Framework Symfony

Refrencia https://uniwebsidad.com/libros/symfony-2-4/capitulo-2/el-controlador-frontal-al-rescate

Capítulo 1. Symfony2 y los fundamentos de HTTP

1.1. HTTP es simple

HTTP ("*HyperText Transfer Protocol*") **es un lenguaje basado en texto** que permite a dos máquinas comunicarse entre sí. ¡Eso es todo! La siguiente *conversación* es la que por ejemplo tiene lugar cuando quieres acceder a la última tira cómica publicada por el sitio xkcd:

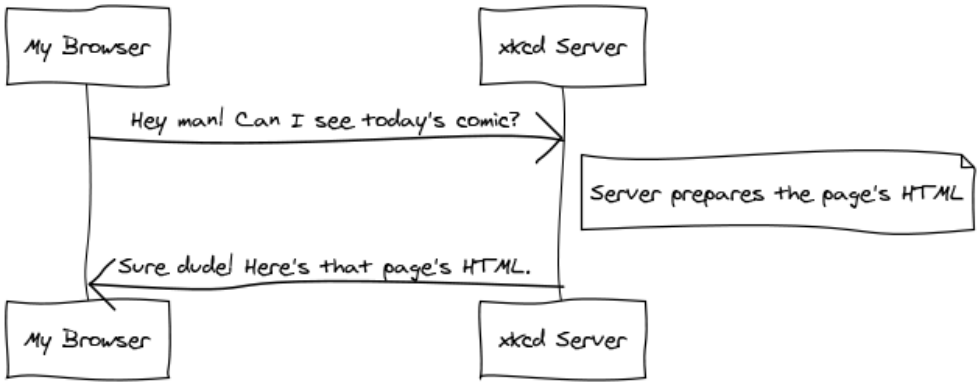

Figura 1.1 Flujo HTTP para obtener la tira cómica más reciente de Xkcd

Y aunque el lenguaje realmente utilizado es un poco más formal, sigue siendo bastante simple.

HTTP es el término utilizado para describir este lenguaje simple basado en texto. Y no importa cómo desarrolles en la web, el objetivo de tu servidor siempre es entender las peticiones de texto simple, **y devolver respuestas en texto simple**.

Symfony2 está diseñado en base a esa realidad. Aunque a veces no te des cuenta, HTTP es algo que usas todos los días.

1.1.1. Paso 1: El cliente envía una petición

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

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

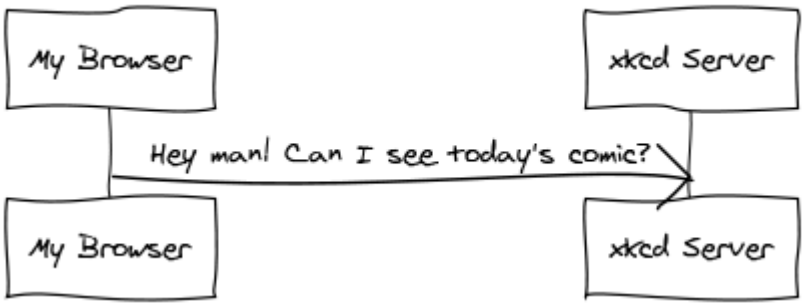

Figura 1.2 Petición HTTP para obtener la tira cómica más reciente de Xkcd

Utilizando el lenguaje HTTP, **esta petición en realidad sería algo parecido a lo siguiente**:

```
GET / HTTP/1.1

Host: xkcd.com

Accept: text/html

User-Agent: Mozilla/5.0 (Macintosh)
```

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

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

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

Teniendo esto en cuenta, puedes imaginar cómo sería por ejemplo la petición HTTP necesaria para borar un artículo específico de un blog:

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

NotaEn realidad, hay nueve métodos HTTP definidos por la especificación HTTP, pero muchos de ellos no se utilizan o no están soportados. De hecho, muchos navegadores modernos no soportan los métodos `PUT` y `DELETE`.

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

1.1.2. Paso 2: El servidor devuelve una respuesta

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

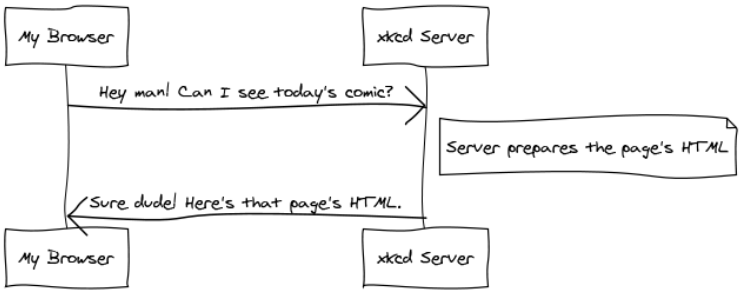

Figura 1.3 Respuesta HTTP para obtener la tira cómica más reciente de Xkcd

Traducida a HTTP, la respuesta enviada de vuelta al navegador es similar a lo siguiente:

```
HTTP/1.1 200 OK

Date: Sat, 02 Apr 2011 21:05:05 GMT

Server: lighttpd/1.4.19

Content-Type: text/html



<html>

  <!-- HTML de la tira cómica de Xkcd -->

</html>
```

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

Al igual que la petición, **una respuesta HTTP contiene datos adicionales conocidos como cabeceras HTTP**. Por ejemplo, una cabecera importante de la respuesta HTTP es `Content-Type`. Un mismo recurso se puede devolver en varios formatos diferentes (HTML, XML, JSON, etc.) y la cabecera `Content-Type` dice al cliente qué formato se ha utilizado (para ello utiliza valores estándar como `text/html` que se conocen como *Internet Media Types*). Puedes encontrar la lista completa de tipos de contenido en el artículo *Lista de tipos de contenido de Internet* en la Wikipedia.

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

1.1.3. Peticiones, respuestas y desarrollo web

Esta conversación **petición-respuesta es el proceso fundamental en el que se basa toda la comunicación en la web**. Y a pesar de ser tan importante y poderoso, al mismo tiempo es muy sencillo.

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

Symfony está diseñado para adaptarse a esta realidad.

TrucoPuedes obtener más información acerca de la especificación HTTP, en la referencia original HTTP 1.1 RFC. También puedes leer la referencia HTTP Bis, que es una versión actualizada y más detallada de la referencia anterior. Una gran herramienta para comprobar tanto la petición como las cabeceras de la respuesta mientras navegas es la extensión *Live HTTP Headers* de Firefox o el *Inspector Web* de los navegadores Chrome y Safari.

1.2. Peticiones y respuestas en PHP

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

```
<?php

$uri = $_SERVER['REQUEST_URI'];

$foo = $_GET['foo'];

header('Content-type: text/html');

echo 'La URI solicitada es: '.$uri;

echo 'El valor del parámetro "foo" es: '.$foo;
```

Por extraño que parezca, **esta pequeña aplicación está obteniendo información de la petición HTTP y la utiliza para crear una respuesta HTTP.** En lugar de analizar el mensaje HTTP de la petición, PHP crea *variables superglobales* como `$_SERVER` y `$_GET` que contienen toda la información de la petición. Del mismo modo, en lugar de devolver la respuesta HTTP con formato de texto, puedes usar la función `header()` para crear las cabeceras de la respuesta y simplemente imprimir el contenido que se enviará en el mensaje de la respuesta. Después PHP crea la verdadera respuesta HTTP que se devuelve al cliente:

```
HTTP/1.1 200 OK

Date: Sat, 03 Apr 2011 02:14:33 GMT

Server: Apache/2.2.17 (Unix)

Content-Type: text/html

La URI solicitada es: /testing?foo=symfony

El valor del parámetro "foo" es: symfony
```

1.3. Peticiones y respuestas en Symfony

Symfony ofrece una alternativa al enfoque de PHP a través de dos clases que te permiten interactuar con la petición HTTP y la respuesta de una manera más fácil.

La clase `Request` representa la petición HTTP siguiendo la filosofía de orientación a objetos. Con ella, tienes toda la información a tu alcance:

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

$request = Request::createFromGlobals();

// la URI solicitada (p.e. /contacto) menos algunos parámetros de la consulta

$request->getPathInfo();

// recupera las variables GET y POST respectivamente
```

```
$request->query->get('foo');

$request->request->get('bar', 'valor utilizado si "bar" no existe');

// recupera las variables de SERVER

$request->server->get('HTTP_HOST');



// recupera una instancia del archivo subido identificado por 'foo'

$request->files->get('foo');



// recupera un valor de una COOKIE

$request->cookies->get('PHPSESSID');



// recupera una cabecera HTTP de la petición, normalizada, con índices en minúscula

$request->headers->get('host');

$request->headers->get('content_type');

$request->getMethod();      // GET, POST, PUT, DELETE, HEAD

$request->getLanguages();   // un array de idiomas aceptados por el cliente
```

Una ventaja añadida es que la clase `Request` hace un montón de trabajo adicional del que no tienes que preocuparte. Por ejemplo, el método `isSecure()` internamente comprueba tres valores PHP diferentes que pueden indicar si el usuario está conectado a través de una conexión segura (es decir, `https`).

`ParameterBags` y atributos de la petición

Como vimos anteriormente, las variables `$_GET` y `$_POST` son accesibles a través de las propiedades `query` y `request` de la clase `Request`, respectivamente. Cada uno de estos objetos es un objeto de la clase `ParameterBag`, la cual cuenta con métodos cómo `get()`, `has()` y `all()` entre otros.

De hecho, todas las propiedades públicas utilizadas en el código del ejemplo anterior son un ejemplo del `ParameterBag`.

La clase `Request` también tiene una propiedad pública `attributes`, que almacena información especial relacionada sobre el funcionamiento interno de la aplicación. En Symfony2, la propiedad `attibutes` guarda por ejemplo los valores relacionados con el sistema de enrutamiento que se explicará más adelante (como por ejemplo, `_controller` y `_route`). El propósito de la propiedad `attributes` es el de preparar y almacenar información del contexto específico de la petición.

Symfony también proporciona una clase `Response`, que simplifica la representación de la respuesta HTTP en PHP. Esto permite que tu aplicación utilice una **interfaz orientada a objetos para construir la respuesta que será devuelta al cliente:**

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

$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
```

```
$response->setStatusCode(Response::HTTP_OK);

$response->headers->set('Content-Type', 'text/html');



// imprime las cabeceras HTTP seguidas por el contenido

$response->send();
```

NotaLas constantes que definen los códigos de estado de HTTP (como por ejemplo `Response::HTTP_OK`) fueron añadidas en la versión 2.4 de Symfony.

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

TrucoLas clases `Request` y `Response` forman parte de un componente independiente incluido en Symfony llamado `HttpFoundation`. Este componente se puede utilizar en aplicaciones independientes de Symfony y también proporciona clases para manejar sesiones y subir archivos.

1.4. El viaje desde la petición hasta la respuesta

Al igual que el mismo HTTP, **los objetos `Petición` y `Response` son bastante simples**. **La parte difícil de la construcción de una aplicación es escribir lo que viene en el medio.**

En otras palabras, **el verdadero trabajo viene al escribir el código que interpreta la información de la petición y crea la respuesta**.

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

Symfony fue creado precisamente para resolver estos problemas y para que no tengas que hacerlo a mano.

1.4.1. El controlador frontal

Antiguamente, las aplicaciones web se construían de modo que **cada página web del sitio tenía su propio archivo físico**:

```
index.php

contacto.php

blog.php
```

Esta filosofía de trabajo tiene varios problemas, como la falta de flexibilidad de las *URL* (¿qué pasa si quieres cambiar `blog.php` a `noticias.php` sin romper todos tus enlaces?) y el hecho de que **cada archivo debe incluir a mano todos los archivos necesarios para la seguridad, conexiones a base de datos y para aplicar los estilos gráficos del sitio.**

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

Archivo solicitado	Archivo realmente ejecutado

Archivo solicitado	Archivo realmente ejecutado
`/index.php`	`index.php`
`/index.php/contacto`	`index.php`
`/index.php/blog`	`index.php`

Truco **Usando el módulo `mod_rewrite` de *Apache*** (o el equivalente en otros servidores web), las *URL* se pueden *limpiar* fácilmente para que quiten la parte del `index.php` y se queden simplemente en `/`, `/contacto` y `/blog`.

Ahora, todas las peticiones se manejan exactamente igual. En lugar de URL individuales ejecutando diferentes archivos PHP, **el controlador frontal siempre se ejecuta, y la redirección de cada URL a una parte diferente de la aplicación se realiza internamente. Esto resuelve los problemas comentados anteriormente.**

Casi todas las aplicaciones web modernas siguen la filosofía del *controlador frontal*, incluyendo aplicaciones como *WordPress*.

1.4.2. Mantente organizado

Una vez dentro del controlador frontal, **¿cómo sabes qué página debes generar y cómo puedes generar todas las páginas sin que la aplicación se vuelva caótica? El truco consiste en comprobar la URI entrante y ejecutar diferentes partes de tu código en función de ese valor**. Aunque es bastante *chapucero*, el siguiente código te podría servir para ello:

```
// Contenido del archivo index.php

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;



$request = Request::createFromGlobals();

$path = $request->getPathInfo(); // La ruta URI solicitada



if (in_array($path, array('', '/')) {

    $response = new Response('Bienvenido a nuestra portada.');

} elseif ($path == '/contacto') {

    $response = new Response('Contáctanos');

} else {

    $response = new Response('Página no encontrada.', Response::HTTP_NOT_FOUND);

}

$response->send();
```

IMPORTANTE : Escalar el código anterior para una aplicación real es un problema difícil de resolver. Afortunadamente esto es exactamente para lo que Symfony está diseñado

1.4.3. El flujo de las aplicaciones Symfony

Cuando **dejas que Symfony controle cada petición**, **tu vida como programador es mucho más fácil**. Symfony sigue el mismo patrón simple en cada petición. **Las peticiones entrantes son interpretadas por el enrutador y pasadas a las funciones controladoras, que devuelven objetos `Response`.**

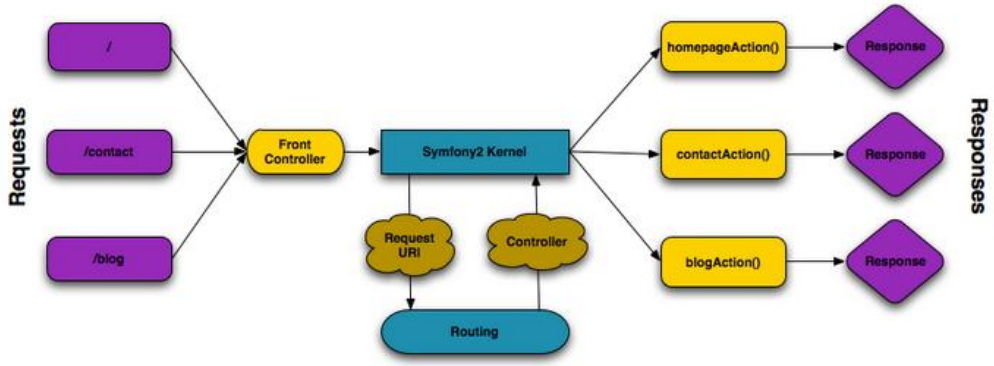

Figura 1.4 Flujo de la petición en Symfony2

Cada página de tu sitio está definida en un archivo de configuración de rutas que asigna a cada URL una función PHP diferente. **El trabajo de cada función PHP (conocida como *controlador*)**, es utilizar la información de la petición — junto con muchas otras herramientas que Symfony pone a tu disposición — para crear y devolver un objeto `Response`. **En otras palabras, el controlador es donde está tu código: ahí es dónde se interpreta la petición y se crea una respuesta**. ¡Así de fácil!

Repasemos:

- **Cada petición ejecuta un archivo controlador frontal**;
- **El sistema de enrutado determina qué función PHP se ejecuta en base a la información de la petición** y la configuración de enrutado que hemos creado;
- **Se ejecuta la función PHP adecuada**, donde tu código crea y devuelve el objeto `Response`.

1.4.4. Una petición Symfony en la práctica

Sin entrar demasiado en los detalles, veamos este proceso en la práctica. Supongamos que deseas agregar una página `/contacto` a tu aplicación Symfony. En primer lugar, empezamos agregando una entrada `/contacto` a tu archivo de configuración de rutas:

- **YAML**
- XML
- PHP

```yaml
# app/config/routing.yml

contacto:

    path:     /contacto

    defaults: { _controller: AcmeDemoBundle:Main:contacto }
```

```xml
<route id="contacto" path="/contacto">

    <default key="_controller">AcmeBlogBundle:Main:contacto</default>

</route>
```

```php
// app/config/routing.php
```

```
use Symfony\Component\Routing\RouteCollection;

use Symfony\Component\Routing\Route;



$collection = new RouteCollection();

$collection->add('contacto', new Route('/contacto', array(

    '_controller' => 'AcmeBlogBundle:Main:contacto',

)));



return $collection;
```

NotaEn este ejemplo utilizamos `YAML` para definir la configuración de enrutado, pero también se pueden utilizar los formatos XML y PHP.

Cuando alguien visita la página `/contacto`, **Symfony2 detecta que se trata de esta ruta y se ejecuta el controlador especificado**. **Como veremos en el capítulo de enrutamiento, la cadena `AcmeDemoBundle:Main:contacto` es un atajo que apunta al método `contactoAction()`** de PHP dentro de una clase llamada `MainController`:

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

namespace Acme\DemoBundle\Controller;



use Symfony\Component\HttpFoundation\Response;



class MainController

{

    public function contactoAction()

    {

        return new Response('<h1>Contáctanos</h1>');

    }

}
```

En este ejemplo sencillo, **el controlador simplemente crea un objeto `Response` con el código HTML `"<h1>Contáctanos</h1>"`.** En el capítulo **dedicado a los controladores aprenderás cómo hacer que los controladores** utilicen plantillas para generar el contenido HTML en vez de tener que escribirlo directamente en el objeto `Response`. **Esto hace que el controlador deba**

preocuparse sólo de las cosas difíciles: la interacción con la base de datos, la manipulación de la información o el envío de mensajes de correo electrónico.

1.5. Symfony2: construye tu aplicación, no tus herramientas

Ahora sabemos que **el objetivo de cualquier aplicación es interpretar cada petición entrante y crear una respuesta adecuada**. **Cuando una aplicación crece, es más difícil organizar tu código y que a la vez sea fácil de mantener**. Además, siempre te vas a encontrar con **las mismas tareas complejas**: la persistencia de información a la base de datos, procesar y reutilizar plantillas, manejar los formularios, enviar mensajes de correo electrónico, validar los datos del usuario, administrar la seguridad del sitio web, etc.

La buena noticia es que **todos los programadores web se encuentran con esos mismos problemas.** Así que Symfony proporciona una plataforma completa, con herramientas que te permiten construir tu aplicación, no tus herramientas. Con Symfony2, nada se te impone: **eres libre de usar la plataforma Symfony completa, o simplemente aquellas partes de Symfony que quieras.**

1.5.1. Herramientas independientes: los componentes de Symfony2

Con todo lo anterior, ¿qué es exactamente Symfony2?. En primer lugar, **Symfony2 es una colección de más de veinte librerías independientes que se pueden utilizar dentro de cualquier proyecto PHP**. **Estas librerías, llamadas *componentes de Symfony2*, son bastante útiles práticamente para cualquier aplicación**, independientemente de cómo desarrolles tu proyecto. Las siguientes son algunas de las más destacadas:

- `HttpFoundation`: **contiene las clases `Request` y `Response`**, así como otras clases para manejar sesiones y cargar archivos.
- `Routing`: **potente y rápido sistema de enrutado** que te permite asignar una URI específica (por ejemplo `/contacto`) a cierta información acerca de cómo se debe manejar dicha petición (por ejemplo, ejecutar el método `contactoAction()`).
- `Form`: una **completa y flexible plataforma para crear formularios y procesar los datos** presentados en ellos.
- `Validator`: un **sistema para crear reglas sobre datos y así comprobar si los datos que presenta el usuario son válidos** o no siguiendo esas reglas.
- `ClassLoader`: una **librería para cargar automáticamente clases PHP sin necesidad de añadir instrucciones `require`** a mano en los archivos que contienen esas clases.
- `Templating`: juego de herramientas para **utilizar plantillas, que soporta desde la herencia de plantillas** (es decir, una plantilla está decorada con un diseño) y hasta otras tareas comunes de las plantillas.
- `Security`: una poderosa librería para manejar **todo tipo de seguridad dentro de una aplicación.**
- `Translation`: plataforma para traducir cadenas de texto en tu aplicación.

Todos y cada uno de estos componentes están *desacoplados*, lo que significa que puedes utilizarlos en cualquier proyecto PHP, independientemente de si utilizas la plataforma Symfony2.

Cada componente está diseñado para utilizarlo si es conveniente o para sustituirlo cuando sea necesario.

1.5.2. La solución completa: la plataforma Symfony2

¿Qué es la plataforma Symfony2? La **plataforma Symfony2** es una librería PHP que realiza dos tareas diferentes:

- Proporciona una selección de componentes **Symfony2 y algunas librerías de terceros** (por ejemplo, `SwiftMailer` para enviar mensajes de correo electrónico).
- **Define una configuración adecuada** e incluye una capa que integra todas las diferentes partes (componentes, librerías, etc.)

El objetivo de la plataforma es **integrar muchas herramientas independientes con el fin de proporcionar una experiencia coherente al desarrollador.**

Symfony2 proporciona un potente conjunto de herramientas para desarrollar aplicaciones web rápidamente sin afectar excesivamente a tu forma de trabajar. Si estás comenzando con Symfony2, **lo mejor es que utilices una** distribución de Symfony2**, que proporciona un *esqueleto* de un proyecto** Symfony2 de prueba con varios parámetros ya preconfigurados. Si eres un usuario avanzado, puedes crearte tu propia distribución o incluso utilizar componentes individuales de Symfony2.

Capítulo 2. De PHP a Symfony2

¿Por qué Symfony2 es mejor que escribir código PHP *a pelo*? Si nunca has usado una plataforma PHP, o no estás familiarizado con la filosofía *MVC*, o simplemente te preguntas qué es todo ese *ruido* generado en torno a Symfony2, este capítulo es para ti. En vez de contarte que Symfony2 te permite desarrollar software más rápido y mejor que con PHP simple, vas a poder comprobarlo tu mismo.

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

Al final del capítulo, verás cómo Symfony2 se encarga de todas las tareas comunes (y aburridas) mientras que te permite recuperar el control de tu código.

2.1. Un blog sencillo creado con PHP simple

En este capítulo, crearemos la típica aplicación de blog utilizando sólo PHP simple.

Para empezar, crea una página que muestre las entradas del blog que se han persistido en la base de datos. Escribirla en PHP simple es rápido, pero un poco *sucio*:

```
<?php

// index.php



$link = mysql_connect('localhost', 'myuser', 'mypassword');

mysql_select_db('blog_db', $link);



$result = mysql_query('SELECT id, title FROM post', $link);

?>



    <html>
```

```
        <head>

        <title>List of Posts</title>

        </head>

        <body>

        <h1>List of Posts</h1>

        <ul>

            <?php while ($row = mysql_fetch_assoc($result)): ?>

            <li>

                <a href="/show.php?id=<?php echo $row['id'] ?>">

                    <?php echo $row['title'] ?>

                    </a>

            </li>

            <?php endwhile; ?>

        </ul>

        </body>

    </html>
```

```
<?php

mysql_close($link);
```

El código anterior es fácil de escribir y se ejecuta muy rápido, pero en cuanto la aplicación crece, es muy difícil de mantener. Sus principales problemas son los siguientes:

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

NotaOtro problema no mencionado aquí es el hecho de que la **base de datos está vinculada a MySQL**. Aunque no se ha tratado aquí, Symfony2 integra Doctrine, una librería que permite abstraer el acceso a la base de datos y el manejo de la información.

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

2.1.1. Aislando la parte de la vista

El código se puede mejorar fácilmente **separando la lógica de la aplicación (código PHP puro) y *"la parte de la vista"* o *"presentación"*,** que está formada por todo lo relacionado con el código HTML:

```
<?php

// index.php



$link = mysql_connect('localhost', 'myuser', 'mypassword');

mysql_select_db('blog_db', $link);



$result = mysql_query('SELECT id, title FROM post', $link);



$posts = array();

while ($row = mysql_fetch_assoc($result)) {

    $posts[] = $row;

}



mysql_close($link);



// incluye el código HTML de la vista

require 'templates/list.php';
```

Ahora el código HTML está guardado en un archivo separado (`templates/list.php`). Este archivo contiene todo el código HTML junto con algunas pocas instrucciones PHP que utilizan la sintaxis alternativa recomendada para las plantillas PHP:

```
<html>

    <head>

    <title>List of Posts</title>

    </head>

    <body>
```

```
    <h1>List of Posts</h1>

    <ul>

        <?php foreach ($posts as $post): ?>

        <li>

            <a href="/read?id=<?php echo $post['id'] ?>">

                <?php echo $post['title'] ?>

                </a>

        </li>

        <?php endforeach; ?>

    </ul>

    </body>

</html>
```

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

En este caso, **nuestro controlador obtiene los datos de la base de datos y, luego los incluye en una plantilla para presentarlos al usuario.** Con el controlador aislado, podríamos cambiar fácilmente sólo el archivo de la plantilla si queremos servir los contenidos del blog en otro formato (por ejemplo, `list.json.php` para el formato *JSON*).

2.1.2. Aislando la lógica de la aplicación (*el dominio*)

Por ahora la aplicación sólo contiene una página. Pero ¿qué pasa si una segunda página necesita utilizar la misma conexión a la base de datos, e incluso el mismo resultado de la búsqueda de las entradas del blog? La solución consiste en refactorizar el código para que todas estas funciones básicas de acceso a datos de la aplicación estén aisladas en un **nuevo archivo llamado `model.php`:**

```
<?php

// model.php



function open_database_connection()

{

    $link = mysql_connect('localhost', 'myuser', 'mypassword');

    mysql_select_db('blog_db', $link);
```

```
    return $link;

}



function close_database_connection($link)

{

    mysql_close($link);

}



function get_all_posts()

{

    $link = open_database_connection();



    $result = mysql_query('SELECT id, title FROM post', $link);

    $posts = array();

    while ($row = mysql_fetch_assoc($result)) {

        $posts[] = $row;

    }

    close_database_connection($link);



    return $posts;

}
```

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

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

```
<?php

require_once 'model.php';
```

```
$posts = get_all_posts();
```

```
require 'templates/list.php';
```

Ahora, la única tarea del controlador es conseguir los datos de la capa del modelo de la aplicación (el modelo) y utilizar una plantilla para mostrar los datos.

Este es un ejemplo muy simple del patrón *modelo - vista - controlador*.

2.1.3. Aislando el diseño

La aplicación **ahora está dividida en tres partes distintas**, lo que nos ofrece varias ventajas y la oportunidad de volver a utilizar casi todo en diferentes páginas.

La única parte del código que no se puede reutilizar es el diseño HTML + CSS de la página. Vamos a solucionar este problema creando un nuevo archivo llamado `base.php`:

```
<!-- templates/base.php -->

<html>

    <head>

    <title><?php echo $title ?></title>

    </head>

    <body>

    <?php echo $content ?>

    </body>

</html>
```

La plantilla original (`templates/list.php`) ahora se puede simplificar para que utilice el archivo `base.php` anterior como base:

```
<?php $title = 'List of Posts' ?>
```

```
<?php ob_start() ?>

    <h1>List of Posts</h1>

    <ul>

        <?php foreach ($posts as $post): ?>

        <li>
```

```
            <a href="/read?id=<?php echo $post['id'] ?>">

                <?php echo $post['title'] ?>

                </a>

        </li>

        <?php endforeach; ?>

    </ul>

<?php $content = ob_get_clean() ?>



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

El código anterior introducido una metodología que nos permite reutilizar el diseño. Desafortunadamente, para conseguirlo estamos obligados a utilizar en la plantilla algunas funciones de PHP poco recomendables (`ob_start()`, `ob_get_clean()`). Symfony2 utiliza un componente de plantillas (`Templating`) que nos permite realizar esto de una manera más *limpia* y sencilla. En breve lo verás en acción.

2.2. Agregando una página `show` al blog

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

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

```
// model.php

function get_post_by_id($id)

{

    $link = open_database_connection();



    $id = mysql_real_escape_string($id);

    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;

    $result = mysql_query($query);

    $row = mysql_fetch_assoc($result);



    close_database_connection($link);
```

```
    return $row;

}
```

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

```
<?php

require_once 'model.php';



$post = get_post_by_id($_GET['id']);



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

Por último, crea el nuevo archivo de plantilla (`templates/show.php`) para mostrar una entrada individual del blog:

```
<?php $title = $post['title'] ?>



<?php ob_start() ?>

    <h1><?php echo $post['title'] ?></h1>



    <div class="date"><?php echo $post['date'] ?></div>

    <div class="body">

        <?php echo $post['body'] ?>

        </div>

<?php $content = ob_get_clean() ?>



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

Como has visto, **es muy fácil crear la segunda página sin duplicar código**. Sin embargo, esta nueva página introduce algunos problemas adicionales que una plataforma web puede resolver por ti. **Por ejemplo, si el usuario proporciona un valor ilegal para el parámetro `id` o un valor vacío, la consulta hará que la página se bloquee. En este caso, sería mejor que la aplicación generara una página de error de tipo `404` (algo que no es fácil con el código actual de la aplicación). Peor aún, si no te acuerdas de *limpiar* con la función `mysql_real_escape_string()` el parámetro `id` que te pasa el usuario, tu base de datos sería vulnerable a los ataques de tipo inyección `SQL`.**

Otro problema importante es que **cada archivo de tipo controlador debe incluir el archivo `model.php`. ¿Qué pasaría si cada archivo de controlador de repente tuviera que incluir un archivo adicional o realizar alguna tarea global (por ejemplo, reforzar la seguridad)?**

Tal como está ahora, **el código tendría que incluir a mano todos los archivos de los controladores**. Si olvidas incluir algo en un solo archivo, esperamos que no sea alguno relacionado con la seguridad.

2.3. El controlador frontal al rescate

Una solución mucho mejor **es usar un *controlador frontal*: un único archivo PHP a través del cual se procesen todas las peticiones del usuario.** Con un controlador frontal, la URI de la aplicación cambia un poco, pero se vuelve mucho más flexible:

```
Sin controlador frontal:

/index.php       => (ejecuta index.php) la página que lista los artículos.

/show.php        => (ejecuta show.php)  la página que muestra un artículo específico.



Con index.php como controlador frontal

/index.php       => (ejecuta index.php) la página que lista los artículos.

/index.php/show  => (ejecuta index.php) la página que muestra un artículo específico.
```

TrucoEl nombre del archivo `index.php` se puede eliminar en las URLs si utilizas las reglas de reescritura del servidor web Apache (o las equivalentes de los demás servidores web). En ese caso, la URI resultante de la página `show` del blog sería simplemente `/show`.

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

2.3.1. Creando el controlador frontal

Estás a punto de dar un gran paso en la aplicación. Con un archivo manejando todas las peticiones, puedes centralizar cosas como el manejo de la seguridad, la carga de la configuración y el enrutamiento. En nuestra aplicación, `index.php` ahora debe ser lo suficientemente inteligente como para mostrar la lista de entradas del blog o mostrar la página de una entrada particular basándose en la URI solicitada:

```php
<?php

// index.php



// carga e inicia algunas librerías globales

require_once 'model.php';

require_once 'controllers.php';



// encamina la petición internamente

$uri = $_SERVER['REQUEST_URI'];

if ($uri == '/index.php') {
```

```
    list_action();

} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {

    show_action($_GET['id']);

} else {

    header('Status: 404 Not Found');

    echo '<html><body><h1>Page Not Found</h1></body></html>';

}
```

Para organizar mejor el código, los dos controladores anteriores (archivos `index.php` y `show.php`) se han convertido en funciones PHP del nuevo archivo `controllers.php`:

```
function list_action()

{

    $posts = get_all_posts();

    require 'templates/list.php';

}




function show_action($id)

{

    $post = get_post_by_id($id);

    require 'templates/show.php';

}
```

Como controlador frontal, `index.php` ha asumido un papel completamente nuevo, que **incluye la carga de los archivos principales de la aplicación (`model.php` y `controllers.php`) y la decisión de ejecutar uno de los dos controladores** (las funciones `list_action()` y `show_action()`). **En realidad, el controlador frontal está empezando a parecerse y actuar como el mecanismo que define Symfony2 para la manipulación y enrutado de peticiones**.

TrucoOtra ventaja del controlador frontal es la flexibilidad de las URL. Ten en cuenta que la URL de la página `show` del blog se puede cambiar de `/show` a `/read` cambiando el código en un único lugar. Antes, era necesario cambiar todo un archivo para cambiar el nombre. En Symfony2, las URL son incluso más flexibles.

Por ahora, la aplicación **ha evolucionado de un único archivo PHP, a una estructura organizada que permite la reutilización de código.** Debes estar contento, pero **aún lejos de estar satisfecho**. Por ejemplo, **el sistema de enrutamiento es muy mejorable, ya que no reconoce por ejemplo que la página `list` (`/index.php`) también debe ser accesible a través de `/`** (si has agregado las reglas de reescritura de *Apache*). Además, **en lugar de programar el blog, has *perdido* tu tiempo en preparar la "arquitectura" del código (por ejemplo, el enrutamiento, los controladores, las plantillas, etc.) Y pronto tendrás que perder más tiempo en la gestión de los formularios, la validación de la información, crear un archivo de log, la gestión de la seguridad, etc.**

¿Por qué tienes que dedicarte a solucionar estos problemas que se repiten una y otra vez en todos los proyectos?

2.3.2. Añadiendo un toque de Symfony2

¡Symfony2 al rescate! Antes de utilizar Symfony2, debes descargar sus archivos. Para ello se utiliza la herramienta llamada *Composer*, que se encarga de descargar la versión correcta de los archivos de Symfony, todas sus dependencias y también proporciona un cargador automático de clases (llamado *class loader* en inglés). Un cargador automático es una herramienta que permite empezar a utilizar clases PHP sin incluir explícitamente el archivo que contiene la clase.

Dentro del directorio raíz del proyecto crea un archivo llamado `composer.json` con el siguiente contenido:

```
{

    "require": {

        "symfony/symfony": "2.4.*"

    },

    "autoload": {

        "files": ["model.php","controllers.php"]

    }

}
```

A continuación, descarga Composer y ejecuta después el siguiente código para descargar Symfony dentro del directorio `vendor/` del proyecto:

```
$ php composer.phar install
```

Además de descargar todas las dependencias, *Composer* genera un archivo llamado `vendor/autoload.php`, que se encarga de cargar automáticamente todas las clases del *framework* Symfony y de todas las librerías que añadas en la sección `autoload` del archivo `composer.json`.

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

```
<?php

// index.php

require_once 'vendor/autoload.php';



use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;



$request = Request::createFromGlobals();
```

```php
$uri = $request->getPathInfo();

if ($uri == '/') {

    $response = list_action();

} elseif ($uri == '/show' && $request->query->has('id')) {

    $response = show_action($request->query->get('id'));

} else {

    $html = '<html><body><h1>Page Not Found</h1></body></html>';

    $response = new Response($html, Response::HTTP_NOT_FOUND);

}



// envía las cabeceras y la respuesta

$response->send();
```

Los controladores se encargan de devolver un objeto `Response`. Para simplificar la generación del código HTML de la respuesta, crea una nueva función llamada `render_template()`, la cual, por cierto, actúa un poco como el motor de plantillas de Symfony2:

```php
// controllers.php

use Symfony\Component\HttpFoundation\Response;



function list_action()

{

    $posts = get_all_posts();

    $html = render_template('templates/list.php', array('posts' => $posts));



    return new Response($html);

}



function show_action($id)
```

```
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', array('post' => $post));

    return new Response($html);
}

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

    return $html;
}
```

Aunque **solo ha añadido una pequeñísima parte de las ideas de Symfony2, la aplicación ahora es más flexible y fiable**. El objeto `Request` proporciona una manera segura para acceder a la información de la petición HTTP. El método `getPathInfo()` por ejemplo devuelve una URI limpia (siempre devolviendo `/show` y nunca `/index.php/show`).

Por lo tanto, incluso si el usuario va a `/index.php/show`, la aplicación es lo suficientemente inteligente para encaminar la petición hacia `show_action()`.

El objeto `Response` proporciona flexibilidad al construir la respuesta HTTP, permitiendo que las cabeceras HTTP y el contenido se definan a través de una interfaz orientada a objetos.

Y aunque las respuestas en esta aplicación son simples, esta flexibilidad será muy útil en cuanto tu aplicación crezca.

2.3.3. Aplicación de ejemplo en Symfony2

El blog ha mejorado bastante, pero todavía contiene una gran cantidad de código para ser una aplicación tan simple. Durante las mejoras de la aplicación, **se ha creado un sistema de enrutamiento muy sencillo y otro método que utiliza `ob_start()` y `ob_get_clean()` para procesar plantillas**. Si vas a continuar desarrollando este blog, **lo mejor es que utilices los componentes `Routing` y `Templating` de Symfony para mejorar esa parte sin esfuerzo.**

En lugar de resolver de nuevo los mismos problemas de siempre, deja que Symfony2 se encargue de ellos por ti. Aquí está la misma aplicación de ejemplo, pero construida ahora con Symfony2:

```php
<?php

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

namespace Acme\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')
            ->getManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
            ->execute();

        return $this->render('AcmeBlogBundle:Blog:list.html.php', array('posts' => $posts));
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getManager()
            ->getRepository('AcmeBlogBundle:Post')
            ->find($id);

        if (!$post) {
```

```
            // hace que se muestre la página de error 404

            throw $this->createNotFoundException();

        }



        return $this->render('AcmeBlogBundle:Blog:show.html.php', array('post' => $post));

    }

}
```

Los dos controladores siguen siendo más o menos pequeños. Cada **uno utiliza la librería ORM de Doctrine para recuperar objetos de la base de datos y el componente `Templating` para generar el código HTML con una plantilla** y devolver un objeto `Response`. **La plantilla `list` ahora es un poco más simple:**

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/lista.html.php -->

<?php $view->extend('::base.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>List of Posts</h1>

<ul>

    <?php foreach ($posts as $post): ?>

    <li>

        <a href="<?php echo $view['router']->generate('blog_show', array('id' => $post->getId())) ?>">

            <?php echo $post->getTitle() ?>

            </a>

    </li>

    <?php endforeach; ?>

</ul>
```

Y el diseño de la plantilla base también es muy parecido al de antes:

```
<!-- app/Resources/views/base.html.php -->

<html>

    <head>

    <title><?php echo $view['slots']->output('title', 'Default title') ?></title>
```

```
    </head>

    <body>

    <?php echo $view['slots']->output('_content') ?>

    </body>

</html>
```

NotaTe vamos a dejar como ejercicio la plantilla `show`, porque debería ser trivial crearla basándote en la plantilla `list`.

Cuando **arranca el motor de Symfony2 (llamado `kernel`), necesita un *mapa* para saber qué controladores ejecutar** en base a la información solicitada por los usuarios. **Este *mapa* se crea con la configuración de enrutamiento**, que proporciona esta información en formato legible:

```
# app/config/routing.yml

blog_list:

    path:     /blog

    defaults: { _controller: AcmeBlogBundle:Blog:list }



blog_show:

    path:     /blog/show/{id}

    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

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

```
<?php

// web/app.php

require_once __DIR__.'/../app/bootstrap.php';

require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

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

$kernel->handle(Request::createFromGlobals())->send();
```

El único trabajo del controlador frontal es iniciar el motor de Symfony2 (`Kernel`) y pasarle un objeto `Request` para que lo manipule. Después, el núcleo de Symfony2 utiliza la información de enrutamiento para determinar qué controlador se ejecuta. Al igual que antes, el método controlador es el responsable de devolver el objeto `Response` final. Y eso es todo lo que hace el controlador frontal.

2.3.4. Qué más ofrece Symfony2

En los siguientes capítulos, aprenderás más acerca de cómo funciona cada parte de Symfony y la organización recomendada de un proyecto. Ahora vamos a ver cómo nos ha facilitado nuestro trabajo de programadores el migrar el blog de PHP simple a Symfony2:

- Tu aplicación cuenta con **código claro y bien organizado** (aunque Symfony no te obliga a ello). Esto facilita la **reutilización** de código y permite a los nuevos desarrolladores ser productivos en el proyecto con mayor rapidez.
- El 100% del código que escribes es para tu aplicación. **No necesitas desarrollar o mantener herramientas de bajo nivel** como la carga automática de clases, el enrutamiento o los controladores.
- Symfony2 te proporciona **acceso a herramientas de software libre** tales como Doctrine, plantillas, seguridad, formularios, validación y traducción (por nombrar algunas).
- La aplicación ahora dispone de **URLs totalmente flexibles** gracias al componente `Routing`.
- La arquitectura centrada en HTTP de Symfony2 te da acceso a herramientas muy potentes, como por ejemplo la **caché HTTP** (mediante la **caché HTTP interna de Symfony2** o mediante herramientas más avanzadas como Varnish). Esto se explica más adelante en el capítulo sobre la caché.

Y lo mejor de todo, utilizando Symfony2, ¡ahora tienes acceso a un conjunto de herramientas de **software libre de alta calidad desarrolladas por la comunidad Symfony2**!

Puedes encontrar una buena colección de herramientas comunitarias de Symfony2 en KnpBundles.com.

2.4. Mejorando las plantillas

Si quieres utilizarlo, Symfony2 incluye de serie un motor de plantillas llamado Twig que hace que las plantillas se escriban más rápido y sean más fáciles de leer.

En otras palabras, la aplicación de ejemplo desarrollada anteriormente podría incluso tener menos código. Considera por ejemplo la siguiente plantilla `list` reescrita con Twig:

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

{% extends "::base.html.twig" %}

{% block title %}List of Posts{% endblock %}

{% block body %}

    <h1>List of Posts</h1>

    <ul>

        {% for post in posts %}

        <li>

            <a href="{{ path('blog_show', { 'id': post.id }) }}">

                {{ post.title }}

                </a>

        </li>

        {% endfor %}

    </ul>
```

```
{% endblock %}
```

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

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



<html>

    <head>

    <title>{% block title %}Default title{% endblock %}</title>

    </head>

    <body>

    {% block body %}{% endblock %}

    </body>

</html>
```

Twig está completamente integrado en Symfony2. Aunque Symfony2 siempre soportará las plantillas PHP, en los siguientes capítulos se van a seguir presentando algunas de las muchas ventajas de Twig respecto a las plantillas PHP. Para más información, consulta el capítulo sobre las plantillas.