BUENAS PRÁCTICAS PARA APLICACIONES SYMFONY

FABIEN POTENCIER
RYAN WEAVER
JAVIER EGUILUZ

LIBROSWEB

Esta página se ha dejado vacía a propósito

Índice de contenidos

Capítulo 1.

Las buenas prácticas del framework Symfony

El framework Symfony es conocido por ser **muy flexible**, ya que se utiliza en ámbitos tan diferentes como sitios web diminutos, aplicaciones empresariales que sirven miles de millones de peticiones e incluso como base de otros frameworks.

Desde que se publicó en julio de 2011, la comunidad Symfony ha recorrido un largo camino de aprendizaje, tanto en lo que se refiere a qué es capaz Symfony de hacer, como en cuál es la mejor manera de hacerlo.

Los diferentes recursos creados por la comunidad, desde artículos de blogs hasta presentaciones en conferencias, han originado una serie de recomendaciones y buenas prácticas *oficiosas* para desarrollar aplicaciones Symfony. Lamentablemente, muchas de estas recomendaciones son erróneas. De hecho, complican en exceso el desarrollo de aplicaciones y no siguen en absoluto la filosofía pragmática de los creadores de Symfony.

1.1. ¿En qué consiste esta guía?

El objetivo de esta guía es solucionar el problema anterior estableciendo una serie de **buenas prácticas oficiales para desarrollar aplicaciones con el framework Symfony2**. Estas son las buenas prácticas que mejor encajan con la filosofía del framework según su principal responsable, Fabien Potencier (https://connect.sensiolabs.com/profile/fabpot) .

Sabemos que es difícil deshacerse de los hábitos adquiridos durante años y por eso puede que alguna de estas buenas prácticas te choque o no estés de acuerdo con ella. En cualquier caso, creemos que si sigues estas buenas prácticas podrás desarrollar aplicaciones mucho más rápido, mucho más fácilmente y con la misma o más calidad de siempre. Además, esta lista de buenas prácticas se ampliará y actualizará de manera continua a partir de ahora.

En cualquier caso, ten en cuenta que esto son **recomedaciones opcionales** que tú y tu equipo podéis seguir o no al desarrollar aplicaciones Symfony. Si prefieres seguir utilizando tus propias metodolo-

gías y buenas prácticas, eres libre de hacerlo. Symfony es flexible y se adaptará a tu forma de trabajar. Eso nunca va a cambiar.

1.2. A quién se dirige este libro

Esta guía está pensada para cualquier programador Symfony, sin importar si es experto o principiante. No obstante, como esta guía no es un tutorial paso a paso, es necesario disponer de ciertos conocimientos básicos de Symfony para poder seguirla. Y si no sabes nada de Symfony, ¡bienvenido a la comunidad! y no olvides leer primero el tutorial *Getting Started* (http://symfony.com/doc/current/quick_tour/the_big_picture.html) .

Además, hemos decidido que esta guía sea lo más corta posible, para que se a muy fácil leerla y consultarla las veces que necesites. Así que no vamos a repetir las explicaciones que puedes encontrar en la documentación de Symfony, como qué es la inyección de dependencias o cómo funcionan los controladores frontales. Nos centraremos exclusivamente en explicar cómo hacer lo que ya conoces.

1.3. La aplicación

Junto con esta guía encontrarás una aplicación de prueba desarrollada teniendo en cuenta todas estas recomendaciones. **La aplicación es tan sencilla como un blog**, ya que queremos que te fijes exclusivamente en lo que está relacionado con Symfony y te olvides de las particularidades de la propia aplicación.

Aunque la aplicación es sencilla no la vamos a desarrollar paso a paso en la guía. En su lugar, seleccionaremos algunos trozos de código a lo largo de los siguientes capítulos. En el último capítulo se explica cómo instalar la aplicación.

Capítulo 2.

Creando el proyecto

2.1. Instalando Symfony

Solamente existe una manera recomendada para instalar Symfony:

BUENA PRÁCTICA Utiliza Composer (https://getcomposer.org/) para instalar Symfony.

Composer es el gestor de dependencias que utilizan todas las aplicaciones PHP modernas. Gracias a Composer podrás añadir o quitar fácilmente dependencias en tus proyectos y también podrás actualizar sin esfuerzo las versiones de las librerías de terceros que utilizas en tus aplicaciones.

2.1.1. Gestionando dependencias con Composer

Antes de instalar Symfony, asegúrate de que tienes **Composer instalado globalmente**. Abre tu consola de comandos o terminal y ejecuta el siguiente comando:

```
$ composer --version
Composer version 1e27ff5e22df81e3cd0cd36e5fdd4a3c5a031f4a 2014-08-11 15:46:48
```

En tu caso verás un identificador de versión diferente. No te preocupes porque como Composer se actualiza continuamente, su versión no importa siempre que sea reciente.

2.1.2. Instalando Composer globalmente

Si todavía no has instalado Composer globalmente en tu ordenador, ejecuta los siguientes comandos en tu sistema operativo Linux o Mac OS X (no te asustes, pero el segundo comando te pedirá tu contraseña de usuario):

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

NOTA Dependiendo de la distribución de Linux que utilices es posible que debas ejecutar el comando su en vez de sudo.

Si tu sistema operativo es Windows, descarga el instalador de Composer (https://getcomposer.org/download/) desde su sitio web oficial, ejecútalo y sigue los pasos que se indican.

2.2. Creando la aplicación del blog

Ahora que ya tenemos todo listo, ya podemos crear un nuevo proyecto Symfony. Para ello, abre tu consola de comandos, entra en un directorio donde tengas permiso para crear archivos y directorios (por ejemplo, `proyectos/`) y ejecuta lo siguiente:

```
$ cd proyectos/
$ composer create-project symfony/framework-standard-edition blog/
```

Este comando creará un nuevo directorio llamado `blog/` que contiene una aplicación Symfony vacía basada en la versión más reciente disponible del framework.

2.2.1. Comprobando la instalación de Symfony

Cuando termine la instalación de Symfony, entra en el directorio `blog/` y ejecuta este comando para comprobar que Symfony se ha instlado correctamente:

```
$ cd blog/
$ php app/console --version

Symfony version 2.6.* - app/dev/debug
```

Si ves la versión instalada de Symfony, todo funcionó correctamente. Si no, ejecuta el siguiente *script* para detectar qué está fallando en tu ordenador y que impide ejecutar aplicaciones Symfony:

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

Dependiendo del estado de tu ordenador, el resultado del *script* anterior puede ser una o dos listas. La primera lista es la de los requisitos técnicos obligatorios que tu sistema no cumple. La segunda lista muestra los requisitios opcionales que tampoco cumples, pero que no te impiden ejecutar aplicaciones Symfony (so simplemente recomendaciones para que Symfony se ejecute mejor):

```
Symfony2 Requirements Checker
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

> PHP is using the following php.ini file:
  /usr/local/zend/etc/php.ini

> Checking Symfony requirements:
  .....E.......................W.....

[ERROR]
Your system is not ready to run Symfony2 projects
```

```
Fix the following mandatory requirements
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 * date.timezone setting must be set
   > Set the "date.timezone" setting in php.ini* (like Europe/Paris).

Optional recommendations to improve your setup
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 * short_open_tag should be disabled in php.ini
   > Set short_open_tag to off in php.ini*.
```

TRUCO Todas las versiones de Symfony están firmadas digitalmente por motivos de seguridad. Si quieres verificar la integridad de la versión de Symfony que has instalado, consulta el repositorio público de firmas digitales (https://github.com/sensiolabs/checksums) y sigue los pasos que se explican en este artículo (http://fabien.potencier.org/article/73/signing-project-releases) para poder verificarlas.

2.3. Estructurando la aplicación

Una vez creada la aplicación, entra en el directorio `blog/` y verás la siguiente jerarquía de archivos y directorios:

```
blog/
├─ app/
│   ├─ console
│   ├─ cache/
│   ├─ config/
│   ├─ logs/
│   └─ Resources/
├─ src/
│   └─ AppBundle/
├─ vendor/
└─ web/
```

Symfony propone eta jerarquía de archivos y directorios para estructurar tus aplicaciones. El propósito de cada archivo/directorio es el siguiente:

- `app/cache/`, almacena todos lso archivos de caché generados por la aplicación.
- `app/config/`, almacena toda la información definida para cada entorno de ejecución.
- `app/logs/`, almacena todos los archivos de log donde se guardan los mensajes de log generados por la aplicación.
- `app/Resources/`, almacena todas las plantillas de la aplicación y todos los archivos de traducciones.
- `src/AppBundle/`, almacena todo el código específico de Symfony (controladores, rutas, etc.) y todo tu código propio (clases de tu lógica de negocio, entidades de Doctrine, etc.)

- `vendor/`, este es el directorio donde Composer instala las depdendencias de la aplicación y por tanto nunca deberías tocar nada en este directorio.
- `web/`, almacena los controladores frontales y todos los *assets* relacionados con el *frontend*, tales como archivos CSS, archivos JavaScript y las imágenes.

2.4. Los bundles de la aplicación

Cuando se publicó Symfony 2.0 por primera vez, la respuesta natural de los programadores fue seguir estructurando sus aplicaciones como si se tratara de la antigua versión symfony 1.x. Así que muchas aplicaciones Symfony acabaron divididas en *módulos lógicos*, similares a los de symfony 1.x.

Por eso hoy en día es habitual encontrarse con aplicaciones Symfony2 que definen *bundles* como los siguientes: `UserBundle`, `ProductBundle`, `InvoiceBundle`, etc.

No obstante, los *bundles* están pensados para que sean elementos de software que se pueden reutilizar de manera autónoma. En otras palabras, si tienes por ejemplo un *bundle* llamado `UserBundle` pero que no puedes reutilizarlo tal y como está en otras aplicaciones Symfony2, entonces ese *bundle* está mal hecho. Lo mismo que si tu *bundle* `InvoiceBundle` depende de otro *bundle* como `ProductBundle`. Se trata de un error y estos dos elementos no deberían ser *bundles* independientes.

BUENA PRÁCTICA Crea solamente un *bundle* llamado `AppBundle` en tu aplicación.

Al tener un único *bundle* llamdo `AppBundle`, el código de tus aplicaciones será mucho más fácil de escribir y de leer. Además, a partir de la versión 2.6, la documentación oficial de Symfony siempre mostrará sus ejemplos con este *bundle* `AppBundle`.

NOTA No es necesario añadir ningún prefijo al nombre `AppBundle` para indicar el *vendor*, como por ejemplo el nombre de tu empresa o proyecto: `AcmeAppBundle`. El motivo es que este *bundle*, por su propia naturaleza, nunca va a ser reutilizado en otras aplicaciones y nunca va a ser publicado como *bundle* de terceros.

Con todo esto, la estructura de directorios recomendada para una aplicación Symfony que siga las buenas prácticas oficiales debería ser la siguiente:

```
blog/
├─ app/
│  ├─ console
│  ├─ cache/
│  ├─ config/
│  └─ logs/
│  └─ Resources/
├─ src/
│  └─ AppBundle/
├─ vendor/
└─ web/
   ├─ app.php
   └─ app_dev.php
```

TRUCO Si utilizas Symfony 2.6 o superior, este *bundle* `AppBundle` ya está generado, por lo que no tienes que hacer nada más. Si utilizas una versión anterior de Symfony, puedes generar el *bundle* ejecutando el siguiente comando:

```
$ php app/console generate:bundle --namespace=AppBundle --dir=src
--format=annotation --no-interaction
```

2.5. Utilizando otra estructura de directorios

Si tu proyecto o la infraestructura que utilizas requieren cambios en la jerarquía de directorios por defecto, puedes leer este artículo (http://symfony.com/doc/current/cookbook/configuration/override_dir_structure.html) para saber cómo redefinir la localización de los directorios `cahce/`, `logs/` y `web/`.

Por otra parte, la futura versión Symfony 3 utilizará una estructura de directorios ligeramente diferente:

```
blog-symfony3/
├─ app/
│  ├─ config/
│  └─ Resources/
├─ bin/
│  └─ console
├─ src/
├─ var/
│  ├─ cache/
│  └─ logs/
├─ vendor/
└─ web/
```

Aunque los cambios son muy sutiles, te aconsejamos que por el momento continúes con la estructura de directorios por defecto de Symfony2.

Esta página se ha dejado vacía a propósito

Capítulo 3.

Configuración

La configuración de una aplicación normalmente está relacionada con muchas partes diferentes (infraestructura tecnológica, seguridad, etc.) y con varios entornos de ejecución (desarrollo, producción, etc.) Por eso Symfony recomienda dividir la configuración de la aplicación en tres partes diferenciadas.

3.1. Configuración relacionada con la infraestructura

BUENA PRÁCTICA Define las opciones de configuración relacionadas con la infraestructura tecnológica en el archivo `app/config/parameters.yml`.

El archivo `parameters.yml` creado por defecto por Symfony sigue esta recomendación y define las opciones relacionadas con la base de datos y el servidor de correo:

```
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     127.0.0.1
    database_port:     ~
    database_name:     symfony
    database_user:     root
    database_password: ~

    mailer_transport:  smtp
    mailer_host:       127.0.0.1
    mailer_user:       ~
    mailer_password:   ~

    # ...
```

Estas opciones no se definen en el archivo `app/config/config.yml` porque no tienen nada que ver con el comportamiento de la aplicación. En otras palabras, a la aplicación le da exactamente igual dónde

se encuentra la base de datos o cómo se accede a ella. Lo único que le importa es que haya una base de datos preparada.

3.1.1. Parámetros canónicos

BUENA PRÁCTICA Define todos los parámetros de la aplicación en el archivo `app/config/parameters.dist.yml`.

Desde la versión 2.3 Symfony incluye un archivo de configuración llamado `parameters.dist.yml`, que almacena todos los parámetros de configuración que deben estar definidos en la aplicación para que ésta funcione correcamente. Estos parámetros se llaman **parámetros canónicos de la aplicación** (*canonical parameters* en inglés).

Cada vez que definas un nuevo parámetro de configuración para tu aplicación, no olvides añadirlo también a este archivo `parameters.dist.yml` y subir después los cambios al respositorio de código.

De esta manera, cada vez que un miembro del equipo de desarrollo actualice el proyecto o lo instale en producción, Symfony comprobará si el número de parámetros del archivo `parameters.yml` coincide con los parámetros definidos en `parameters.yml.dist`. Si existe cualquier diferencia, Symfony te preguntará qué valor quieres asignar a los parámetros que te faltan por definir y los añadirá al archivo `parameters.yml`. Así la aplicación nunca fallará porque le falte un parámetro de configuración.

3.2. Configuración relacionada con la aplicación

BUENA PRÁCTICA Define las opciones de configuración relacionadas con la aplicación en el archivo `app/config/config.yml`.

El archivo `config.yml` define las opciones que se utilizan para modificar el comportamiento de la aplicación, tales como el remitente de los emails de notificación, los *feature toggles* (http://en.wikipedia.org/wiki/Feature_toggle) , etc. Definir estos valores en el archivo `parameters.yml` es posible, pero añadiría una capa adicional de configuración totalmente innecesaria, ya que normalmente no necesitas cambiar estos valores en función del servidor en el que se encuentre la aplicación.

Las opciones de configuración definidas en el archivo `config.yml` normalmente varían en función del entorno de ejecución (http://librosweb.es/symfony_2_x/capitulo_4/entornos.html) . Por eso Symfony define dos archivos distintos llamados `app/config/config_dev.yml` y `app/config/config_prod.yml` para que puedas definir diferentes valores para cada entorno.

3.2.1. Constantes u opciones de configuración

Uno de los errores más habituales al definir la configuración de una aplicación consiste en añadir opciones para valores que nunca van a cambiar, como por ejemplo el número de elementos mostrados en una paginación.

BUENA PRÁCTICA Utiliza constantes en vez de opciones de configuración para los valores que casi nunca varían.

La forma tradicional en la que se dfine la configuración de las aplicaciones ha hecho que muchas aplicaciones Symfony2 incluyan opciones como la siguiente, que controla cuantos artículos se muestran en la portada del blog:

```
# app/config/config.yml
parameters:
    homepage.num_items: 10
```

Si te preguntas cuándo fue la última vez que cambiaste el valor de una opción de este tipo, seguramente la respuesta será **nunca jamás**. Crear una opción para un valor que apenas cambia es absurdo. En estos casos es mucho mejor definir una constante en la clase apropiada. Así por ejemplo se puede definir la constante `NUM_ITEMS` en la clase que define la entidad `Post`:

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

class Post
{
    const NUM_ITEMS = 10;

    // ...
}
```

La principal ventaja de las constantes es que puedes utilizar sus valores en cualquier punto de la aplicación. Cuando defines parámetros de configuración, sólo puedes utilizarlos en los lugares en los que puedes acceder al contenedor de Symfony.

Así por ejemplo puedes acceder al valor de cualquier constante desde todas las plantillas Twig gracias a la función `constant()`:

```
<p>
    Mostramos los {{ constant('NUM_ITEMS', post) }} artículos más recientes.
</p>
```

También puedes acceder a estos valores desde las clases de las entidades y los repositorios de Doctrine, que son lugares en los que habitualmente se necesitan estos valores y donde no es posible acceder al contenedor de Symfony:

```
namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;
use AppBundle\Entity\Post;

class PostRepository extends EntityRepository
{
    public function findLatest($limit = Post::NUM_ITEMS)
```

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

La única desventaja importante de usar constantes como opciones de configuración es que no puedes redefinirlas fácilmente en los tests de la aplicación.

3.3. No utilices la configuración semántica

BUENA PRÁCTICA No definas en tus *bundles* una configuración semántica para el contenedor de inyección de dependencias.

Como se explica en el artículo *How to Expose a semantic Configuration for a Bundle* (http://symfony.com/doc/current/cookbook/bundles/extension.html) , los *bundles* de Symfony disponen de dos opciones para gestionar sus propias opciones de configuración: la configuración *normal* definida en el archivo `services.yml` y la configuración *semática* mediante una clase especial de tipo `*Extension`.

Aunque la configuración semántica es mucho más avanzada y ofrece características realmente interesantes, como la validación de las opciones de configuración, el trabajo necesario para definir esa configuración no merece la pena para los *bundles* internos de la aplicación y que por tanto, no se van a reutilizar como *bundles* de terceros.

3.4. Define las opciones de configuración sensibles fuera de Symfony

Al trabajar con opciones de configuración sensibles, como por ejemplo las credenciales para acceder a la base de datos, la recomendación consiste en almacenarlas fuera de la aplicación Symfony y acceder a ellas mediante las variables de entorno. Para saber cómo hacerlo, consulta el artículo *How to Set external Parameters in the Service Container* (http://symfony.com/doc/current/cookbook/configuration/external_parameters.html) .

Capítulo 4.

Organizando la lógica de negocio

La **lógica de negocio** o *business logic* consiste en todo el código que escribes para tu aplicación y que no está relacionado con el propio framework (controladores, rutas, etc.) Las clases del dominio, las entidades de Doctrine y las clases PHP normales y corrientes son buenos ejemplos de lo que es la lógica de negocio de una aplicación Symfony.

En la mayoría de tus proyectos, deberías almacenar toda tu lógica de negocio dentro del *bundle* `AppBundle`. Para organizar mejor tu código puedes crear todos los directorios que necesites:

```
symfoy2-project/
├─ app/
├─ src/
│  └─ AppBundle/
│     └─ Utils/
│        └─ MyClass.php
├─ vendor/
└─ web/
```

4.1. Almacenando clases fuera del *bundle*

No existe ninguna limitación técnica para guardar tu lógica de negocio fuera de cualquier *bundle*. Así que si lo prefieres, puedes crear tu propio *namespace* dentro del directorio `src/` y organizar el código de esa manera:

```
symfoy2-project/
├─ app/
├─ src/
│  ├─ Acme/
│  │  └─ Utils/
│  │     └─ MyClass.php
│  └─ AppBundle/
```

```
├─ vendor/
└─ web/
```

TRUCO La recomendación de utilizar el *bundle* `AppBundle` es para que todo sea más sencillo de gestionar. Si tienes la experiencia suficiente como para decidir lo que tiene que colocarse dentro o fuera del *bundle*, entonces deberías dejarte guiar por tu propia experiencia.

4.2. Configurando los servicios

La aplicación del blog necesita una utilidad que transforme los títulos de los artículos (ejemplo *Hola Mundo*) es un *slug* (ejemplo *hola-mundo*). Los *slug* son las cadenas de texto que se utilizan como parte de la URL de los artículos para evitar los caracteres problemáticos (espacios en blanco, acentos, etc.)

Así que se crea una nueva clase llamada `Slugger` dentro del directorio `src/AppBundle/Utils/` y se define el siguiente método `slugify()`:

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

class Slugger
{
    static function slugify($string)
    {
        return preg_replace(
            '/[^a-z0-9]/', '-', strtolower(trim(strip_tags($string)))
        );
    }
}
```

Para poder utilizar esta clase en tu aplicación, define un nuevo servicio de la siguiente manera:

```
# app/config/services.yml
services:
    # el nombre de los servicios debería ser muy corto
    slugger:
        class: AppBundle\Utils\Slugger
```

Normalmente los programadores Symfony utilizan nombres de servicios muy largos, copuestos por el nombre de la clase y su clase, *bundle* o *namespace* para evitar colisiones. Así que en este caso el servicio podría haberse llamado por ejemplo `app.utils.slugger`. Pero cuando se utilizan nombres cortos para los servicios, el código se simplifica y es más fácil de leer.

BUENA PRÁCTICA El nombre de los servicios debería ser tan corto como sea posible, idealmente una sola palabra corta.

Ahora ya puedes utilizar el servicio `slugger` en cualquier controlador, como por ejemplo `AdminController`:

```
public function createAction(Request $request)
{
    // ...

    if ($form->isSubmitted() && $form->isValid()) {
        $slug = $this->get('slugger')->slugify($post->getTitle()));
        $post->setSlug($slug);

        // ...
    }
}
```

4.2.1. El formato de configuración YAML

En la sección anterior, se utilizó YAML para definir el servicio.

BUENA PRÁCTICA Utiliza YAML para definir tus propios servicios.

Sabemos que esta recomendación es muy controvertida. Según nuestra propia experiencia, el uso de YAML y XML está bastante repartido, con una ligera ventaja para YAML. En cualquier caso, como los dos formatos tienen el mismo rendimiento, al final la decisión de cuál utilizar es una cuestión de gusto personal.

La recomendación de utilizar YAML se debe a que los nuevos programadores Symfony pueden así definir los servicios más fácilmente y de manera muy concisa. Pero en tus aplicaciones puedes elegir el formato que prefieras.

4.2.2. No definas parámetros para las clases de los servicios

Probablemente te habrás dado cuenta de que antes no hemos creado un parámetro de configuración para definir la clase del servicio:

```
# app/config/services.yml

# creando un parámetro para definir la clase de un servicio
# NO lo hagas en tus aplicaciones
parameters:
    slugger.class: AppBundle\Utils\Slugger

services:
    slugger:
        class: "%slugger.class%"
```

Esta práctica es muy pesada y además, totalmente innecesaria para los servicios propios de tus aplicaciones.

BUENA PRÁCTICA No definas parámetros de configuración para las clases de tus servicios.

Esta mala práctica tiene su origen de nuevo en los *bundles* de terceros. Si desarrollas un *bundle* para compartirlo públicamente, entonces probablmente sí que tienes que definir parámetros para las clases de tus servicios. Pero si estás definiendo servicios internos que sólo utilizará tu aplicación, entonces no hay necesidad de hacer que sus clases sean configurables.

4.3. Utilizando una capa de persistencia

Symfony es un framework HTTP que simplemente se encarga de generar respuestas HTTP a partir de peticiones HTTP. Por eso Symfony no incluye ninguna utilidad para comunicarse con las capas de persistencia, como por ejemplo bases de datos y APIs externas. Así que eres libre de escoger la librería o estrategia que prefieras para solventar esta carencia.

En la práctica la mayoría de aplicaciones Symfony hacen uso del proyecto Doctrine (http://www.doctrine-project.org/) para definir su modelo mediante las entidades y los resposito-rios. Al igual que con el resto de la lógica de negocio, se recomienda almacenar las entidades de Doctrine dentro del *bundle* `AppBundle`.

Las tres entidades definidas en la aplicación del blog son un buen ejemplo de cómo organizar estas clases:

```
symfony2-project/
├─ ...
└─ src/
   └─ AppBundle/
      └─ Entity/
         ├─ Comment.php
         ├─ Post.php
         └─ User.php
```

TRUCO Si tienes la experiencia suficiente, por supuesto puedes almacenar las entidades de Doctrine en cualquier otro directorio dentro de `src/`.

4.3.1. La información de mapeo de Doctrine

Las entidades de Doctrine son objetos PHP normales y corrientes cuya información se almacena en la base de datos. La única información que tiene Doctrine sobre tus entidades es la información de *mapeo* (*mapping* en inglés) que definas para ellas. Doctrine soporta cuatro formatos para definir esta información: YAML, XML, PHP y anotaciones.

BUENA PRÁCTICA Utiliza anotaciones para definir la información de mapeo de las entidades de Doctrine.

Las anotaciones son con mucha diferencia el formato más cómodo y la manera más ágil de definir la información de mapeo:

```
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;
```

```
/**
 * @ORM\Entity
 */
class Post
{
    const NUM_ITEMS = 10;

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

    /**
     * @ORM\Column(type="string")
     */
    private $title;

    /**
     * @ORM\Column(type="string")
     */
    private $slug;

    /**
     * @ORM\Column(type="text")
     */
    private $content;

    /**
     * @ORM\Column(type="string")
     */
    private $authorEmail;

    /**
     * @ORM\Column(type="datetime")
     */
    private $publishedAt;

    /**
     * @ORM\OneToMany(
     *      targetEntity="Comment",
     *      mappedBy="post",
     *      orphanRemoval=true
     * )
     * @ORM\OrderBy({"publishedAt" = "ASC"})
     */
    private $comments;
```

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

    // getters and setters ...
}
```

Como todos los formatos tienen el mismo rendimiento, de nuevo esta decisión es una cuestión de gusto personal y de elegir el formato con el que tú y tu equipo os encontréis más a gusto.

4.3.2. Datos de prueba

Symfony no incluye por defecto el soporte para crear datos de prueba, por lo que deberías ejecutar el siguiente comando para instalar el *bundle* para crear datos de prueba de Doctrine:

```
$ composer require "doctrine/doctrine-fixtures-bundle":"~2"
```

Después, activa el nuevo *bundle* en `AppKernel.php`, pero hazlo solamente en el entorno de desarrollo (`dev`) y pruebas (`test`), ya que en producción no se necesita y así evitamos que penalice el rendimiento:

```
use Symfony\Component\HttpKernel\Kernel;

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

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

        return $bundles;
    }

    // ...
}
```

Por simplicidad, es recomendable crear una única clase para definir los datos de prueba (http://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html#writing-simple-fixtures) . Si esta clase crece mucho, divide sus contenidos en varias clases más pequeñas.

Suponiendo que se haya definido al menos una clase con archivos de prueba y que el acceso a la base de datos está bien configurado, puedes cargar todos los datos ejecutando el siguiente comando:

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

Careful, database will be purged. Do you want to continue Y/N ? Y
  > purging database
  > loading AppBundle\DataFixtures\ORM\LoadFixtures
```

4.4. Estándares de código

El código fuente de Symfony2 sigue las recomendaciones de los estándares PSR-1 (http://www.php-fig.org/psr/psr-1/) y PSR-2 (http://www.php-fig.org/psr/psr-2/) definidos por la comunidad PHP. Para más información, puedes leer el artículo sobre los estándares de código de Smyfony (http://symfony.com/doc/current/contributing/code/standards.html) . También puedes utilizar la herramienta de consola PHP-CS-Fixer (https://github.com/fabpot/PHP-CS-Fixer) para reformatear todo el código fuente de tu aplicación en unos pocos segundos y con la garantía de no romperlo con la actualización.

Esta página se ha dejado vacía a propósito

Capítulo 5.

Controladores

Symfony sigue la filosofía de crear controladores muy pequeños y clases de modelo muy grandes (en inglés, *thin controllers and fat models*). Por eso los controladores de Symfony se comportan como una fina capa de código que une y coordina las diferentes partes de la aplicación.

La regla de oro de los controladore se resume en **5-10-20**, ya que los controladores deberían definir **5** variables o menos, contener **10** acciones o menos e incluir **20** líneas de código o menos en cada acción. Aunque puede haber excepciones puntuales a esta regla, te puede servir como orientación para saber cuándo tienes que refactorizar tus controladores para convertir partes de su código en servicios.

BUENA PRÁCTICA Haz que tus controladores extiendan de la clase `Controller` base proporcionada por el *bundle* `FrameworkBundle` y usa anotaciones para definir el enrutamiento, la caché y la seguridad siempre que sea posible.

Cuando *acoplas* los controladores al framework que estás utilizando, puedes aprovechar todas sus funcionalidades y eso aumenta tu productividad.

Como los controladores no son más que una capa muy fina que apenas contiene unas pocas líneas de código para unir las diferentes partes de la aplicación, dedicar decenas de horas a desacoplar ese código respecto al framework es un trabajo enorme que difícilmente te compensará a la larga.

Además, al usar anotaciones para el enrutamiento, la caché y la seguridad, se simplifica enormemente la configuración de la aplicación. Ya no tendrás que rebuscar entre decenas de archivos de configuración creados con diferentes formatos (YAML, XML, PHP). Ahora toda la configuración relevante está justo donde la necesitas y utiliza un único formato.

En resumen, la recomendación es **desacoplar totalmente tu lógica de negocio** respecto del framework, pero al mismo tiempo, **acoplar totalmente los controladores y el enrutamiento** al framework, para aprovechar todas sus posibilidades de la manera más ágil posible.

5.1. Configurando el enrutamiento

Para cargar todas las rutas definidas como anotaciones en los controladores del *bundle* `AppBundle`, añade la siguiente configuración en el archivo principal de enrutamiento:

```
# app/config/routing.yml
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
```

Esta configuración hace que se carguen todas las anotaciones de cualquier controlador definido dentro del directorio `src/AppBundle/Controller/` y de todos sus subdirectorios. Así que si en tu aplicación has definido muchos controladores, puede ser una buena idea organizarlos en subdirectorios:

```
<tu-proyecto>/
├─ ...
└─ src/
   └─ AppBundle/
      ├─ ...
      └─ Controller/
         ├─ DefaultController.php
         ├─ ...
         ├─ Api/
         │  ├─ ...
         │  └─ ...
         └─ Backend/
            ├─ ...
            └─ ...
```

5.2. Configurando las plantillas

BUENA PRÁCTICA No utilices la anotación `@Template()` para definir la plantilla utilizada por el controlador.

Aunque la anotación `@Template` es útil, se trata de la anotación que más *magia* implica. Y ese es el motivo por el que se recomienda no utilizarla.

La mayoría de controladores utilizan la anotación `@Template` sin parámetros, lo que hace más difícil a los programadores saber exactamente la plantilla que se está renderizando. Además, para los programadores que empiezan con Symfony este comportamiento resulta confuso, ya que según se explica en la documentación, los controladores deberían devolver como resultado una respuesta en forma de objeto `Response` (a menos que utilices la capa de la vista y devuelvas una plantilla renderizada).

Por último, la anotación `@Template` utiliza la clase `TemplateListener` para *escuchar* el evento `kernel.view` lanzado pr el framework. Este *listener* introduce un impacto no despreciable en el rendimiento de la aplicación. En la aplicación de blog, renderizar la portada cuesta menos de 5 milisegundos cuando se utiliza el método `$this->render()` en el controlador y más de 26 milisegundos cuando se utiliza la anotación `@Template`.

5.3. Cómo deberían ser los controladores Symfony

Considerando todo lo anterior, a continuación se muestra un ejemplo de cómo debería ser el código de los controladores de una aplicación Symfony típica:

```
namespace AppBundle\Controller;

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

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        $em = $this->getDoctrine()->getManager();
        $posts = $em->getRepository('App:Post')->findLatest();

        return $this->render('default/index.html.twig', array(
            'posts' => $posts
        ));
    }
}
```

5.4. Utilizando los ParamConverter

Si en tu aplicación utilizas Doctrine, entonces puedes utilizar si quieres lo que se llama ParamConverter (http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html) para realizar búsquedas en la base de datos automáticamente y pasar el resultado al controlador.

BUENA PRÁCTICA Utiliza los ParamConverter para buscar las entidades Doctrine automáticamente siempre que la búsqueda sea sencilla.

Ejemplo

```
/**
 * @Route("/{id}", name="admin_post_show")
 */
public function showAction(Post $post)
{
    $deleteForm = $this->createDeleteForm($post);

    return $this->render('admin/post/show.html.twig', array(
        'post'      => $post,
        'delete_form' => $deleteForm->createView(),
    ));
}
```

Lo habitual en este tipo de controladores consiste en pasar como parámetro a `showAction()` el `$id` del artículo que se quiere leer. Sin embargo, al utilizar como parámetro la variable `$post` con el tipo de dato `Post` (en inglés a esta técnica se le llama *type hinting*), Symfony aplica un *ParamConverter* para buscar automáticamente el objeto cuyo atributo `id` coincida con el valor `$id` obtenido mediante la ruta. Si no se encuentra ningún objeto, se muestra automáticamente una página de error `404`.

5.4.1. Realizando búsquedas más avanzadas

El ejemplo anterior funciona sin tener que realizar ninguna configuración adicional porque `{id}` coincide exactamente con el nombre de una entidad. Cuando esto no sucede, o si quieres hacer búsquedas más complejas, puede resultar más sencillo realizar la búsqueda de Doctrine a mano. Esto es por ejemplo lo que sucede en la clase `CommentController` de la aplicación:

```
/**
 * @Route("/comment/{postSlug}/new", name = "comment_new")
 */
public function newAction(Request $request, $postSlug)
{
    $post = $this->getDoctrine()
        ->getRepository('AppBundle:Post')
        ->findOneBy(array('slug' => $slug));

    if (!$post) {
        throw $this->createNotFoundException();
    }

    // ...
}
```

Por supuesto también podrías utilizar la configuración avanzada de `@ParamConverter` porque es bastante flexible:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @Route("/comment/{postSlug}/new", name = "comment_new")
 * @ParamConverter("post", options={"mapping": {"postSlug": "slug"}})
 */
public function newAction(Request $request, Post $post)
{
    // ...
}
```

La conclusión es que los *ParamConverter* son geniales para casos sencillos, pero no deberías olvidar que hacer las consultas Doctrine a mano también es bastante sencillo.

5.5. Ejecutando código antes y después del controlador

Si quieres ejecutar cierto código antes o después de la ejecución de los controladores de la aplicación, utiliza el componente *EventDispatcher* tal y como se explica en el artículo *How to Setup before and after Filters* (http://symfony.com/doc/current/cookbook/event_dispatcher/before_after_filters.html) .

Esta página se ha dejado vacía a propósito

Capítulo 6.

Plantillas

Cuando se publicó PHP por primera vez hace más de 20 años, a los programadores les encantó su sencillez y lo fácil que era mezclar código PHP con código HTML. Sin embargo, con el paso del tiempo an surgido lenguajes de plantillas como Twig (http://twig.sensiolabs.org/) que son mucho mejores para crear las plantillas de las aplicaciones.

BUENA PRÁCTICA Utiliza solamente Twig para crear tus plantillas.

En general, las plantillas PHP son mucho más verbosas que las plantillas Twig porque no tienen soporte nativo para muchas de las funcionalidades que necesitan las plantillas modernas, tales como la herencia, el *escapado* automático de información y los filtros.

Twig es el formato por defecto de las plantillas Symfony y es el formato distinto a PHP con la mayor comunidad de usuarios, ya que se utiliza en proyectos tan importantes como Drupal 8.

Por último, Twig es el único formato que tienen garantizado el soporte en la futura versión Symfony 3.0. Tanto es así, que es muy posible que desaparezca el soporte de plantillas PHP.

6.1. Organizando las plantillas

BUENA PRÁCTICA Almacena todas las plantillas de la aplicación en el directorio `app/Resources/views/`.

La mayoría de programadores Symfony almacena las plantillas de la aplicación en los directorios `Resources/views/` de cada *bundle*. Después utilizan el *nombre lógico* de la plantilla para referirse a ella (ejemplo `AcmeDemoBundle:Default:index.html.twig`).

Aunque esta práctica es correcta para los *bundles* de terceros, en el caso de las plantillas de la aplicación es mucho más cómodo almacenarlas en el directorio `app/Resources/views/`. El primer motivo es que se simplifica de manera radical el *nombre lógico* de las plantillas:

Plantillas almacenadas en *bundles*	Plantillas almacenadas en app/
AcmeDemoBunde:Default:index.html.twig	default/index.html.twig
::layout.html.twig	layout.html.twig
AcmeDemoBundle::index.html.twig	index.html.twig
AcmeDemoBundle:Default:subdir/index.html.twig	default/subdir/index.html.twig
AcmeDemoBundle:Default/subdir:index.html.twig	default/subdir/index.html.twig

Otra de las ventajas de centralizar todas las plantillas es que se simplifica mucho el trabajo de tu equipo *frontend* y de tus diseñadores. Ya no es necesario buscar las plantillas entre decenas de directorios porque todas se encuentran en el mismo lugar.

6.2. Extensiones Twig

BUENA PRÁCTICA Define tus extensiones Twig en el directorio `AppBundle/Twig/` y configúralas en el archivo `app/config/services.yml`.

A la aplicación del blog le vendría muy bien un filtro de Twig llamado `md2html` que transforme el contenido Markdown en contenido HTML.

Antes de crear este filtro, añade como dependencia de tu proyecto la excelente librería Parsedown (http://parsedown.org/) que transforma el código Markdown en HTML. Para ello, ejecuta el siguiente comando en la consola:

```
$ composer require erusev/parsedown
```

Después, define un nuevo servicio llamado `markdown` para que lo utilice la extensión de Twig. En este caso la definición del servicio es tn simple como indicar la ruta a la clase PHP asociada:

```
# app/config/services.yml
services:
    # ...
    markdown:
        class: AppBundle\Utils\Markdown
```

Esta clase `Markdown` define un único método que transforma el contenido Markdown pasado como argumento en contenido HTML:

```
namespace AppBundle\Utils;

class Markdown
{
    private $parser;

    public function __construct()
```

```
    {
        $this->parser = new \Parsedown();
    }

    public function toHtml($text)
    {
        $html = $this->parser->text($text);

        return $html;
    }
}
```

Ahora ya puedes crear la extensión Twig para definir un nuevo filtro llamado `md2html` mediante la clase `Twig_SimpleFilter`. No olvides inyectar el servicio `markdown` en el constructor de la extensión Twig:

```
namespace AppBundle\Twig;

use AppBundle\Utils\Markdown;

class AppExtension extends \Twig_Extension
{
    private $parser;

    public function __construct(Markdown $parser)
    {
        $this->parser = $parser;
    }

    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter(
                'md2html',
                array($this, 'markdownToHtml'),
                array('is_safe' => array('html'))
            ),
        );
    }

    public function markdownToHtml($content)
    {
        return $this->parser->toHtml($content);
    }

    public function getName()
    {
        return 'app_extension';
    }
}
```

Para poder utilizar la extensión de Twig en la aplicación, define un nuevo servicio y etiquétalo con la etiqueta `twig.extension` (el nombre de este servicio es irrelevante porque no lo vas a usar directamente en la aplicación):

```
# app/config/services.yml
services:
    app.twig.app_extension:
        class:     AppBundle\Twig\AppExtension
        arguments: ["@markdown"]
        tags:
            - { name: twig.extension }
```

Capítulo 7.

Formularios

Los formularios son una de las partes de Symfony que peor se utilizan, debido a su gran complejidad y su interminable lista de funcionalidades. En este capítulo se muestran algunas de las buenas prácticas que pueden hacerte aprovechar los formularios de una manera sencilla.

7.1. Creando los formularios

BUENA PRÁCTICA Define los formularios como clases PHP.

El componente *Form* permite crear los formularios directamente dentro de los controladores. Si el formulario es pequeño y no vas a reutilizarlo nunca, no es incorrecto definirlo dentro del propio controlador. Pero para formularios complejos que se reutilizan en muchas partes de la aplicación, es recomendable definir cada formulario en su propia clase PHP:

```
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('summary', 'textarea')
            ->add('content', 'textarea')
            ->add('authorEmail', 'email')
            ->add('publishedAt', 'datetime')
        ;
    }
```

```
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Post'
        ));
    }

    public function getName()
    {
        return 'post';
    }
}
```

Para utilizar esta clase en un controlador, emplea el atajo `createForm` e instancia una nueva clase de tipo formulario:

```
use AppBundle\Form\PostType;
// ...

public function newAction(Request $request)
{
    $post = new Post();
    $form = $this->createForm(new PostType(), $post);

    // ...
}
```

7.1.1. Registrando formularios como servicios

Symfony permite registrar los formularios como servicios (http://symfony.com/doc/current/cookbook/form/create_custom_field_type.html#creating-your-field-type-as-a-service) , pero no lo recomendamos a menos que vayas a reutilizar ese tipo de formulario en muchos sitios diferentes o vayas a embeberlo dentro de otros formularios mediante el tipo "collection" (http://symfony.com/doc/current/reference/forms/types/collection.html) .

En la mayoría de los casos, los formularios solamente se utilizan para crear o modificar información, por lo que registrarlos como servicios es totalmente innecesario. Además, esto hace que sea más difícil saber exactamente qué clase de formulario se está utilizando dentro del controlador.

7.2. Configurando los botones del formulario

Las clases de formularios deberían ser agnósticas respecto a dónde se van a utilizar. De esta manera los formularios se podrán reutilizar más fácilmente.

BUENA PRÁCTICA Añade los botones de los formularios en las plantillas, no en las clases de formulario o en los controladores.

Desde la versión 2.5 de Symfony es posible añadir botones como campos de tus propios formularios. La ventaja es que así se simplifica el código de la plantilla que muestra el formulario. La desventaja es que al añadir los botones al propio formulario, estás limitando su funcionalidad:

```
class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('save', 'submit', array('label' => 'Create Post'))
        ;
    }

    // ...
}
```

Originalmente este formulario se diseño para crear artículos del blog y por eso incluye el botón *Create Post*. Sin embargo, este formulario podría reutilizarse también para la acción de modificar los artículos. En este caso el texto del botón sería incorrecto, ya que no estás *creando* artículos sino *modificándolos*. Para evitar estos problemas, algunos programadores prefieren añadir los botones directamente en el controlador:

```
namespace AppBundle\Controller\Admin;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use AppBundle\Entity\Post;
use AppBundle\Form\PostType;

class PostController extends Controller
{
    // ...

    public function newAction(Request $request)
    {
        $post = new Post();
        $form = $this->createForm(new PostType(), $post);
        $form->add('submit', 'submit', array(
            'label' => 'Create',
            'attr'  => array('class' => 'btn btn-default pull-right')
        ));

        // ...
    }
}
```

Esta solución también es un error, ya que estás mezclando el código de las plantillas (etiquetas, clases CSS, etc.) con el código PHP puro del controlador. Separar las diferentes capas de la aplicación es una buena práctica que siempre deberías seguir. Por eso es mejor colocar todo lo relacionado con la vista en las propias plantillas:

```
<form method="POST" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Create"
           class="btn btn-default pull-right" />
</form>
```

7.3. Renderizando el formulario

Symfony permite renderizar los formularios de muchas formas diferentes, desde renderizar todo el formulario a la vez con una instrucción hasta renderizar cada una de las partes que forman cada campo de formulario. La mejor manera de renderizar el formulario siempre depende de cuánto necesites personalizar los formularios.

Lo más sencillo, especialmente durante el desarrollo de la aplicación, es crear las etiquetas `<form>` y `</form>` a mano y después utilizar la función `form_widget()` para renderizar todos los campos del formulario:

```
<form method="POST" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
</form>
```

BUENA PRÁCTICA No utilices las funciones `form()`, `form_start()` o `form_end()` para renderizar las etiquetas de apertura y cierre del formulario.

Los programadores Symfony más experimentados se habrán dado cuenta de que en el ejemplo anterior no se utilizan las funciones `form_start()` o `form()` para renderizar la etiqueta `<form>` del formulario. El motivo es que, aunque las funciones son muy cómodas, no aportan tantas ventajas teniendo en cuenta que reducen ligeramente la legibilidad del formulario.

TRUCO La única excepción es el formulario para borrar artículos porque es tan simple que su único contenido es un botón y entonces sí que merece la pena utilizar estos atajos.

Si quieres controlar de manera más precisa cómo se renderizan los campos del formulario, no utilices la función `form_widget(form)` y en su lugar, renderiza cada campo de formulario individualmente. Consulta el artículo *How to Customize Form Rendering* (http://symfony.com/doc/current/cookbook/form/form_customization.html) para saber cómo hacerlo y para aprender a definir temas de formulario.

7.4. Procesando el envío de formularios

El procesamiento de los formularios Symfony en los controladores debería seguir la siguiente estructura:

```
public function newAction(Request $request)
{
    // crear el formulario ...

    $form->handleRequest($request);
```

```
    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($post);
        $em->flush();

        return $this->redirect($this->generateUrl(
            'admin_post_show',
            array('id' => $post->getId())
        ));
    }

    // renderizar la plantilla ...
}
```

En el código anterior es importante fijarse en dos cosas. Primero, es mejor utilizar una única acción tanto para mostrar el formulario como para procesarlo. Aunque podrías haber definido una acción `newAction` que solamente muestre el formulario y después otra acción `createAction` que sólo se encargue de procesarlo, las dos acciones serían prácticamente idénticas. Así que es mucho más fácil si la acción `newAction` se encarga de las dos tareas.

En segundo lugar, te aconsejamos que añadas el método `$form->isSubmitted()` dentro de la instrucción `if` para hacer el código más claro. Técnicamente no es necesario hacerlo, ya que el método `isValid()` ejecuta primero el método `isSubmitted()`. Pero si no lo añades, el flujo de ejecución del formulario queda un poco raro, ya que parece que siempre se está procesando el formulario, incluso en las peticiones `GET` que simplemente lo muestran.

Esta página se ha dejado vacía a propósito

Capítulo 8.

Internacionalización

La internacionalización y la localización se encargan de adaptar las aplicaciones y sus contenidos para la región o idioma específicos del usuario. En Symfony esta funcionalidad es opcional y por eso debes activarla antes de utilizarla. Para ello, descomenta la opción de configuración `translator` y define el idioma por defecto de la aplicación:

```
# app/config/config.yml
framework:
    # ...
    translator: { fallback: "%locale%" }

# app/config/parameters.yml
parameters:
    # ...
    locale:     en
```

8.1. Formato de los archivos de traducción

El componente *Translation* de Symfony soporta muchos formatos diferentes para crear los archivos de traducción: PHP, Qt, `.po`, `.mo`, JSON, CSV, INI, etc.

BUENA PRÁCTICA Utiliza el formato XLIFF para crear los archivos de traducción.

De todos los formatos de traducción disponibles, solamente XLIFF y *gettext* están soportados por todas las herramientas que utilizan los traductores profesionales. Además, al estar basado en XML, puedes validar los contenidos del archivo a medida que lo creas.

A partir de la versión 2.6 de Symfony los archivos XLIFF soportan la creación de notas y comentarios. Como una buena traducción depende totalmente de un buen contexto, estas notas XLIFF son ideales para proporcionar toda esa información de contexto.

TRUCO El *bundle* JMSTranslationBundle (https://github.com/schmittjoh/JMSTranslationBundle) ofrece una interfaz web para ver y editar los archivos de traducción. También dispone de herramientas avanzadas para extraer de las plantillas las cadenas de texto que deben traducirse e incluso puede actualizar automáticamente los archivos XLIFF.

8.2. Organizando los archivos de traducción

BUENA PRÁCTICA Almacena todos los archivos de traducción en el directorio `app/Resources/translations/`.

Los programadores Symfony normalmente guardaban los archivos de traducción dentro de los directorios `Resources/translations/` de los *bundles* de la aplicación.

Sin embargo, almacenar todos los archivos en `app/Resources/translations/` es más cómodo porque así se centralizan todos estos archivos. Además, cualquier archivo colocado en `app/Resources/` se considera con más prioridad que los archivos equivalente definidos en el directorio `Resources/` de los *bundles*. Así que tus archivos de traducción *machacarán* cualquier traducción idéntica definida en los *bundles* de terceros, algo que es muy útil en las aplicaciones.

8.3. Definiendo claves para las traducciones

BUENA PRÁCTICA Utiliza claves en vez de contenidos para identificar cada traducción.

Emplear claves en vez de contenidos simplifica mucho la gestión de los archivos de traducción. El motivo es que puedes cambiar los contenidos del idioma principal sin necesidad de cambiar todos los archivos de todos los idiomas.

Además, las claves siempre deben describir su *propósito* y no su *localización*. Si por ejemplo defines un formulario con un campo cuyo título es `Nombre de usuario`, entonces la clave debería ser `label.nombre_usuario` y no `formulario_edit.campos_obligatorios.label.nombre_usuario`.

8.4. Ejemplo de archivo de traducción

Aplicando todas las buenas práticas anteriores, el archivo de traducción de la aplicación para el idioma inglés tendría este aspecto:

```
<!-- app/Resources/translations/messages.en.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" target-language="en" datatype="plaintext">
        <body>
            <trans-unit id="1">
                <source>title.post_list</source>
                <target>Post List</target>
            </trans-unit>
        </body>
```

```
    </file>
</xliff>
```

Esta página se ha dejado vacía a propósito

Capítulo 9.

Seguridad

9.1. Autenticación y *firewalls*

En las aplicaciones Symfony puedes autenticar a tus usuarios con cualquier método que definas y puedes cargar la información de los usuarios desde cualquier fuente. Como este es un tema bastante complejo, puedes echar un vistazo a los tutoriales oficiales sobre la seguridad de Symfony (http://symfony.com/doc/current/cookbook/security/index.html) .

Independientemente de tus necesidades, la autenticación se configura en el archivo `security.yml` bajo la clave `firewalls`.

BUENA PRÁCTICA A menos que definas dos mecanismos de autenticación diferentes (ejemplo, formulario de login para el sitio y sistema de *tokens* para la API) define un único *firewall* en tu aplicación con la opción `anonymous` activada.

La mayoría de aplicaciones solamente utilizan un mecanismo de autenticación y un conjunto de usuarios. Por eso te basta con tener un único *firewall*. Una excepción válida es cuando utilizas más de un mecanismo de autenticación, como por ejemplo cuando dispones de una API protegida.

Además, deberías activar siempre la opción `anonymous` del *firewall*. Para restringir el acceso a las diferentes partes de la aplicación, utiliza la configuración de la opción `access_control`.

BUENA PRÁCTICA Utiliza `bcrypt` para codificar las contraseñas de tus usuarios.

Cuando se almacenan las contraseñas de los usuarios en el sistema, la recomendación consiste en utilizar el codificador `bcrypt` en vez del tradicional codificador basado en `SHA-512`. La principal ventaja de `bcrypt` es que ya incluye el *salt* como parte de la contraseña, lo que le protege frente a ataques de tipo *rainbow table*. Además, al tratarse de un algoritmo adaptativo, es posible ralentizar su ejecución para resistir mejor a los ataques de fuerza bruta.

Con todo esto, la configuración de la aplicación de prueba queda de la siguiente manera para utilizar un formulario de login y cargar los usuarios mediante la base de datos:

```
security:
    encoders:
        AppBundle\Entity\User: bcrypt

    providers:
        database_users:
            entity: { class: AppBundle:User, property: username }

    firewalls:
        secured_area:
            pattern: ^/
            anonymous: true
            form_login:
                check_path: security_login_check
                login_path: security_login_form

            logout:
                path: security_logout
                target: homepage

# No se muestra la sección 'access_control'
```

TRUCO El código fuente de la aplicación de prueba incluye comentarios que explican detalladamente cada parte de este archivo.

9.2. Autorización

Symfony también define varias formas de configurar la autorización, es decir, de restringir el acceso de los usuarios a los recursos. En concreto, puedes hacer uso de la configuración `access_control` en el archivo security.yml (http://symfony.com/doc/current/reference/configuration/security.html), de la anotación @Security (http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/security.html) y del método isGranted() (http://symfony.com/doc/current/book/security.html#access-control) del servicio `security.context`.

BUENA PRÁCTICA

- Utiliza `access_control` para proteger secciones completas de tu sitio web.
- Utiliza la anotación `@Security` para proteger recursos individuales.
- Utiliza el servicio `security.context` cuando la lógica relacionada con la seguridad sea más compleja.

Al margen de lo anterior, existen otras formas de centralizar toda la lógica relacionada con la seguridad, como por ejemplo los *security voters* y las ACL o *listas de control de acceso*.

[bestpractice]

- Utiliza un *security voter* para definir las restricciones de seguridad de los recursos de la aplicación.

- Utiliza una ACL cuando definas restricciones de seguridad granulares en las que cada objeto define el acceso permitido para cada usuario. [/bestpractice]

9.3. La anotación @Security

La anotación `@Security` es la forma más sencilla de controlar el acceso a cada controlador de la aplicación. Su sintaxis es muy fácil de leer y resulta muy cómodo configurar el acceso de cada controlador justo encima del código de la acción.

En la aplicación de prueba, para crear un nuevo artículo en el blog es necesario disponer del *role* `ROLE_ADMIN`. Con la anotación `@Security`, el código resultante es:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
// ...

/**
 * Displays a form to create a new Post entity.
 *
 * @Route("/new", name="admin_post_new")
 * @Security("has_role('ROLE_ADMIN')")
 */
public function newAction()
{
    // ...
}
```

9.3.1. Definiendo restricciones complejas mediante expresiones

Si la lógica de tu aplicación es más compleja, puedes utilizar expresiones (http://symfony.com/doc/current/components/expression_language/introduction.html) dentro de la anotación `@Security`. En el siguiente ejemplo, se restringe el acceso para que sólo puedan editar los artículos aquellos usuarios cuyo email coincida con el autor del artículo (obtenido mediante el método `getAuthorEmail` del objeto `Post`):

```
use AppBundle\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Route("/{id}/edit", name="admin_post_edit")
 * @Security("user.getEmail() == post.getAuthorEmail()")
 */
public function editAction(Post $post)
{
    // ...
}
```

Esta configuración requiere del uso de los ParamConverters (http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html) , ya que hay que realizar una consulta a la base de datos para obtener el objeto `Post` y asignarlo al argumento `$post` del controlador.

Esto es lo que hace que puedas utilizar una variable llamada `post` dentro de la expresión de la anotación `@Security`.

La gran desventaja de las expresiones es que no se pueden reutilizar fácilmente en otras partes de la aplicación. Imagina que quieres añadir en una plantilla un enlace a la acción de editar el artículo pero solamente si el usuario que está accediendo a la aplicación es autor de ese artículo. Ahora mismo tendrías que repetir esa misma expresión utilizando el código Twig:

```
{% if app.user and app.user.email == post.authorEmail %}
    <a href=""> ... </a>
{% endif %}
```

La solución más sencilla para reutilizar esta lógica sería crear un nuevo método en la entidad `Post` que compruebe si el usuario indicado es el autor del artículo:

```
// src/AppBundle/Entity/Post.php
// ...

class Post
{
    // ...

    /**
     * ¿Es el usuario indicado el autor del Post?
     *
     * @return bool
     */
    public function isAuthor(User $user = null)
    {
        return $user && $user->getEmail() == $this->getAuthorEmail();
    }
}
```

Ahora ya puedes reutilizar este mismo método tanto en el controlador como en la plantilla:

```
use AppBundle\Entity\Post;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Route("/{id}/edit", name="admin_post_edit")
 * @Security("post.isAuthor(user)")
 */
public function editAction(Post $post)
{
    // ...
}
```

```
{% if post.isAuthor(app.user) %}
    <a href=""> ... </a>
{% endif %}
```

9.4. Comprobando los permisos sin @Security

El ejemplo de la sección anterior utiliza `@Security` junto con un *ParamConverter* para poder acceder al valor de la variable `post`. Si no te gusta hacerlo así o tus requerimientos son más complejos, recuerda que la seguridad se puede comprobar fácilmente con código PHP:

```
/**
 * @Route("/{id}/edit", name="admin_post_edit")
 */
public function editAction($id)
{
    $post = $this->getDoctrine()->getRepository('AppBundle:Post')
        ->find($id);

    if (!$post) {
        throw $this->createNotFoundException();
    }

    if (!$post->isAuthor($this->getUser())) {
        throw $this->createAccessDeniedException();
    }

    // ...
}
```

9.5. Los *Security Voters*

Si la lógica relacionada con la seguridad es compleja y no se puede centralizar en un método de la entidad, como el anterior método `isAuthor()`, entonces deberías utilizar un *security voter* propio. La gran ventaja de los *voters* respecto a las ACL (http://symfony.com/doc/current/cookbook/security/acl.html) es que son un orden de magnitud más sencillos de crear y en casi todos los casos ofrecen la misma flexibilidad.

Para utilizar este mecanismo, crea primero la clase del *voter*. El siguiente ejemplo muestra la case utilizada en la aplicación para el *voter* basado en el método `getAuthorEmail`:

```
namespace AppBundle\Security;

use Symfony\Component\Security\Core\Authorization\Voter\AbstractVoter;
use Symfony\Component\Security\Core\User\UserInterface;

// La clase AbstractVoter está disponible a partir de Symfony 2.6
class PostVoter extends AbstractVoter
{
    const CREATE = 'create';
    const EDIT   = 'edit';

    protected function getSupportedAttributes()
    {
        return array(self::CREATE, self::EDIT);
```

```
    }

    protected function getSupportedClasses()
    {
        return array('AppBundle\Entity\Post');
    }

    protected function isGranted($attribute, $post, $user = null)
    {
        if (!$user instanceof UserInterface) {
            return false;
        }

        if ($attribute == self::CREATE && in_array(ROLE_ADMIN, $user->getRoles())) {
            return true;
        }

        if ($attribute == self::EDIT && $user->getEmail() === $post->getAuthorEmail()) {
            return true;
        }

        return false;
    }
}
```

Para activar el *voter* en la aplicación, define un nuevo servicio y etiquétalo con la etiqueta `security.voter`:

```
# app/config/services.yml
services:
    # ...
    post_voter:
        class:      AppBundle\Security\PostVoter
        public:     false
        tags:
           - { name: security.voter }
```

Lo bueno del *voter* es que también se puede utilizar junto con la anotación `@Security`:

```
/**
 * @Route("/{id}/edit", name="admin_post_edit")
 * @Security("is_granted('edit', post)")
 */
public function editAction(Post $post)
{
    // ...
}
```

Y por supuesto también se puede combinar con el servicio `security.context` mediante el método `isGranted()`:

```
/**
 * @Route("/{id}/edit", name="admin_post_edit")
 */
public function editAction($id)
{
    $post = // query for the post ...

    if (!$this->get('security.context')->isGranted('edit', $post)) {
        throw $this->createAccessDeniedException();
    }
}
```

9.6. Siguientes pasos

El *bundle* FOSUserBundle () , desarrollado por la comunidad Symfony, simplifica la gestión de los usuarios almacenados en la base de datos. También se encarga de tareas como el registro de usuarios o el envío de emails cuando a un usuario se le olvida su contraseña.

La funcionalidad Remember Me (http://symfony.com/doc/current/cookbook/security/remember_me.html) del componente de seguridad de Symfony permite mantener conectados a tus usuarios para que no tengan que introducir sus credenciales cada vez que visiten tu sitio.

Una de las funcionalidades más útiles del componente de seguridad, y que es esencial por ejemplo para dar soporte al usuario, consiste en acceder a la aplicación *impersonando* a otro usuario. De esta manera puedes comprobar de primera mano el error o problema que está sufriendo el usuario. Consulta la sección sobre impersonar usuarios (http://symfony.com/doc/current/cookbook/security/impersonating_user.html) para conocer los detalles.

Por último, si en tu empresa se utiliza un método de *login* no soportado por Symfony, puedes crear tu propio proveedor de usuarios (http://symfony.com/doc/current/cookbook/security/custom_provider.html) y tu propio proveedor de autenticación (http://symfony.com/doc/current/cookbook/security/custom_authentication_provider.html) .

Esta página se ha dejado vacía a propósito

Capítulo 10.

Assets web

Los *assets web* son las hojas de estilo CSS, los archivos JavaScript y las imágenes que se utilizan en el *frontend* de las aplicaciones para que tengan un buen aspecto. Normalmente los programadores Symfony crean estos archivos en los directorios `Resources/public/` de los *bundles*.

BUENA PRÁCTICA Almacena todos los *assets* en el directorio `web/` del proyecto.

Dispersar los *assets* entre decenas de directorios diferentes hace que sea más difícil gestionarlos por parte de tu equipo. El trabajo de los diseñadores será mucho más fácil si todos los *assets* se encuentran en un único lugar.

Además, al centralizar tus *assets*, el código de las plantillas se simplifica porque los enlaces son mucho más concisos:

```
<link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}" />
<link rel="stylesheet" href="{{ asset('css/main.css') }}" />

{# ... #}

<script src="{{ asset('js/jquery.min.js') }}"></script>
<script src="{{ asset('js/bootstrap.min.js') }}"></script>
```

NOTA El directorio `web/` es público, por lo cualquier usuario puede acceder a todos sus contenidos directamente desde su navegador. Si prefieres que los usuarios no accedan a los archivos originales de los *assets* (por ejemplo los archivos CoffeScript, Sass y LESS), guarda los archivos originales en `app/Resources/assets/` y almacena en `web/` solamente el resultado de compilar esos archivos.

10.1. Utilizando Assetic

Hoy en día es prácticamente imposible que un sitio web defina solamente unos pocos archivos CSS y JavaScript estáticos. Seguramente tus proyectos utilizarán varios archivos Sass o LESS que se compi-

lan a código CSS y después se combinan y minimizan para mejorar el rendimiento de la parte frontal de la aplicación.

Existen muchas herramientas que resuelven bien este problema, incluyendo algunas especialmente diseñadas para el *frontend* y que no utilizan PHP, como por ejemplo GruntJS.

BUENA PRÁCTICA Utiliza Assetic para compilar, combinar y minimizar los *assets web*, a menos que sepas utilizar bien herramientas del *frontend* como GruntJS.

Assetic (http://symfony.com/doc/current/cookbook/assetic/asset_management.html) es un gestor de *assets* capaz de compilar los *assets* desarrollados con decenas de tecnologías, inlucyendo por supuesto LESS, Sass y CoffeScript. Combinar todos estos *assets* en un único archivo mediante Assetic es tan sencillo como utilizar la etiqueta `{% stylesheets %}` de Twig:

```
{% stylesheets
    'css/bootstrap.min.css'
    'css/main.css'
    filter='cssrewrite' output='css/compiled/all.css' %}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}

{# ... #}

{% stylesheets
    'js/jquery.min.js'
    'js/bootstrap.min.js'
    output='js/compiled/all.js' %}
    <script src="{{ asset_url }}"></script>
{% endstylesheets %}
```

10.1.1. Aprendiendo más sobre Assetic

Assetic también es capaz de minimizar los archivos CSS y JavaScript utilizando UglifyCSS/UglifyJS (http://symfony.com/doc/current/cookbook/assetic/uglifyjs.html) , para aumentar el rendimiento de tu sitio web. De hecho, Assetic también puede comprimir las imágenes (http://symfony.com/doc/current/cookbook/assetic/jpeg_optimize.html) para reducir su tamaño antes de servirlas a los usuarios.

10.2. Aplicaciones basadas en el *frontend*

Algunas tecnologías recientes, como AngularJS, han ganado mucha popularidad entre la comunidad de programadores para desarrollar aplicaciones *frontend* que se comunican con el servidor mediante una API.

Si estás desarrollando una aplicación de este tipo, es preferible que utilices las herramientas recomendadas por la tecnología que utilices, como por ejemplo Bower y GruntJS. De hecho, deberías desarrollar tu *frontend* totalmente desacoplado respecto del *backend* Symfony (incluso separando los repositorios de código). Consulta la documentación oficial de Assetic (https://github.com/kriswallsmith/assetic) para conocer todas sus funcionalidades.

Capítulo 11.

Tests

Generalmente los programadores crean dos tipos de tests. Los tests unitarios comprueban la entrada/salida de determindos métodos o funciones. Los tests funcionales utilizan un navegador *emulado* para navegar por las páginas de tu sitio o aplicación, pinchar en sus enlaces y rellenar sus formularios para comprobar si los resultados obtenidos son los esperados.

11.1. Tests unitarios

Los tests unitarios comprueban que la lógica de negocio de tu aplicaicón funciona bien. Esta lógica de negocio es totalmente independiente del framework y por eso Symfony no incluye ninguna herramienta para estos tests. La mayoría de programadores utilizan PhpUnit (https://phpunit.de/) y PhpSpec (http://www.phpspec.net/) .

11.2. Tests funcionales

Como crear los tests funcionales de la aplicación lleva un tiempo, muchos programadores los ignoran completamente y no hacen ningún test. La recomendación es que al menos desarrolles unos tests funcionales sencillos, ya que te costará muy poco tiempo y verás que son muy útiles para descubrir lo antes posible errores graves en tu aplicación.

BUENA PRÁCTICA Define al menos un test funcional que compruebe que todas las páginas de tu sitio o aplicación se cargan bien.

Este test funcional es tan sencillo como lo siguiente:

```
/** @dataProvider provideUrls */
public function testPageIsSuccessful($url)
{
    $client = self::createClient();
    $client->request('GET', $url);

    $this->assertTrue($client->getResponse()->isSuccessful());
}
```

```
public function provideUrls()
{
    return array(
        array('/'),
        array('/posts'),
        array('/post/fixture-post-1'),
        array('/blog/category/fixture-category'),
        array('/archives'),
        // ...
    );
}
```

El código de este ejemplo comprueba que todas las URLs indicadas se cargan bien. Para ello se utiliza el método `isSuccessful()`, que comprueba si el código de estado HTTP se encuentra entre `200` y `299`. Aunque puede parecerte que este test no es demasiado útil, en realidad merece la pena añadirlo a tu *suite* de tests por el poco esfuerzo que te costará crearlo.

11.2.1. No generes las URL de los tests

Quizás te estés preguntando por qué en el anterior test funcional no se usa el servicio generador de URL a partir del nombre de las rutas.

BUENA PRÁCTICA Escribe las URL de los tests a mano, sin utilizar el servicio `router` de Symfony para generarlas.

Imagina que dispones del siguiente test funcional que utiliza el servicio `router` para generar la URL a partir del nombre de la ruta:

```
public function testBlogArchives()
{
    $client = self::createClient();
    $url = $client->getContainer()->get('router')->generate('blog_archives');
    $client->request('GET', $url);

    // ...
}
```

Aunque el código anterior funciona correctamente, tiene una desventaja muy grande. Si un programador cambia sin querer la URL de la ruta `blog_archives`, este test no se romperá y por tanto no te darás cuenta del error. Sin embargo, la URL anterior dejará de funcionar para los visitantes del sitio, por lo que todos los enlaces anteriores y todos los marcadores dejarán de funcionar sin que te des cuenta.

11.2.2. Archivos de datos

Si en tu aplicación quieres generar buenos archivos con datos de prueba, puedes utilizar las librerías Faker (https://github.com/fzaninotto/Faker) y Alice (https://github.com/nelmio/alice) .

11.3. Tests para el código JavaScript

El *emulador* de navegador que utiliza Symfony para los tests funcionales no permite probar el comportamiento JavaScript de tus páginas. Así que si en tu aplicación es importante probar ese código, deberías utilizar la librería Mink (http://mink.behat.org) junto con PHPUnit.

Obviamente si tu aplicación utiliza JavaScript en todas sus funcionalidades, sería mejor utilizar una herramienta de tests especialmente pensada para JavaScript.