

```
$ php symfony test:all
```

20.3.2. Los controladores y las vistas

El siguiente paso lógico consiste en mover los módulos al directorio del plugin. Para evitar duplicidades con el nombre de los módulos, te aconsejamos prefijar el nombre de cada módulo con el nombre del propio plugin:

```
$ mkdir plugins/sfJobeetPlugin/modules/
$ mv apps/frontend/modules/affiliate plugins/sfJobeetPlugin/modules/
sfJobeetAffiliate
$ mv apps/frontend/modules/api plugins/sfJobeetPlugin/modules/sfJobeetApi
$ mv apps/frontend/modules/category plugins/sfJobeetPlugin/modules/
sfJobeetCategory
$ mv apps/frontend/modules/job plugins/sfJobeetPlugin/modules/sfJobeetJob
$ mv apps/frontend/modules/language plugins/sfJobeetPlugin/modules/
sfJobeetLanguage
```

No te olvides de modificar también el nombre de la clase en todos los archivos `actions.class.php` y `components.class.php` de cada módulo (por ejemplo, la clase `affiliateActions` se debe renombrar a `sfJobeetAffiliateActions`).

Cambia también las llamadas a `include_partial()` y `include_component()` en las siguientes plantillas:

- `sfJobeetAffiliate/templates/_form.php` (cambia `affiliate` por `sfJobeetAffiliate`)
- `sfJobeetCategory/templates/showSuccess.atom.php`
- `sfJobeetCategory/templates/showSuccess.php`
- `sfJobeetJob/templates/indexSuccess.atom.php`
- `sfJobeetJob/templates/indexSuccess.php`
- `sfJobeetJob/templates/searchSuccess.php`
- `sfJobeetJob/templates/showSuccess.php`
- `apps/frontend/templates/layout.php`

Actualiza las acciones `search` y `delete`:

```
// plugins/sfJobeetPlugin/modules/sfJobeetJob/actions/actions.class.php
class sfJobeetJobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    if (!$query = $request->getParameter('query'))
    {
      return $this->forward('sfJobeetJob', 'index');
    }

    $this->jobs = JobeetJobPeer::getForLuceneQuery($query);

    if ($request->isXmlHttpRequest())
    {
      if ('*' == $query || !$this->jobs)
      {
        return $this->renderText('No results.');
      }
      else
      {
        return $this->renderPartial('sfJobeetJob/list', array('jobs' =>
$this->jobs));
      }
    }
  }

  public function executeDelete(sfWebRequest $request)
  {
    $request->checkCSRFProtection();

    $jobeet_job = $this->getRoute()->getObject();
    $jobeet_job->delete();

    $this->redirect('sfJobeetJob/index');
  }
```

```
  // ...
}
```

Por último, modifica el archivo `routing.yml` para que tenga en cuenta todos los cambios anteriores:

```
# apps/frontend/config/routing.yml
affiliate:
  class:   sfPropelRouteCollection
  options:
    model:          JobeetAffiliate
    actions:        [new, create]
    object_actions: { wait: GET }
    prefix_path:    /:sf_culture/affiliate
    module:         sfJobeetAffiliate
  requirements:
    sf_culture: (?:fr|en)

api_jobs:
  url:     /api/:token/jobs.:sf_format
  class:   sfPropelRoute
  param:   { module: sfJobeetApi, action: list }
  options: { model: JobeetJob, type: list, method: getForToken }
  requirements:
    sf_format: (?:xml|json|yaml)

category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfPropelRoute
  param:   { module: sfJobeetCategory, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object, method: doSelectForSlug }
  requirements:
    sf_format: (?:html|atom)
    sf_culture: (?:fr|en)

job_search:
  url:   /:sf_culture/search
  param: { module: sfJobeetJob, action: search }
  requirements:
    sf_culture: (?:fr|en)

job:
  class:   sfPropelRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: PUT, extend: PUT }
    prefix_path:    /:sf_culture/job
    module:         sfJobeetJob
  requirements:
    token: \w+
    sf_culture: (?:fr|en)

job_show_user:
  url:     /:sf_culture/job/:company_slug/:location_slug/:id/:position_slug
```

```
  class:   sfPropelRoute
  options:
    model: JobeetJob
    type:  object
    method_for_criteria: doSelectActive
  param:   { module: sfJobeetJob, action: show }
  requirements:
    id:        \d+
    sf_method: GET
    sf_culture: (?:fr|en)

change_language:
  url:   /change_language
  param: { module: sfJobeetLanguage, action: changeLanguage }

localized_homepage:
  url:   /:sf_culture/
  param: { module: sfJobeetJob, action: index }
  requirements:
    sf_culture: (?:fr|en)

homepage:
  url:   /
  param: { module: sfJobeetJob, action: index }
```

Si ahora accedes al sitio web de Jobeet, verás que se muestran excepciones indicando que los módulos no están activados. Como los plugins están disponibles en todas las aplicaciones de un mismo proyecto, debes indicar explícitamente en el archivo de configuración `settings.yml` los módulos que están activados en cada aplicación:

```
# apps/frontend/config/settings.yml
all:
  .settings:
    enabled_modules:
      - default
      - sfJobeetAffiliate
      - sfJobeetApi
      - sfJobeetCategory
      - sfJobeetJob
      - sfJobeetLanguage
```

El último paso de la migración consiste en arreglar las pruebas funcionales en las que probamos el nombre del módulo.

Activando los plugins

Para que un plugin esté disponible en el proyecto, debes activarlo en la clase `ProjectConfiguration`. Esta activación no es necesaria con la configuración por defecto, ya que Symfony emplea la estrategia de la *lista negra*, que activa todos los plugins salvo los que se indican explícitamente:

```
// config/ProjectConfiguration.class.php
public function setup()
{
```

```
  $this->enableAllPluginsExcept(array('sfDoctrinePlugin',
'sfCompat10Plugin'));
}
```

Esta estrategia se utiliza para mantener la compatibilidad con las versiones anteriores de Symfony, pero te aconsejamos que utilices la estrategia de la *lista blanca*, donde se activan explícitamente los plugins con el método `enablePlugins()`:

```
// config/ProjectConfiguration.class.php
public function setup()
{
  $this->enablePlugins(array('sfPropelPlugin', 'sfGuardPlugin',
'sfFormExtraPlugin', 'sfJobeetPlugin'));
}
```

20.3.3. Las tareas

Mover las tareas al plugin es muy sencillo:

```
$ mv lib/task plugins/sfJobeetPlugin/lib/
```

20.3.4. Los archivos de internacionalización

Los plugins también pueden contener archivos en formato XLIFF:

```
$ mv apps/frontend/i18n plugins/sfJobeetPlugin/
```

20.3.5. El sistema de enrutamiento

Los plugins también pueden incluir sus propias reglas en el sistema de enrutamiento:

```
$ mv apps/frontend/config/routing.yml plugins/sfJobeetPlugin/config/
```

20.3.6. Los archivos CSS y JavaScript

A pesar de que puede no parecer evidente, los plugins también pueden contener archivos web como imágenes, hojas de estilos y archivos JavaScript. Como no vamos a redistribuir Jobeet como plugin, no tiene sentido que añadamos todos estos archivos, pero si quieres hacerlo, crea un directorio llamado `plugins/sfJobeetPlugin/web/` y copia en el todos estos archivos.

Para que los archivos web del plugin se puedan ver desde el navegador, es necesario hacerlos accesibles en el directorio `web/` del proyecto. La tarea `plugin:publish-assets` se encarga de ello creando enlaces simbólicos en sistemas operativos Unix y copiando los archivos en sistemas operativos Windows:

```
$ php symfony plugin:publish-assets
```

20.3.7. El usuario

Mover los métodos de la clase `myUser` que se encargan de crear el historial de las ofertas de trabajo visitadas es un poco más complicado. Se podría crear una clase llamada `JobeetUser` y hacer que `myUser` herede de ella. No obstante, existe una forma mejor de hacerlo, sobre todo si varios plugins diferentes quieren añadir métodos a la clase.

Los objetos internos de Symfony notifican durante su tiempo de vida diferentes eventos que podemos *escuchar*. En nuestro caso, queremos *escuchar* el evento `user.method_not_found`, que se notifica cuando se invoca un método que no existe en el objeto `sfUser`.

Cuando se inicializa Symfony, también se inicializan todos los plugins que tienen una clase de configuración:

```
// plugins/sfJobeetPlugin/config/sfJobeetPluginConfiguration.class.php
class sfJobeetPluginConfiguration extends sfPluginConfiguration
{
  public function initialize()
  {
    $this->dispatcher->connect('user.method_not_found', array('JobeetUser',
'methodNotFound'));
  }
}
```

Las notificaciones de los eventos se gestionan mediante el objeto sfEventDispatcher (http://www.symfony-project.org/api/1_2/sfEventDispatcher) . Registrar un *listener* (es decir, un método que *escucha* eventos) es tan sencillo como realizar una llamada al método `connect()`. El método `connect()` asocia un nombre de evento con un elemento ejecutable de PHP, también llamado *"PHP callable"*.

Nota

Un elemento ejecutable de PHP (http://www.php.net/manual/es/function.is-callable.php) es una variable de PHP que se puede utilizar en la función `call_user_func()` y que devuelve `true` cuando se pasa a la función `is_callable()`. Si el elemento ejecutable es una función, se indica mediante una cadena de texto. Si el elemento ejecutable es el método de una clase u objeto, se indica mediante un array.

El código del ejemplo anterior hace que el objeto `myUser` invoque el método estático `methodNotFound()` de la clase `JobeetUser` cada vez que no se encuentre un método en ese objeto. Después, el método `methodNotFound()` se encarga de procesar o ignorar el método que no existe en `myUser`.

Elimina todos los métodos de la clase `myUser` y crea en su lugar la clase `JobeetUser`:

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
}
// plugins/sfJobeetPlugin/lib/JobeetUser.class.php
class JobeetUser
```

```
{
  static public function methodNotFound(sfEvent $event)
  {
    if (method_exists('JobeetUser', $event['method']))
    {
      $event->setReturnValue(call_user_func_array(
        array('JobeetUser', $event['method']),
        array_merge(array($event->getSubject()), $event['arguments'])
      ));

      return true;
    }
  }

  static public function isFirstRequest(sfUser $user, $boolean = null)
  {
    if (is_null($boolean))
    {
      return $user->getAttribute('first_request', true);
    }
    else
    {
      $user->setAttribute('first_request', $boolean);
    }
  }

  static public function addJobToHistory(sfUser $user, JobeetJob $job)
  {
    $ids = $user->getAttribute('job_history', array());

    if (!in_array($job->getId(), $ids))
    {
      array_unshift($ids, $job->getId());
      $user->setAttribute('job_history', array_slice($ids, 0, 3));
    }
  }

  static public function getJobHistory(sfUser $user)
  {
    return JobeetJobPeer::retrieveByPks($user->getAttribute('job_history',
array()));
  }

  static public function resetJobHistory(sfUser $user)
  {
    $user->getAttributeHolder()->remove('job_history');
  }
}
```

Cuando se invoca el método `methodNotFound()`, el encargado de notificar los eventos pasa como argumento un objeto de tipo sfEvent (http://www.symfony-project.org/api/1_2/sfEvent) .

Si el método existe en la clase `JobeetUser`, se invoca y el valor devuelto se devuelve al notificador de eventos. Si no existe el método, Symfony utiliza el siguiente *listener* registrado para ese evento y si ya no existen más *listeners*, se lanza una excepción.

El método `getSubject()` se puede utilizar para determinar el notificador del evento, que en este caso sería el objeto `myUser`.

Como siempre que creas nuevas clases, no te olvides de borrar la cache de Symfony antes de probar la aplicación o antes de ejecutar las pruebas:

```
$ php symfony cc
```

20.3.8. Arquitectura por defecto vs. arquitectura de los plugins

Si utilizas la arquitectura de los plugins, puedes organizar tu código de una forma completamente diferente:

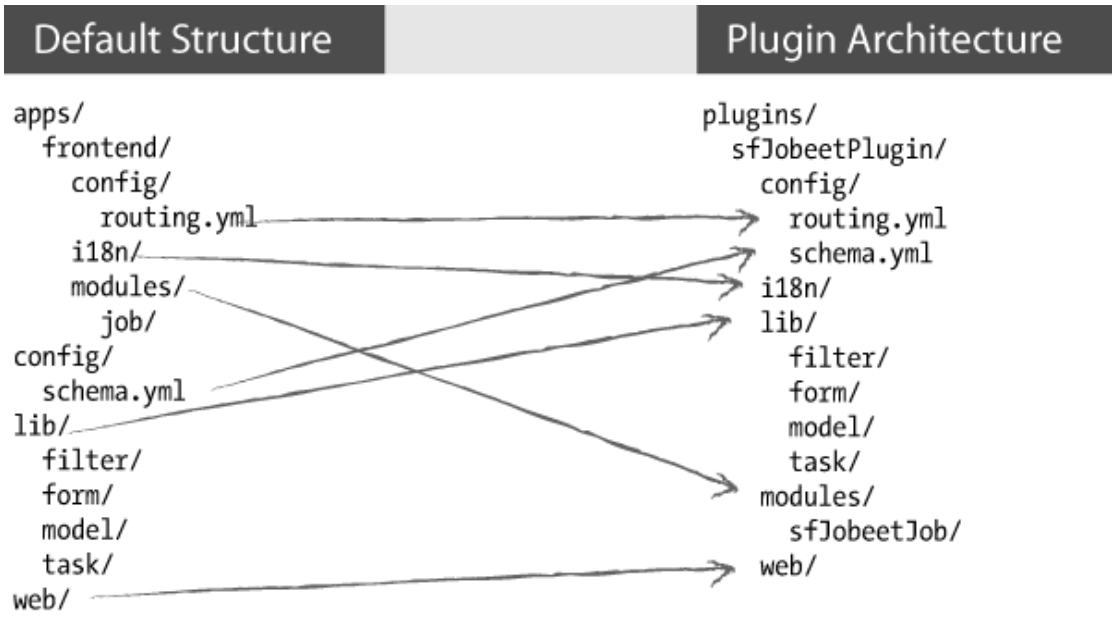

Figura 20.1. Diferencias entre la arquitectura tradicional y la arquitectura de los plugins

20.4. Utilizando los plugins

Siempre que implementas una nueva característica en tu aplicación y siempre que tratas de resolver un problema clásico de las aplicaciones web, lo más seguro es que otra persona ya haya resuelto antes ese problema y quizás hasta haya publicado un plugin Symfony con la solución. Si quieres buscar plugins públicos de Symfony, lo mejor es que accedas a la sección de plugins (http://www.symfony-project.org/plugins/) del sitio web oficial de Symfony.

Como los plugins no son más que una estructura de directorios, existen varias formas de instalarlos:

- Utilizar la tarea `plugin:install`, que sólo funciona si el desarrollador del plugin ha creado un paquete con sus contenidos y lo ha subido al sitio web de Symfony.

- Descargar el paquete a mano y descomprimirlo en el directorio `plugins/` de tu proyecto, por lo que también es necesario que el desarrollador del plugin haya creado y subido el paquete.
- Crear un nuevo `svn:externals` en el directorio `plugins/` para el plugin que se quiere descargar, que sólo funciona si el desarrollador del plugin publica el plugin en un repositorio público de Subversion.

Las dos últimas formas de instalar un plugin son muy sencillas pero poco flexibles. La primera forma se encarga de instalar la versión más reciente del plugin disponible para la versión de Symfony que utilizas, permite actualizar fácilmente los plugins y permite gestionar de forma sencilla las dependencias entre plugins.

20.5. Publicando tu plugin

20.5.1. Creando el paquete del plugin

Si quieres crear el paquete del plugin, debes añadir algunos archivos obligatorios a la estructura de directorios del plugin. En primer lugar, crea un archivo llamado `README` en el directorio raíz del plugin que contenga las intrucciones de instalación del plugin y que explique lo que proporciona y lo que no. Este archivo `README` debe estar escrito en el formato Markdown (http://daringfireball.net/projects/markdown/syntax) . Además, este archivo es el que utiliza el sitio web de Symfony para mostrar la información y documentación del plugin. Si quieres probar cómo se transforma tu archivo `README` al formato HTML, puedes utilizar la herramienta Symfony plugin dingus (http://www.symfony-project.org/plugins/markdown_dingus) .

Tareas para crear plugins

Si creas muchos plugins públicos o privados, quizás te interese utilizar algunas de las tareas del plugin sfTaskExtraPlugin (http://www.symfony-project.com/plugins/sfTaskExtraPlugin) . Este plugin lo mantienen los propios creadores de Symfony e incluye varias tareas que facilitan la creación de plugins, como por ejemplo:

- `generate:plugin`
- `plugin:package`

Además del archivo `README`, también debes crear un archivo llamado `LICENSE`. Elegir la licencia adecuada para tu plugin no es algo sencillo, pero la sección de plugins del sitio web de Symfony sólo muestra los plugins que se publican con una licencia similar a la del propio framework (MIT, BSD, LGPL y PHP). El contenido del archivo `LICENSE` se muestra en la pestaña *"license"* de la página del plugin.

El último archivo obligatorio se llama `package.xml` y debe estar en el directorio raíz del plugin. Este archivo `package.xml` se debe crear siguiendo la sintaxis de los paquetes PEAR (http://pear.php.net/manual/en/guide-developers.php) .

Nota

La mejor forma de aprender la sintaxis del archivo `package.xml` consiste en copiar el archivo de cualquier otro plugin, como por ejemplo el archivo package.xml de sfGuardPlugin (http://svn.symfony-project.com/plugins/sfGuardPlugin/branches/1.2/package.xml) .

La siguiente plantilla de ejemplo muestra las diferentes partes que componen el archivo `package.xml`:

```
<!-- plugins/sfJobeetPlugin/package.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.1" 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>sfJobeetPlugin</name>
  <channel>plugins.symfony-project.org</channel>
  <summary>A job board plugin.</summary>
  <description>A job board plugin.</description>
  <lead>
    <name>Fabien POTENCIER</name>
    <user>fabpot</user>
    <email>fabien.potencier@symfony-project.com</email>
    <active>yes</active>
  </lead>
  <date>2008-12-20</date>
  <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.com/license">
    MIT license
  </license>
  <notes />

  <contents>
    <!-- CONTENT -->
  </contents>

  <dependencies>
   <!-- DEPENDENCIES -->
  </dependencies>

  <phprelease>
</phprelease>

<changelog>
```

```
  <!-- CHANGELOG -->
</changelog>
</package>
```

La etiqueta `<content>` especifica los archivos que contiene el paquete:

```
<contents>
  <dir name="/">
    <file role="data" name="README" />
    <file role="data" name="LICENSE" />

    <dir name="config">
      <file role="data" name="config.php" />
      <file role="data" name="schema.yml" />
    </dir>

    <!-- ... -->
  </dir>
</contents>
```

La etiqueta `<dependencies>` define todas las dependencias que tiene el plugin respecto a PHP, Symfony y/o el resto de plugins. Esta información es la que utiliza la tarea `plugin:install` para instalar la versión del plugin que mejor se adapta al entorno de trabajo y también para instalar todas las dependencias existentes con otros plugins.

```
<dependencies>
  <required>
    <php>
      <min>5.0.0</min>
    </php>
    <pearinstaller>
      <min>1.4.1</min>
    </pearinstaller>
    <package>
      <name>symfony</name>
      <channel>pear.symfony-project.com</channel>
      <min>1.2.0</min>
      <max>1.3.0</max>
      <exclude>1.3.0</exclude>
    </package>
  </required>
</dependencies>
```

Como se muestra en el ejemplo anterior, siempre deberías establecer la dependencia de tu plugin con Symfony. Al declarar la versión mínima y máxima de Symfony con las que el plugin es compatible, la tarea `plugin:install` puede determinar la versión de Symfony necesaria, ya que cada versión de Symfony contiene diferencias en su API.

También puedes declarar dependencias con otros plugins:

```
<package>
  <name>sfFooPlugin</name>
  <channel>plugins.symfony-project.org</channel>
  <min>1.0.0</min>
  <max>1.2.0</max>
```

```
  <exclude>1.2.0</exclude>
</package>
```

La etiqueta `<changelog>` es opcional, pero proporciona información útil sobre los cambios realizados por cada versión del plugin. Esta información se muestra en la pestaña *"changelog"* del plugin y también está disponible en el canal RSS de los plugins de Symmfony (http://www.symfony-project.org/plugins/recently.rss) .

```
<changelog>
  <release>
    <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.com/license">
      MIT license
    </license>
    <date>2008-12-20</date>
    <license>MIT</license>
    <notes>
       * fabien: First release of the plugin
    </notes>
  </release>
</changelog>
```

20.5.2. Publicar un plugin en el sitio web de Symfony

Si has creado un plugin útil y quieres compartirlo con la comunidad de usuarios de Symfony, puedes crear una cuenta de usuario (http://www.symfony-project.org/user/new) en el sitio web de Symfony y después crear tu plugin (http://www.symfony-project.org/plugins/new) .

Una vez creado, te conviertes automáticamente en el administrador del plugin y por tanto, verás una pestaña llamada *"admin"* en la página del plugin. Desde esta pestaña puedes gestionar toda la información del plugin y puedes subir los paquetes de las nuevas versiones.

Nota

La página plugin FAQ (http://www.symfony-project.org/plugins/FAQ) contiene mucha más información útil para los desarrolladores de plugins.

20.6. Nos vemos mañana

Crear plugins y compartirlos con la comunidad de usuarios de Symfony es una de las mejores formas de devolver parte de lo que te da el proyecto Symfony. Crear plugins es tan sencillo que el repositorio de Symfony está lleno de plugins, muchos de ellos últiles, algunos divertidos y otros hasta un poco ridículos.

Capítulo 21. El día del diseño

El tutorial Jobeet original se publicó durante los primeros 24 días del mes de diciembre de 2008. Durante el día 21 se celebró un concurso de diseño y se eligió mediante votación popular el diseño gráfico definitivo de la aplicación Jobeet.

El diseño ganador fue obra de la empresa centre{source} (http://www.centresource.com/) , y ese es el diseño que te descargaste durante el tutorial del día 4.

Nota

centre{source} (http://www.centresource.com/) es una empresa interactiva que proporciona todos los servicios necesarios para las empresas que consideran a la web como uno de sus activos estratégicos. Proporcionan a sus clientes cuatro servicios esenciales: estrategia, planificación, ejecución y gestión continua.

Capítulo 22. La cache

Hoy hablaremos sobre la cache. El framework Symfony dispone de varias estrategias relacionadas con la cache. Los archivos de configuración YAML por ejemplo se convierten a código PHP y después se guardan en la cache. También hemos visto en los tutoriales de los días anteriores que los módulos creados por el generador de la parte de administración se guardan en la cache para mejorar su rendimiento.

Hoy vamos a hablar de otra cache: la cache de HTML. Para mejorar el rendimiento de tu sitio web puedes guardar en la cache todo el contenido HTML de las páginas o solamente ciertas partes de las páginas.

22.1. Creando un nuevo entorno

La cache de las plantillas de Symfony se encuentra activada por defecto en el archivo de configuración `settings.yml` sólo para el entorno de ejecución `prod` y no para los entornos `test` y `dev`:

```
prod:
  .settings:
    cache: on

dev:
  .settings:
    cache: off

test:
  .settings:
    cache: off
```

Como tenemos que probar la cache antes de subir la aplicación a producción, podemos activar la cache para el entorno `dev` o podemos crear un nuevo entorno. Recuerda que un entorno se define mediante su nombre (una simple cadena de texto), un controlador frontal asociado y opcionalmente, varias opciones de configuración específicas.

Para poder jugar con la cache de la aplicación Jobeet vamos a crear un nuevo entorno llamado `cache` muy similar al entorno `prod`, pero con los mensajes de log y la información de depuración activadas como en el entorno `dev`.

Para crear el controlador frontal del entorno `cache` vamos a copiar el archivo `web/frontend_dev.php` correspondiente al controlador frontal del entorno `dev` al archivo `web/frontend_cache.php`:

```
// web/frontend_cache.php
if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1')))
{
  die('You are not allowed to access this file. Check '.basename(__FILE__).'
for more information.');
}
```

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

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

El código anterior es todo lo que necesitas para crear el nuevo controlador frontal. A partir de este momento, ya puedes hacer uso del nuevo entorno `cache`. La única diferencia con el controlador frontal de desarrollo es que el segundo argumento del método `getApplicationConfiguration()` es `cache`, ya que este argumento indica el nombre del entorno.

Accede al controlador frontal de `cache` para probar este nuevo entorno en el navegador:

```
http://jobeet.localhost/frontend_cache.php/
```

Nota

El script del controlador frontal comienza con un pequeño código que asegura que este controlador sólo se accede desde una dirección IP local. Esta medida de seguridad permite proteger el acceso al controlador frontal de los servidores de producción. En el tutorial de mañana hablaremos más en detalle sobre este asunto.

Por el momento, el entorno `cache` hereda todas sus opciones de la configuración por defecto. Modifica el archivo de configuración `settings.yml` para añadir opciones específicas para el entorno `cache`:

```
# apps/frontend/config/settings.yml
cache:
  .settings:
    error_reporting: <?php echo (E_ALL | E_STRICT)."\n" ?>
    web_debug:       on
    cache:           on
    etag:            off
```

La opción de configuración `cache` activa la cache de las plantillas Symfony, mientras que la opción `web_debug` activa la barra de depuración web.

Como también nos interesa guardar las sentencias SQL en los archivos de log, debemos modificar la configuración de la base de datos. Modifica el archivo `databases.yml` y añade la siguiente configuración al principio del archivo:

```
# config/databases.yml
cache:
  propel:
    class: sfPropelDatabase
    param:
      classname: DebugPDO
```

Para que los cambios sean efectivos, no te olvides de borrar la cache de Symfony, ya que todos los archivos de configuración se guardan en la cache:

```
$ php symfony cc
```

Si refrescas la página en tu navegador, ahora deberías ver la barra de depuración web en la esquina superior derecha de la página, tal y como aparece en el entorno `dev`.

22.2. Configurando la cache

La cache de las plantillas de Symfony se configura en el archivo `cache.yml`. La configuración por defecto de la aplicación se encuentra en `apps/frontend/config/cache.yml`:

```
default:
  enabled:     off
  with_layout: false
  lifetime:    86400
```

Como todas las páginas de la aplicación pueden contener información dinámica, por defecto la cache se deshabilita de forma global (`enabled: off`). No vamos a cambiar esta opción porque vamos a activar la cache página a página.

La opción `lifetime` establece el tiempo de vida en segundos de la cache en el servidor (`86400` equivale a un día completo).

Sugerencia

Si quieres también puedes utilizar la estrategia opuesta: habilitar de forma global la cache y deshabilitarla para todas las páginas que no se deben guardar en la cache. La decisión sobre la estrategia a utilizar depende exclusivamente de la que te suponga menos trabajo.

22.3. Guardando páginas en la cache

Como la portada de Jobeet será la página más visitada de todo el sitio, no vamos a obtener los datos de la base de datos cada vez que un usuario visita la página, sino que la vamos a guardar en la cache.

Crea un archivo llamado `cache.yml` para el módulo `sfJobeetJob`:

```
# plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml
index:
  enabled:     on
  with_layout: true
```

Sugerencia

El archivo de configuración `cache.yml` tiene las mismas propiedades que cualquier otro archivo de configuración de Symfony como por ejemplo `view.yml`. Por tanto, puedes activar la cache para todas las acciones de un módulo utilizando el valor especial `all`.

Si recargas la página en el navegador, verás que Symfony ha añadido una caja en la esquina superior izquierda de la página indicando que su contenido se ha guardado en la cache:

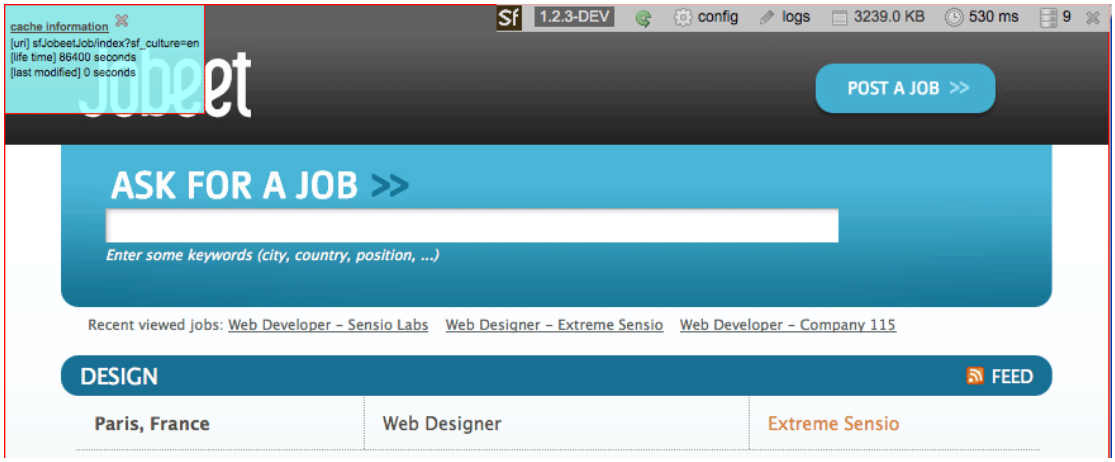

Figura 22.1. Caja que indica que el contenido se ha guardado en la cache

La caja incluye información muy útil para depurar el funcionamiento de la cache, como por ejemplo su tiempo de vida total y su tiempo de vida actual.

Si vuelves a refrescar la página, verás que la caja de la cache ahora se muestra de color amarillo, lo que indica que la página se ha obtenido directamente de la cache:

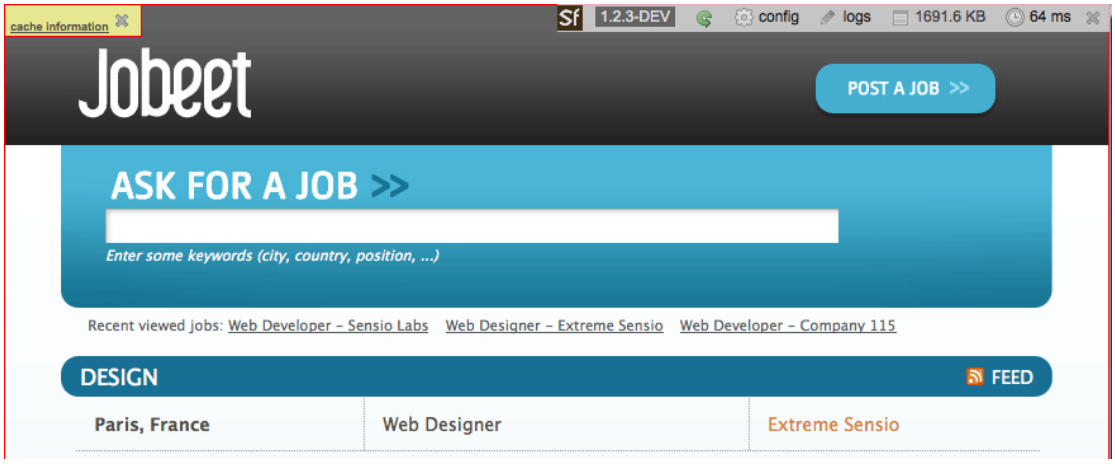

Figura 22.2. Caja que indica que el contenido se ha obtenido de la cache

Si te fijas bien en este segundo caso, verás que la barra de depuración web muestra que no se ha realizado ninguna consulta a la base de datos.

> **Sugerencia**
>
> Aunque cada usuario puede cambiar el idioma de la página, la cache sigue funcionando porque el propio idioma de la página se incluye como parte de la URL.

Cuando una página se puede guardar en la cache, Symfony comprueba si ya existía en la cache. En el caso de que no exista, Symfony almacena en la cache el objeto de la respuesta después de enviar la respuesta al usuario. En las siguientes peticiones la respuesta ya se encuentra en la cache, por lo que Symfony envía directamente la respuesta sin ni siquiera llamar a la parte del controlador:

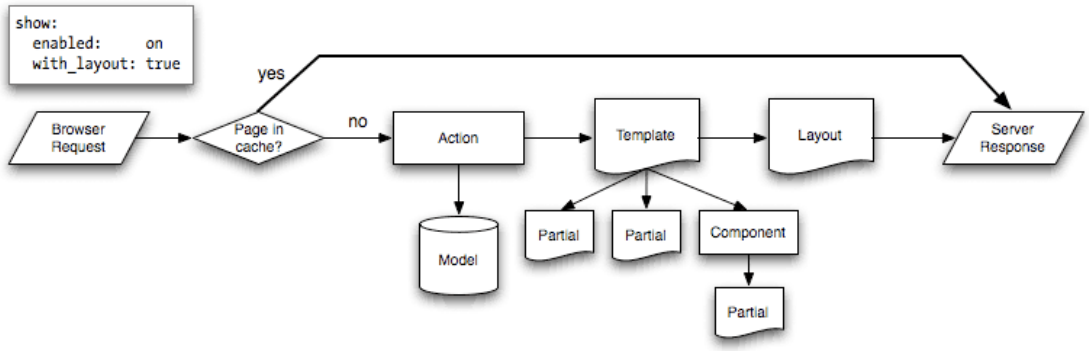

Figura 22.3. Flujo de trabajo al guardar una página en la cache

Este pequeño cambio tiene un impacto enorme en el rendimiento del sitio web, tal y como puedes comprobar tu mismo con herramientas como JMeter (http://jakarta.apache.org/jmeter/) .

Nota

Si la petición del usuario contiene parámetros `GET` o se envía con los métodos `POST`, `PUT` o `DELETE`, Symfony nunca la guarda en la cache, independientemente de la configuración de la página.

El formulario de publicación de una nueva oferta de trabajo también se puede guardar en la cache:

```
# plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml
new:
  enabled:     on

index:
  enabled:     on

all:
  with_layout: true
```

Como las dos páginas se pueden guardar enteras en la cache (incluso con el layout) hemos creado una sección especial de tipo `all` para establecer la configuración por defecto de todas las acciones del módulo `sfJobeetJob`.

22.4. Borrando la cache

Si quieres borrar la cache de páginas, puedes utilizar la tarea `cache:clear`:

```
$ php symfony cc
```

La tarea `cache:clear` borra todos los contenidos que Symfony guarda en la cache del directorio `cache/`. Esta tarea también admite opciones que le indican las partes concretas de la cache que se quieren borrar. Si sólo quieres borrar la cache de las plantillas del entorno `cache`, puedes emplear las opciones `--type` y `--env`:

```
$ php symfony cc --type=template --env=cache
```

Si no quieres borrar la cache cada vez que haces un cambio, puedes deshabilitar la cache añadiendo cualquier variable de tipo `GET` en la URL o puedes pulsar sobre el botón *"Ignore cache"* de la barra de depuración web:

Figura 22.4. Barra de depuración web con el icono para ignorar la cache

22.5. Guardando acciones en la cache

En ocasiones no es posible guardar la página entera en la cache, pero puedes guardar la plantilla asociada a la acción. En otras palabras, puedes guardar en la cache todos los contenidos salvo el layout.

En la aplicación Jobeet no podemos guardar en la cache la página entera debido a la barra del historial de ofertas de trabajo visitadas. Por tanto, modifica la configuración de la cache del módulo `job`:

```
# plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml
new:
  enabled:     on

index:
  enabled:     on

all:
  with_layout: false
```

Al establecer la opción `with_layout` a `false`, impedimos que el layout se guarde en la cache. No olvides borrar la cache para que los cambios tengan efecto:

```
$ php symfony cc
```

Para ver el resultado de la nueva configuración, recarga la página en el navegador:

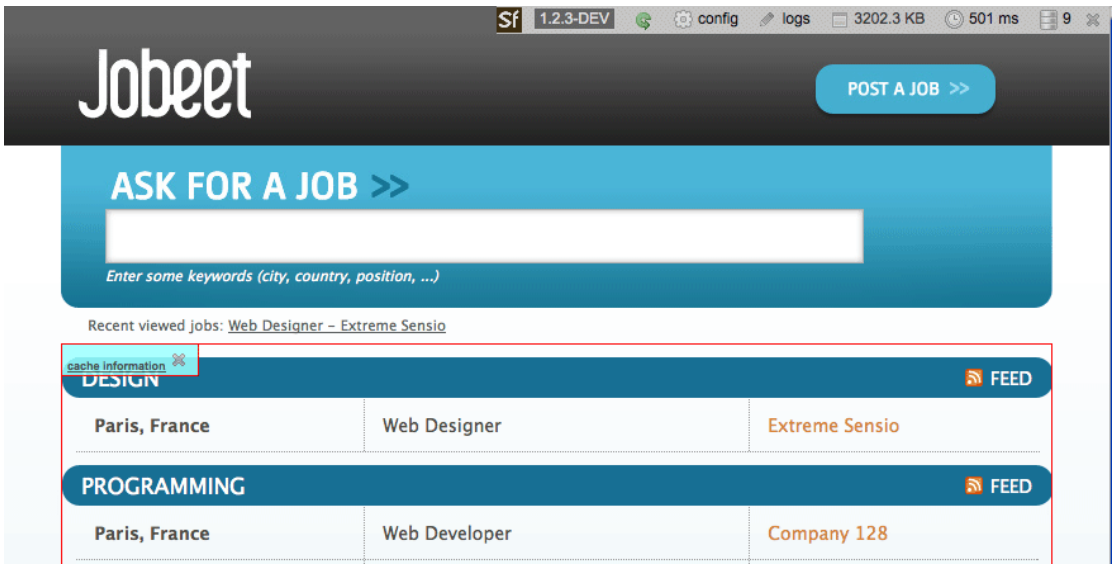

Figura 22.5. Resultado de guardar la plantilla en la cache

Aunque el flujo de la petición es similar al del caso anterior, guardar en la cache una página sin layout requiere de muchos más recursos.

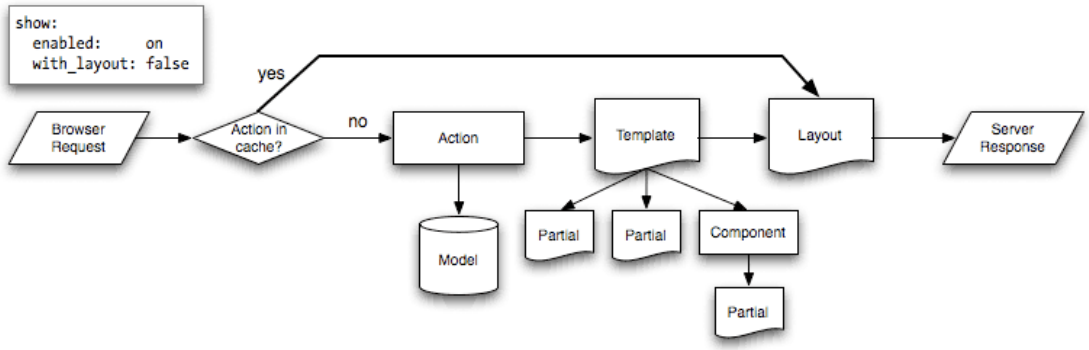

Figura 22.6. Flujo de trabajo al guardar una página sin layout en la cache

22.6. Guardando elementos parciales y componentes en la cache

Si creas sitios web muy dinámicos, es posible que no puedas guardar en la cache la plantilla completa. En estos casos, debes configurar la cache con mucho más detalle. Afortunadamente, Symfony también permite guardar en la cache los elementos parciales y los componentes.

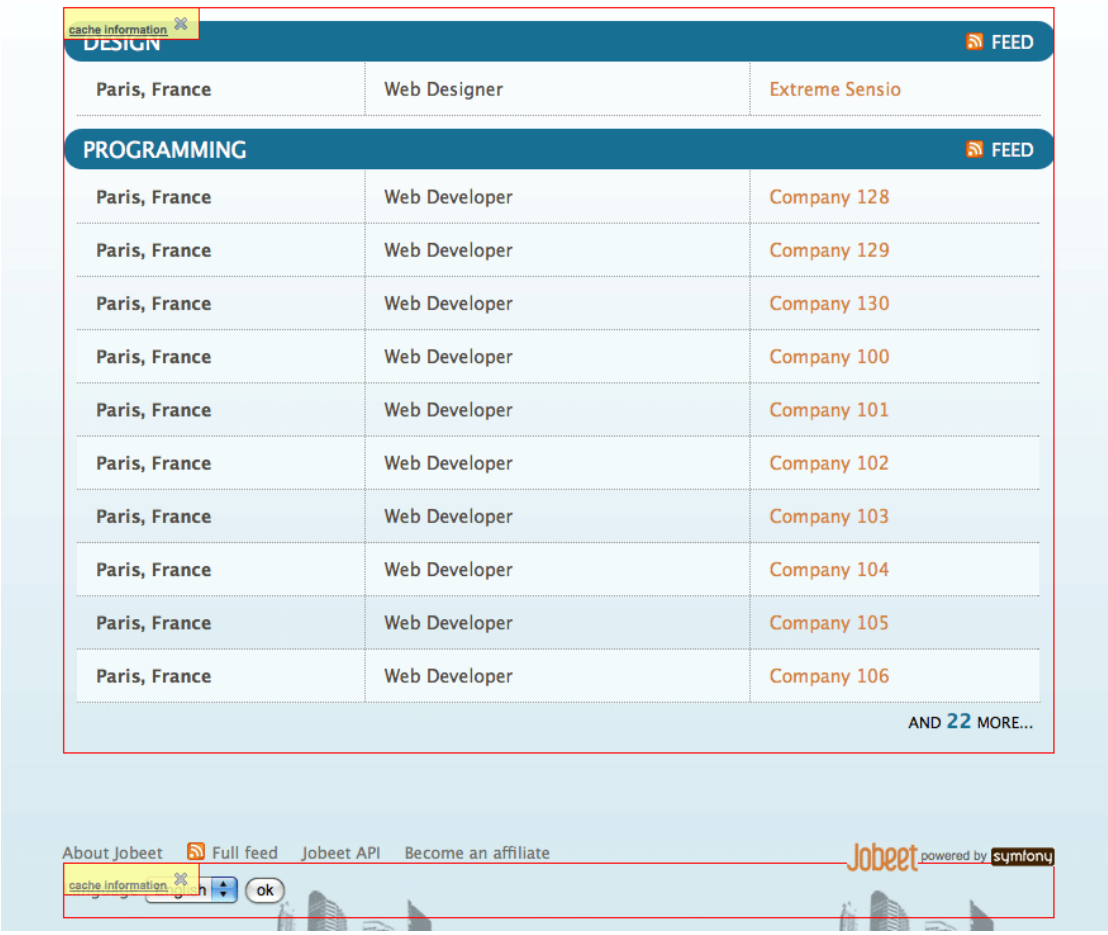

Figura 22.7. Guardando elementos parciales en la cache

A continuación vamos a guardar en la cache el componente `language` creando un archivo de configuración `cache.yml` en el módulo `sfJobeetLanguage`:

```
# plugins/sfJobeetJob/modules/sfJobeetLanguage/config/cache.yml
_language:
  enabled: on
```

Configurar las opciones de cache para un elemento parcial o un componente es tan sencillo como añadir una nueva entrada con su nombre en el archivo de configuración. La opción `with_layout` no se tiene en consideración en este tipo de cache porque no tiene ningún sentido:

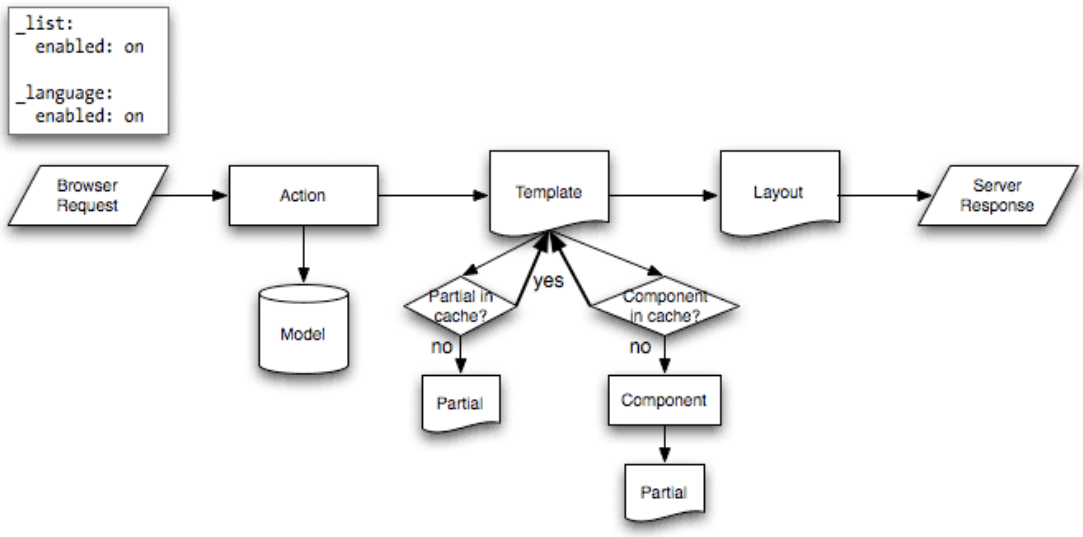

Figura 22.8. Flujo de trabajo al guardar un elemento parcial y un componente en la cache

¿Contextual o independiente?

El mismo elemento parcial o componente se puede utilizar en muchas plantillas diferentes. El elemento parcial `list` por ejemplo se utiliza en los módulos `job` y `category`. Como el resultado mostrado por el elemento parcial siempre es el mismo y no depende del contexto en el que se utiliza, todas las plantillas pueden utilizar la misma versión de la cache (obviamente la cache será diferente si cambian los parámetros del elemento parcial).

Por otra parte, en ocasiones el resultado de un elemento parcial o de un componente varía en función de la acción en la que se utiliza (imagina por ejemplo el lateral de un blog, que varía si se trata de la portada o de la página de un artículo). En estos casos, el elemento parcial o componente es contextual y debes configurar la cache estableciendo la opción `contextual` a `true`:

```
_sidebar:
  enabled:    on
  contextual: true
```

22.7. Guardando formularios en la cache

Guardar en la cache la página de publicación de ofertas de trabajo es complicado porque contiene un formulario. Para que entiendas mejor el problema, accede una vez a la página para publicar una oferta de trabajo. Ahora que la página se ha guardado en la cache, borra la cookie de la sesión y trata de publicar la oferta de trabajo. Si has seguido estos pasos, verás un mensaje de error advirtiendo de un posible ataque de tipo CSRF:

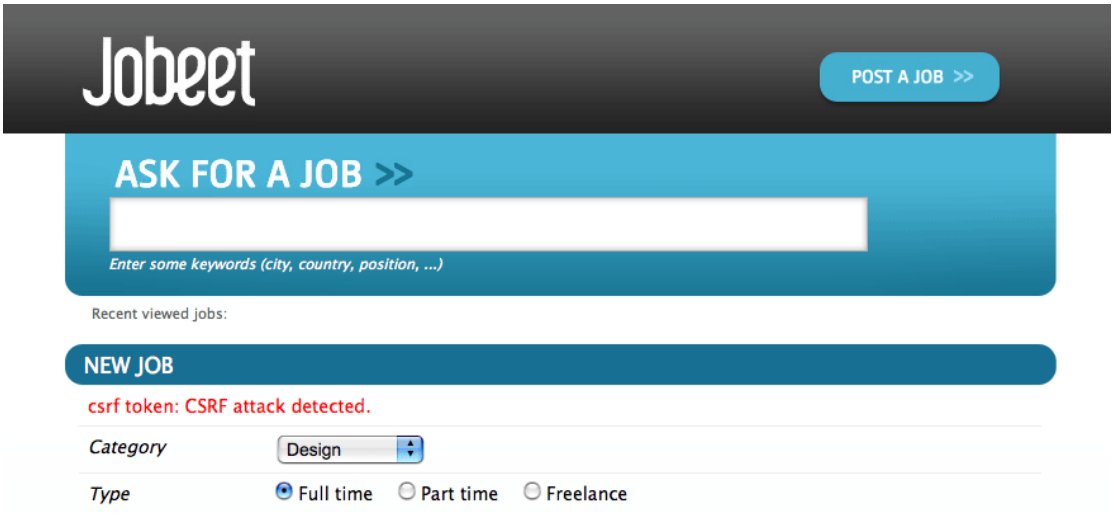

Figura 22.9. Mensaje sobre un posible ataque de tipo CSRF al usar la cache

¿Por qué sucede este error? Como al crear la aplicación `frontend` configuramos una palabra secreta relacionada con CSRF, Symfony incluye un token CSRF en todos los formularios. Para evitar ataques de tipo CSRF, el token es único para cada formulario de cada usuario.

La primera vez que accedes a la página del formulario, el codigo HTML del formulario que se guarda en la cache incluye el token del usuario actual. Si después otro usuario accede a la misma página, el navegador muestra la página guardada en la cache y que contiene el token del primer usuario. Cuando el usuario envía el formulario, Symfony detecta que los dos tokens no coinciden y muestra el mensaje de error sobre un posible ataque de tipo CSRF.

¿Cómo podríamos solucionar el problema y al mismo tiempo seguir guardando el formulario en la cache? El formulario de publicación de ofertas de trabajo no depende del usuario y no modifica ninguna información del usuario actual. Por tanto, en este caso no necesitamos activar la protección CSRF y podemos eliminar el token CSRF del formulario:

```
// plugins/sfJobeetJob/lib/form/PluginJobeetJobForm.class.php
abstract class PluginJobeetJobForm extends BaseJobeetJobForm
{
  public function __construct(BaseObject $object = null, $options = array(),
$CSRFSecret = null)
  {
    parent::__construct($object, $options, false);
  }

  // ...
}
```

Después de realizar este cambio, borra la cache y vuelve a probar el mismo escenario explicado anteriormente para comprobar que ahora todo funciona correctamente.

A continuación aplica la misma configuración al formulario para seleccionar el idioma que se encuentra en el layout y que queremos guardar en la cache. Como utilizamos el

formulario `sfLanguageForm` por defecto, en vez de crear una nueva clase sólo para eliminar el token CSRF, vamos a realizar el cambio directamente en la acción y el componente del módulo `sfJobeetLanguage`:

```
// plugins/sfJobeetJob/modules/sfJobeetLanguage/actions/components.class.php
class sfJobeetLanguageComponents extends sfComponents
{
  public function executeLanguage(sfWebRequest $request)
  {
    $this->form = new sfFormLanguage($this->getUser(), array('languages' =>
array('en', 'fr')));
    unset($this->form[$this->form->getCSRFFieldName()]);
  }
}
// plugins/sfJobeetJob/modules/sfJobeetLanguage/actions/actions.class.php
class sfJobeetLanguageActions extends sfActions
{
  public function executeChangeLanguage(sfWebRequest $request)
  {
    $form = new sfFormLanguage($this->getUser(), array('languages' =>
array('en', 'fr')));
    unset($form[$form->getCSRFFieldName()]);

    // ...
  }
}
```

El método `getCSRFFieldName()` devuelve el nombre del campo que contiene el token CSRF. Eliminar este campo del formulario provoca que también se eliminen el widget y el validador asociados al campo.

22.8. Borrando la cache

Cuando el usuario publica una nueva oferta de trabajo o cuando activa una oferta existente, debemos refrescar la portada de Jobeet para que se muestre en el listado de ofertas de trabajo.

Como no necesitamos que las ofertas de trabajo aparezcan en tiempo real en la portada, vamos a seguir la estrategia de reducir el tiempo de vida de la cache a un valor más aceptable:

```
# plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml
index:
  enabled:  on
  lifetime: 600
```

Mientras que el valor por defecto hace que la cache se guarde durante un día completo, la configuración anterior hace que la portada de Jobeet se borre de la cache cada diez minutos.

No obstante, si quieres actualizar la portada cada vez que un usuario activa una oferta de trabajo, modifica el método `executePublish()` del módulo `sfJobeetJob` para borrar la cache manualmente:

```
// plugins/sfJobeetJob/modules/sfJobeetJob/actions/actions.class.php
public function executePublish(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->publish();

  if ($cache = $this->getContext()->getViewCacheManager())
  {
    $cache->remove('sfJobeetJob/index?sf_culture=*');
    $cache->remove('sfJobeetCategory/
show?id='.$job->getJobeetCategory()->getId());
  }

  $this->getUser()->setFlash('notice', sprintf('Your job is now online for %s
days.', sfConfig::get('app_active_days')));

  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

La cache se gestiona mediante la clase `sfViewCacheManager`, cuyo método `remove()` borra la cache asociada con la URI interna indicada. Si quieres eliminar la cache para todos los posibles valores de una variable, utiliza `*` como valor. El valor `sf_culture=*` utilizado en el ejemplo anterior significa que Symfony elimina de la cache tanto la portada en inglés como la portada en francés.

El borrado de la cache lo hemos incluido dentro de un bloque `if()` porque el gestor de la cache vale `null` cuando la cache se encuentra deshabilitada.

La clase sfContext

El objeto `sfContext` contiene referencias a los objetos internos de Symfony como la petición, la respuesta, el usuario, etc. El objeto `sfContext` actúa como un *singleton*, por lo que puedes utilizar la instrucción `sfContext::getInstance()` en cualquier punto de la aplicación para tener acceso directo a los objetos internos de Symfony:

```
$user = sfContext::getInstance()->getUser();
```

Te recomendamos que te lo pienses dos veces antes de utilizar `sfContext::getInstance()` en alguna de tus clases, ya que su uso impide que el código de la aplicación sea desacoplado. La mejor alternativa consiste en pasar como argumento el objeto que necesitas.

Si lo necesitas, también puedes emplear `sfContext` como un registro en el que puedes añadir tus propios objetos mediante el método `set()` indicando como parámetros el nombre del objeto y el propio objeto. Para obtener de nuevo los objetos, utiliza el método `get()` pasando como argumento el nombre con el que guardaste el objeto:

```
sfContext::getInstance()->set('job', $job);
$job = sfContext::getInstance()->get('job');
```

22.9. Probando la cache

Antes de crear las pruebas, tenemos que activar la cache para el entorno `test` modificando su archivo de configuración:

```
# apps/frontend/config/settings.yml
test:
  .settings:
    error_reporting: <?php echo ((E_ALL | E_STRICT) ^ E_NOTICE)."\n" ?>
    cache:           on
    web_debug:       off
    etag:            off
```

Utiliza el siguiente código para probar la página de publicación de una nueva oferta de trabajo:

```
// test/functional/frontend/jobActionsTest.php
$browser->
  info('  7 - Job creation page')->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->

  createJob(array('category_id' =>
$browser->getProgrammingCategory()->getId()), true)->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->
  with('response')->checkElement('.category_programming .more_jobs', '/23/')
;
```

El *tester* `view_cache` se utiliza para probar la cache. El método `isCached()` requiere dos valores *booleanos*:

- El primero indica si la página debe encontrarse en la cache
- El segundo indica si la página debe guardarse en la cache junto con su layout

Sugerencia

Aunque el framework para pruebas funcionales incluye muchas herramientas útiles, en ocasiones es más sencillo descubrir los problemas en el navegador. Para ello, crea un controlador frontal asociado al entorno de pruebas `test` y echa un vistazo al archivo de log generado en `log/frontend_test.log`.

22.10. Nos vemos mañana

Como muchas otras características de Symfony, el subframework de la cache es muy flexible y permite al programador realizar una configuración increíblemente detallada.

Mañana hablaremos del último paso en el desarrollo de una aplicación: la instalación en los servidores de producción.

Capítulo 23. Pasando a producción

Después de la configuración de la cache que hicimos ayer, el sitio web de Jobeet ya está preparado para instalarlo en los servidores de producción.

A lo largo de 22 días hemos desarrollado Jobeet en una máquina de desarrollo, lo que para la mayoría de vosotros significa que lo habéis desarrollado en vuestro propio ordenador. Si por el contrario habéis programado directamente en el servidor de producción, os aconsejamos que no lo sigáis haciendo para los siguientes proyectos. Por tanto, el siguiente paso consiste en pasar el sitio web a producción.

Hoy vamos a explicar lo que debes hacer antes de pasar a producción, las diferentes estrategias que existen para instalar las aplicaciones y te mostraremos las herramientas más útiles para realizar una buena instalación.

23.1. Preparando el servidor de producción

Antes de instalar la aplicación en producción, asegúrate de que el servidor de producción está correctamente configurado. Quizás necesites volver a leer el tutorial del primer día, donde explicamos cómo configurar el servidor web.

En esta sección suponemos que ya tienes un servidor web, una base de datos y PHP 5.2.4 o posterior correctamente instalados.

> **Nota**
>
> Si tu servidor web no permite el acceso mediante SSH, puedes saltarte la sección en la que necesitas acceder a la línea de comandos.

23.1.1. Configuración del servidor

El primer paso consiste en comprobar que tanto PHP como algunas de sus extensiones están correctamente instaladas y configuradas. Tal y como explicamos durante el primer día, utiliza el script `check_configuration.php` que incluye Symfony. Como en el servidor web no vamos a instalar Symfony, descarga directamente el script desde la siguiente dirección:

```
http://trac.symfony-project.org/browser/branches/1.2/data/bin/
check_configuration.php?format=raw
```

Copia el archivo descargado al directorio raíz de tu servidor web y ejecútalo desde un navegador y desde la línea de comandos:

```
$ php check_configuration.php
```

Corrige todos los errores graves que muestre el script hasta que ya no veas ningún error ni en el navegador ni en la línea de comandos.

23.1.2. Aceleradores PHP

En los servidores de producción siempre se intenta conseguir el máximo rendimiento posible. Instalar un acelerador de PHP (http://en.wikipedia.org/wiki/PHP_accelerator) es una de las formas más sencillas y baratas de mejorar el rendimiento.

> **Nota**
>
> Según la definición de la Wikipedia: *"el funcionamiento de los aceleradores de PHP consiste en guardar en una cache el "bytecode" generado al compilar los scripts de PHP. De esta forma, se evita tener que procesar y compilar el código fuente del script en cada petición"*

APC (http://www.php.net/apc) es uno de los aceleradores más populares y uno de los más fáciles de instalar:

```
$ pecl install APC
```

Dependiendo del sistema operativo que utilices, es posible que puedas instalarlo incluso mediante el gestor de paquetes del propio sistema operativo.

> **Nota**
>
> Te aconsejamos que dediques un tiempo a aprender cómo configurar APC (http://www.php.net/manual/es/apc.configuration.php) .

23.2. Las librerías de Symfony

23.2.1. Incluyendo Symfony

Una de las principales ventajas de Symfony es que los proyectos son autosuficientes. Todos los archivos que necesita un proyecto para funcionar se encuentran bajo el directorio raíz del proyecto. Además, como Symfony sólo utiliza rutas relativas, puedes mover el directorio del proyecto de un sitio a otro y todo seguirá funcionando correctamente sin necesidad de realizar ningún cambio. Por tanto, no es obligatorio que el directorio de producción sea el mismo que el directorio de la máquina de desarrollo.

La única ruta absoluta que puede que te encuentres está en el archivo `config/ProjectConfiguration.class.php`, pero ya la arreglamos durante el primer día. Comprueba que ese archivo contenga una ruta relativa al cargador automático de clases de Symfony:

```
// config/ProjectConfiguration.class.php
require_once dirname(__FILE__).'/../lib/vendor/symfony/lib/autoload/
sfCoreAutoload.class.php';
```

23.2.2. Actualizando Symfony

Aunque todo el proyecto se encuentra en un único directorio, actualizar la versión de Symfony es muy sencillo.

Como los creadores de Symfony están continuamente corrigiendo errores y posibles fallos de seguridad, de vez en cuando te tocará actualizar las librerías de Symfony a la última versión disponible en la rama de desarrollo que utilizas. Como puede que ya sepas, todas las versiones de Symfony se mantienen al menos durante un año y en todo ese tiempo nunca se añaden nuevas características, ni siquiera la más mínima. De esta forma, actualizar Symfony a la última versión estable de cada rama de desarrollo siempre es seguro, rápido y fiable.

Actualizar la versión de Symfony es tan sencillo como modificar el contenido del directorio `lib/vendor/symfony/`. Si has instalado Symfony mediante un archivo comprimido, elimina todos los archivos de ese directorio y copia los contenidos del nuevo archivo comprimido que has descargado.

Si en tu proyecto utilizas Subversion, puedes enlazar ese directorio con la *tag* de la última versión disponible de Symfony 1.2 en el repositorio:

```
$ svn propedit svn:externals lib/vendor/
  # symfony http://svn.symfony-project.com/tags/RELEASE_1_2_1/
```

Actualizar ahora la versión de Symfony es tan sencillo como modificar la *tag* a la que se enlaza dentro del repositorio.

Otra alternativa consiste en enlazar directamente con la rama o *branch* 1.2 del repositorio para obtener todos los cambios en tiempo real:

```
$ svn propedit svn:externals lib/vendor/
  # symfony http://svn.symfony-project.com/branches/1.2/
```

Con la configuración anterior, cada vez que ejecutas el comando `svn up`, se instala en el proyecto la última versión disponible de Symfony 1.2.

Te aconsejamos que cada vez que te actualices a una nueva versión borres la cache de Symfony, sobre todo en el entorno de producción:

```
$ php symfony cc
```

Sugerencia

Si tienes acceso mediante FTP al servidor de producción, puedes emular el efecto del comando `symfony cc` borrando todos los archivos y directorios que se encuentran en el directorio `cache/`.

Si quieres, también es posible probar una versión de Symfony sin desinstalar la versión anterior. Si quieres probar una nueva versión de Symfony y poder volver fácilmente a la versión original, instala la nueva versión en otro directorio (por ejemplo `lib/vendor/symfony_test`), modifica la ruta hasta Symfony en la clase `ProjectConfiguration`, borra la cache y ya puedes probar la nueva versión. Si algo sale mal, puedes volver a la situación anterior borrando el directorio nuevo y volviendo a modificar la ruta hasta Symfony en la clase `ProjectConfiguration`.

23.3. Ajustando la configuración

23.3.1. Configuración de la base de datos

En la mayoría de ocasiones, los datos de conexión con la base de datos de producción son diferentes de los datos de conexión en local. Gracias a los entornos de ejecución de Symfony, es muy sencillo definir una configuración diferente para la base de datos de producción:

```
$ php symfony configure:database "mysql:host=localhost;dbname=prod_dbname"
prod_user prod_pass
```

Recuerda que también puedes realizar la configuración de la base de datos editando a mano el archivo `databases.yml`.

23.3.2. Archivos web

Como Jobeet utiliza plugins que incluyen archivos web (CSS y JavaScript), Symfony crea enlaces simbólicos relativos en el directorio `web/` del proyecto. La tarea `plugin:publish-assets` regenera o crea estos enlaces simbólicos cuando se instalan plugins sin utilizar la tarea `plugin:install`:

```
$ php symfony plugin:publish-assets
```

23.3.3. Páginas de error propias

Antes de subir la aplicación a producción, es conveniente que personalices las páginas de error de Symfony como por ejemplo la página de *"Error 404: Página No Encontrada"* o la página que muestra las excepciones.

Durante el tutorial del día 16 ya configuramos la página de error del formato `YAML` creando los archivos `error.yaml.php` y `exception.yaml.php` en el directorio `config/error/`. Symfony utiliza el archivo `error.yaml.php` en el entorno `prod` mientras que el archivo `exception.yaml.php` se emplea en el entorno `dev`.

Por tanto, para personalizar las páginas de error de las excepciones del formato `HTML`, crea los archivos `config/error/error.html.php` y `config/error/exception.html.php`.

La página del error `404` (*"página no encontrada"*) se puede personalizar modificando las opciones de configuración `error_404_module` y `error_404_action`:

```
# apps/frontend/config/settings.yml
all:
  .actions:
    error_404_module: default
    error_404_action: error404
```

23.4. Modificando la estructura de directorios

Symfony utiliza una estructura de directorios predefinida que permite organizar y estandarizar mejor el código de las aplicaciones. No obstante, en ocasiones no puedes

utilizar esa estructura de directorios porque tienes que seguir las normas de trabajo impuestas por otras personas.

La clase `config/ProjectConfiguration.class.php` permite configurar el nombre de cada directorio.

23.4.1. El directorio web raíz

En algunos servicios de hosting no puedes modificar el nombre del directorio web raíz. Imagina que en tu servidor compartido ese directorio se llama `public_html/` en vez de `web/`:

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

El método `setWebDir()` utiliza como argumento la ruta absoluta hasta el directorio web raíz. Si modificas también la localización del directorio en el que se encuentra el archivo `ProjectConfiguration.class.php`, no te olvides de actualizar su ruta en todos los controladores frontales:

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

23.4.2. Los directorios de cache y de log

El framework Symfony sólo escribe en dos directorios: `cache/` y `log/`. Por motivos de seguridad, algunos servicios de hosting no establecen permisos de escritura en el directorio principal. Si este es tu caso, puedes mover estos directorios a cualquier otro directorio del servidor:

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->setCacheDir('/tmp/symfony_cache');
    $this->setLogDir('/tmp/symfony_logs');
  }
}
```

Como sucede con el método `setWebDir()`, a los métodos `setCacheDir()` y `setLogDir()` se les pasa como argumento la ruta absoluta hasta los nuevos directorios `cache/` y `log/` respectivamente.

23.5. Las factorías

A lo largo del tutorial de Jobeet hemos hablado de los objetos internos de Symfony como `sfUser`, `sfRequest`, `sfResponse`, `sfI18N`, `sfRouting`, etc. El framework Symfony crea, configura y gestiona automáticamente todos estos objetos. Además, estos objetos siempre son accesibles a través del objeto `sfContext`, y como muchos otros elementos del framework, se pueden configurar a través de un archivo de configuración llamado `factories.yml`. Este archivo también permite establecer diferentes opciones para cada entorno.

Cuando `sfContext` inicializa las factorías, lee el contenido del archivo `factories.yml` para determinar el nombre de las clases (`class`) y los parámetros (`param`) que se pasan al constructor:

```
response:
  class: sfWebResponse
  param:
    send_http_headers: false
```

El código anterior hace que cuando Symfony cree la factoría de los objetos de la respuesta, instancie un objeto de la clase `sfWebResponse` y pase `send_http_headers` como argumento al constructor.

Como puedes personalizar las factorías, es posible emplear tus propias clases para los objetos internos de Symfony en vez de los objetos por defecto. También puedes modificar el comportamiento de las clases por defecto variando los parámetros que se les pasan.

A continuación vamos a ver algunas de las configuraciones propias más interesantes.

23.5.1. El nombre de la cookie

Symfony utiliza una cookie para gestionar las sesiones de usuario. Por defecto, esta cookie se llama `symfony`, pero se puede modificar en el archivo `factories.yml`. Dentro de la sección `all`, añade lo siguiente para cambiar el nombre de la cookie por `jobeet`:

```
# apps/frontend/config/factories.yml
storage:
  class: sfSessionStorage
  param:
    session_name: jobeet
```

23.5.2. Cómo se guardan las sesiones

La clase por defecto encargada de guardar las sesiones se llama `sfSessionStorage`. Esta clase hace uso del sistema de archivos para guardar toda la información de las sesiones. Si dispones de varios servidores web, quizás te interese centralizar el almacenamiento de las sesiones en una base de datos:

```
# apps/frontend/config/factories.yml
storage:
  class: sfPDOSessionStorage
  param:
    session_name: jobeet
    db_table:     session
    database:     propel
    db_id_col:    id
    db_data_col:  data
    db_time_col:  time
```

23.5.3. El tiempo de expiración de las sesiones

El tiempo de expiración por defecto de las sesiones de usuario es de `1800` segundos. Si quieres modificarlo, hazlo en la sección `user`:

```
# apps/frontend/config/factories.yml
user:
  class: myUser
  param:
    timeout: 1800
```

23.5.4. Mensajes de log

El entorno `prod` no genera por defecto ningún mensaje de log, ya que la clase utilizada por su *logger* es `sfNoLogger`:

```
# apps/frontend/config/factories.yml
prod:
  logger:
    class:   sfNoLogger
    param:
      level:   err
      loggers: ~
```

Si quieres que se guarden los mensajes de log en algún archivo, puedes cambiar el nombre de la clase de su *logger* por `sfFileLogger`:

```
# apps/frontend/config/factories.yml
logger:
  class: sfFileLogger
  param:
    level: error
    file:  %SF_LOG_DIR%/%SF_APP%_%SF_ENVIRONMENT%.log
```

Nota

En el archivo de configuración `factories.yml`, las cadenas de texto con el formato `%XXX%` se reemplazan por su valor correspondiente del objeto `sfConfig`. Por tanto, utilizar `%SF_APP%` en un archivo de configuración es equivalente a utilizar `sfConfig::get('sf_app')` en el código PHP. Esta notación también se puede utilizar en el archivo `app.yml`. Su principal utilidad es que permite hacer referencia a la ruta de un directorio sin tener que escribir la ruta completa en el archivo de configuración (simplemente debes indicar `SF_ROOT_DIR`, `SF_WEB_DIR`, etc.)

23.6. Instalando aplicaciones

23.6.1. ¿Qué tienes que instalar?

Cuando subimos la aplicación Jobeet a producción, tenemos que tener mucho cuidado de no subir archivos innecesarios y de no borrar los archivos subidos por los usuarios, como por ejemplo los logotipos de las empresas.

En los proyectos creados con Symfony siempre hay tres directorios que no tienes que subir a producción: `cache/`, `log/` y `web/uploads/`. El resto de archivos y directorios puedes subirlos a producción tal y como están.

No obstante, por motivos de seguridad no es buena idea subir los controladores frontales de los entornos que no sean `prod`, como por ejemplo `frontend_dev.php` y `frontend_cache.php`.

23.6.2. Estrategias para la instalación

En esta sección, suponemos que tienes el control absoluto sobre los servidores de producción. Si sólo puedes acceder al servidor con una cuenta de FTP, sólo puedes instalar las aplicaciones Symfony subiendo todos sus archivos cada vez que quieres instalar la aplicación.

La forma más sencilla de instalar tu sitio web en el servidor consiste en utilizar la tarea `project:deploy`. Esta tarea hace uso de `SSH` y `rsync` para realizar la conexión con el servidor y para transferir todos los archivos de un servidor a otro.

Los servidores se configuran en el archivo `config/properties.ini`:

```
# config/properties.ini
[production]
  host=www.jobeet.org
  port=22
  user=jobeet
  dir=/var/www/jobeet/
  type=rsync
  pass=
```

Si quieres instalar la aplicación en el servidor `production` que acabas de configurar, utiliza la tarea `project:deploy`:

```
$ php symfony project:deploy production
```

Nota

Antes de ejecutar por primera vez la tarea `project:deploy`, es necesario que te conectes al servidor y añadas la clave a mano en el archivo de *hosts* conocidos.

Puedes ejecutar tranquilamente el comando anterior porque Symfony sólo simula la transferencia de archivos, pero no los transfiere realmente. Para instalar de verdad el sitio web, debes utilizar la opción `--go`:

```
$ php symfony project:deploy production --go
```

Nota

Aunque en el archivo `properties.ini` puedes incluir la contraseña de SSH, es mucho mejor configurar el servidor con claves SSH que permitan realizar conexiones sin contraseña.

Por defecto Symfony no transfiere ninguno de los directorios comentados anteriormente y tampoco copia los controladores frontales del entorno `dev`. El motivo es que la tarea `project:deploy` excluye los archivos y directorios configurados en el archivo `config/rsync_exclude.txt`:

```
# config/rsync_exclude.txt
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
```

En el caso de Jobeet, vamos a añadir a la lista el controlador frontal `frontend_cache.php`:

```
# config/rsync_exclude.txt
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
/web/frontend_cache.php
```

Sugerencia

También puedes crear un archivo `config/rsync_include.txt` para obligar a que se transfieran ciertos archivos y/o directorios.

Aunque la tarea `project:deploy` es bastante flexible, puede que necesites configurarla todavía más. Como el proceso de instalar aplicaciones varía mucho en función de la configuración y topología de tus servidores, no dudes en crearte tu propia tarea para instalar aplicaciones.

Por último, cada vez que instales una aplicación web en producción, no te olvides de borrar como mínimo la cache de configuración en el servidor de producción:

```
$ php symfony cc --type=config
```

Si has modificado alguna ruta, también tienes que borrar la cache del sistema de enrutamiento:

```
$ php symfony cc --type=routing
```

Nota

Borrar solamente algunas partes de la cache tiene la ventaja de que puedes mantener el resto de la cache, como por ejemplo la parte que guarda las plantillas.

23.7. Nos vemos mañana

Instalar el proyecto en los servidores de producción es el último paso en el desarrollo de una aplicación Symfony. No obstante, esto no significa que haya terminado tu trabajo. En realidad, tu trabajo no ha hecho más que comenzar, ya que las aplicaciones web no son elementos inertes, sino que evolucionan con el tiempo. Seguramente tendrás que corregir algunos errores que has descubierto y añadirás nuevas funcionalidades en la aplicación. Afortunadamente, la estructura y herramientas de Symfony hacen que actualizar un sitio web sea algo sencillo, rápido y seguro.

Mañana es el último tutorial de Jobeet, por lo que echaremos la vista atrás y repasaremos todo lo que hemos aprendido durante los 23 días anteriores.

Capítulo 24. Un repaso a Symfony

Hoy es la última etapa del viaje que hemos realizado por el mundo de Symfony. Durante los últimos 23 días has podido aprender a utilizar Symfony a través de un ejemplo, desde los patrones de diseño utilizados por el framework hasta sus características más avanzadas. Aunque todavía no puedes considerarte un maestro de Symfony, ya dispones de todos los conocimientos que necesitas para empezar a desarrollar aplicaciones Symfony con total confianza.

Ahora que finalizamos el tutorial de Jobeet, vamos a mostrar un punto de vista diferente del framework. Olvídate de Jobeet durante una hora y recuerda todas las funcionalidades que has aprendido durante las últimas tres semanas.

24.1. ¿Qué es Symfony?

El framework Symfony es un conjunto de subframeworks independientes pero cohesionados que forman un completo framework MVC (*Modelo, Vista, Controlador*).

Antes de empezar a programar, dedica un tiempo a leer la historia y filosofía de trabajo de Symfony. Después, repasa los requisitos técnicos de Symfony y utiliza el script check_configuration.php para probar tu configuración.

Por último, instala Symfony. Después de trabajar durante un tiempo con Symfony, seguramente tendrás que actualizarlo a una versión más reciente del framework.

El framework también incluye herramientas que facilitan la instalación de aplicaciones.

24.2. El modelo

La parte del modelo de Symfony se puede desarrollar con ayuda del ORM Propel. A partir de la descripción de la base de datos, genera clases para los objetos, formularios y filtros. Propel también genera las sentencias SQL que se utilizan para crear las tablas de la base de datos.

La configuración de la base de datos se puede realizar mediante una tarea o editando un archivo de configuración. Además de su configuración, es posible insertar datos de prueba en la base de datos mediante los archivos de datos. Incluso es posible crear archivos de datos dinámicos.

Los objetos Propel también pueden ser fácilmente internacionalizados.

24.3. La vista

Por defecto, la capa de la vista de la arquitectura MVC utiliza archivos PHP normales como plantillas.

Las plantillas pueden hacer uso de helpers para facilitar las tareas habituales como crear URL o enlaces.

Las plantillas se decoran mediante un layout para abstraer tanto la cabecera como el pie de las páginas. Para hacer las plantillas más reutilizables, puedes emplear slots, elementos parciales y componentes.

Para mejorar el rendimiento de la aplicación, puedes utilizar el subframework de la cache para guardar en la cache la página entera, sólo la acción e incluso sólo los elementos parciales o componentes. También puedes borrar la cache manualmente.

24.4. El controlador

La parte del controlador se gestiona mediante los controladores frontales y las acciones.

Existen tareas para crear módulos sencillos, módulos CRUD e incluso para generar módulos de administración completos para las clases del modelo.

Los módulos de administración permiten crear una aplicación completamente funcional sin necesidad de escribir ni una sola línea de código.

Para abstraer el funcionamiento interno del sitio web, Symfony utiliza un subframework de enrutamiento que genera URL limpias. Para facilitar el desarrollo de servicios web, Symfony incluye el soporte de los formatos. También puedes crear tus propios formatos.

Las acciones se pueden reenviar o redirigir a otra acción.

24.5. Configuración

El framework Symfony permite establecer diferentes opciones de configuración para cada entorno. Un entorno es un conjunto de opciones que permiten variar el comportamiento de la aplicación en función de si se ejecuta en el servidor de desarrollo o en el de producción. También puedes crear nuevos entornos.

Los archivos de configuración de Symfony se pueden definir en diferentes niveles y la mayoría permiten definir opciones dependientes del entorno:

- app.yml
- cache.yml
- databases.yml
- factories.yml
- generator.yml
- routing.yml
- schema.yml
- security.yml
- settings.yml

- view.yml

La mayoría de archivos de configuración utilizan el formato YAML.

Si no quieres utilizar la estructura de directorios por defecto que organiza los archivos de la aplicación en capas, puedes organizarlos por funcionalidad y agruparlos en un plugin. Hablando de la estructura de directorios por defecto, también puedes modificarla para que se adapte a tus necesidades.

24.6. Depuración

Symfony incluye muchas utilidades para ayudar a los programadores a depurar los errores más fácilmente, como por ejemplo los archivos de log, la barra de depuración web y las excepciones útiles.

24.7. Los principales objetos de Symfony

El framework Symfony incluye varios objetos que abstraen las necesidades habituales de los proyectos web: la petición, la respuesta,el usuario, los mensajes de log, el sistema de enrutamiento y el gestor de la cache de la vista.

Todos los objetos anteriores se gestionan a través del objeto sfContext y se configuran mediante las factorías

El objeto del usuario gestiona la autenticación, la autorización, los mensajes flash y los atributos que se guardan en la sesión del usuario.

24.8. Seguridad

El framework Symfony incluye protección frente a ataques de tipo XSS y CSRF. Estas opciones se pueden configurar desde la línea de comandos o editando un archivo de configuración.

El framework de formularios también incluye varias medidas de seguridad.

24.9. Formularios

Como trabajar con formularios es una de las tareas más tediosas para un programador web, Symfony incluye un subframework de formularios. Este framework de formularios incluye numerosos widgets y validadores. Uno de los puntos fuertes de los formularios es que sus plantillas se pueden personalizar muy fácilmente.

Si utilizas Propel, el framework de formularios también permite generar formularios y filtros de forma sencilla a partir de los modelos de datos.

24.10. Internacionalización y localización

Symfony soporta la internacionalización y localización mediante el estándar ICU. El idioma y el país del usuario se controlan mediante la cultura del usuario. Además, la cultura la puede definir el usuario o se puede incluir en la propia URL.

24.11. Pruebas

Para las **pruebas unitarias** se emplea la librería `lime`, que incluye numerosos métodos para pruebas. También se pueden probar los objetos Propel mediante una bases de datos específica y unos archivos de datos específicos.

Las pruebas unitarias se pueden ejecutar individualmente o todas a la vez.

Las **pruebas funcionales** se crean mediante la clase sfFunctionalTest, que emplea un simulador de navegador y permite la introspección de los objetos internos de Symfony mediante los testers. Symfony incluye testers para el objeto de la petición, el objeto de la respuesta, el objeto del usuario, el objeto del formulario actual, la capa de la cache y los objetos de Propel.

También existen herramientas para depurar tanto la respuesta como los formularios.

Al igual que las pruebas unitarias, las pruebas funcionales se pueden ejecutar individualmente o todas a la vez.

Si quieres también puedes ejecutar todas las pruebas a la vez, tanto unitarias como funcionales.

24.12. Plugins

El framework Symfony sólo proporciona la base para desarrollar las aplicaciones web y delega en los plugins la creación de más funcionalidades. A lo largo de este tutorial hemos hablado de los plugins sfGuardPlugin, sfFormExtraPlugin y sfTaskExtraPlugin.

Después de instalar un plugin, debes activarlo.

Por último, los plugins son la mejor forma de devolver al proyecto Symfony parte de lo recibido.

24.13. Tareas

La línea de comandos de Symfony incluye muchas tareas, la mayoría de las cuales se han visto en este tutorial:

- app:routes
- cache:clear
- configure:database
- generate:project

- generate:app
- generate:module
- help
- i18n:extract
- list
- plugin:install
- plugin:publish-assets
- project:deploy
- propel:build-all
- propel:build-all-load
- propel:build-forms
- propel:build-model
- propel:build-sql
- propel:data-load
- propel:generate-admin
- propel:generate-module
- propel:insert-sql
- test:all
- test:coverage
- test:functional
- test:unit

También es posible crear tus propias tareas.

24.14. Agradecimientos

Escribir un libro es una tarea tan excitante como agotadora. Escribir un libro técnico es todavía más agotador. Hemos dedicado multitud de horas a pensar en cómo transmitir la información, cómo explicar cada concepto y como incluir ejemplos sencillos pero completos y reutilizables.

Escribir un tutorial tan grande es imposible sin contar con gente a tu alrededor que te apoye durante todo el proceso.

El mayor apoyo siempre lo recibes de tu propia familia. **Fabien Potencier**, el autor original del libro, tiene la fortuna de contar con la familia más comprensiva del mundo. Como buen emprendedor que es, Fabien pasa la mayor parte de su tiempo trabajando. Como máximo responsable de Symfony, Fabien dedica casi todo su tiempo libre a idear la próxima versión del framework. Y por si fuera poco, Fabien decidió ponerse a escribir

otro libro. Sin el apoyo de su mujer Hélène y de sus dos hijos Thomas y Lucas, no hubiera sido posible escribir un libro de este tipo en tan poco tiempo.

Fabien también ha recibido la ayuda de varios revisores de primer nivel. Todos ellos son parte de la comunidad de Symfony y quiere agradecerles el tiempo dedicado al proyecto Jobeet.

Kris Wallsmith, es el responsable de la comunidad de Symfony y será el próximo responsable del lanzamiento de Symfony 1.3. Kris se dedicó a leer y corregir mi muy mejorable inglés. Como este tutorial se publicó durante todos los días, y Fabien vive en Francia y Kris en Estados Unidos, Kris se tuvo que levantar muy pronto cada mañana, incluso los fines de semana, para leer y corregir cada tutorial.

Stefan Koopmanschap, uno de los evangelizadores de Symfony más activos, se encargó del repositorio de Subversion. Gracias a su esfuerzo, puedes obtener el código y empezar a leer el tutorial a partir de cualquier día.

Fabian Lange, el responsable del lanzamiento de Symfony 1.2, leyó los contenidos del tutorial desde una perspectiva Windows y desde el punto de vista de un usuario novato. Por cierto, se acaba de comprar un Mac, así que necesitamos a otro usuario que asuma la responsabilidad de probar las cosas en Windows.

Jonathan Wage, el programador jefe de Doctrine, dedicó mucho esfuerzo a crear la edición del tutorial para Doctrine. Gracias a su trabajo, ahora puedes elegir leer el tutorial para Propel o para Doctrine, en función del ORM que utilices.

Pascal Borreli, un usuario muy activo en el canal IRC francés de Symfony y el miembro más amigable de la comunidad Symfony. Su trabajo consistió en leer todos los capítulos lo más rápido posible. Su apoyo continuo y sus amables palabras mantuvieron a Fabien de buen humor para poder escribir el tutorial desde el principio hasta el final.

Como presidente de la empresa Sensio, Fabien también tiene muchas responsabilidades. Por ello agradece el apoyo de todo el equipo de Sensio, sin el cual este libro no hubiera sido posible. Fabien agradece de forma especial el apoyo de Grégory Pascal, su socio desde hace 10 años, que al principio era muy reticente sobre el modelo de negocio del software libre pero que ahora lo apoya completamente. Por último, Fabien también agradece la ayuda de Laurent Vaquette, que le ayuda a resolver muchos problemas del día a día y con el que suele ir a comer un *döner kebab*.

Un agradecimiento especial debe ser para todos los lectores del libro online que han enviado comentarios y sugerencias desde el primer dia. Los lectores han descubierto muchos pequeños y no tan pequeños errores, inconsistencias y conceptos que no estaban demasiado bien explicados.

Si estás leyendo estas líneas en un libro impreso, Fabien te considera todo un héroe. Comprar un libro que puedes leer gratis en Internet es la mejor prueba de que apoyas el proyecto de software libre Symfony.

Por último, Fabien agradece al sitio web lulu.com lo fácil que es publicar tus propios libros. Se trata de un servicio muy rápido y divertido, que demuestra el inmenso poder

de Internet. Gracias a su sencillez, cada vez que compras el libro en lulu.com disfrutas de la última versión con todas las correcciones de errores.

Merci à tous !

24.15. Nos vemos pronto

Antes de que te vayas, nos gustaría hablarte de una última cosa acerca de Symfony. El framework tiene muchas características geniales y mucha documentación gratuita. Sin embargo, uno de los activos más valiosos que puede tener un proyecto de software libre es su comunidad. Afortunadamente, Symfony tiene una de las comunidades más activas y alucinantes que existen. Si vas a utilizar Symfony en tus proyectos, quizás te interese unirte a la comunidad de Symfony:

- Puedes suscribirte a la lista de correo oficial de usuarios de Symfony en inglés
- Puedes suscribirte a la lista de correo oficial de usuarios de Symfony en español
- Puedes suscribirte al canal RSS del blog oficial.
- Puedes suscribirte al canal RSS del planeta Symfony
- Puedes entrar a chatear en el canal #symfony (inglés) o en el canal #symfony-es (español) del IRC.

Sugerencia

Una de las formas más sencillas de acceder a los canales del IRC es el uso del navegador Firefox junto con su extensión ChatZilla.