

# Symfony

```
<h1>
{{ page_title }} </h1>
{% endfor %}
<ul id="navigation">
<a href="{{ item.href }}"
</title> </head> <% for item in navigation %>
<li>
```

Framework php  
orientado a objetos

CATEGORIA   
**PROGRAMACIÓN**



NIVEL   
**INTERMEDIO**



# GUÍA SYMFONY 2

Versión 1 / septiembre 2011

Nivel: Básico / Intermedio

La Guía Symfony se encuentra en línea en:

<http://www.maestrosdelweb.com/editorial/guia-symfony>

Un proyecto de Maestros del Web

- ⇒ Autores: Maycol Alvarez y Juan Ardissoni
- ⇒ Edición: Eugenia Tobar
- ⇒ Diseño y diagramación: Iván E. Mendoza

Este trabajo se encuentra bajo una licencia Creative Commons

Atribución-NoComercial-CompartirIgual 3.0 Unported (CC BY-NC-SA 3.0)

## CONTACTO

[✉ http://www.maestrosdelweb.com/sitio/correo/](mailto:correo@maestrosdelweb.com)

## REDES SOCIALES

[Facebook: http://www.facebook.com/maestrosdelweb](http://www.facebook.com/maestrosdelweb)

[Twitter: http://www.twitter.com/maestros](http://www.twitter.com/maestros)

### Juan Ardissono

@juanardissono



Paraguayo, analista de sistemas, desarrollador Web y parte del plantel de profesores del Instituto de Diseño y Tecnología del país. Utiliza (X)HTML, PHP5, JavaScript, CSS y Symfony Framework desde su primera versión. Apasionado por PHP y por popularizar su utilización aplicando los estándares y patrones de diseño actuales, para continuar siendo uno de los lenguajes más utilizados dentro de la Web.

@juanardissono  
[www.micayael.com](http://www.micayael.com)

### Maycol Alvarez

@maycolalvarez



Venezolano, Desarrollador de Software bajo PHP, HTML, Javascript, CSS; T.S.U. en Informática Egresado del I.U. “Jesús Obrero” de Catia Caracas y Estudiante Actual de la U.N.E. “Simón Rodríguez”, fanático del Framework Symfony desde 1.4, jQuery, Android y de las tecnologías abiertas GNU/Linux.

@maycolalvarez  
[maycolalvarez.com](http://maycolalvarez.com)



# ÍNDICE

1   Sobre la guía .....	2
2   Autores.....	3
3   Capítulo 1: Introducción.....	5
4   Capítulo 2: El proyecto y los Bundles .....	14
5   Capítulo 3: Creando páginas con Symfony 2.....	25
6   Capítulo 4: Sistema de Routing .....	34
7   Capítulo 5: Definición de rutas con comodines .....	40
8   Capítulo 6: El Controlador .....	45
9   Capítulo 7: La Vista y Twig .....	52
10   Capítulo 8: Configurando nuestra Base de Datos.....	62
11   Capítulo 9: Manipulando datos con Doctrine .....	72
12   Capítulo 10: Validación de datos y creación de formularios .....	85
13   Capítulo 11: Integrando AJAX.....	99
14   Capítulo 12: Integrando jQuery.....	101
15   Capítulo 13: Instalando Bundles de Terceros .....	110
16   Capítulo 14: Seguridad de acceso .....	120
17   Más guías de Maestros del web.....	139

## CAPÍTULO 1: INTRODUCCIÓN

Symfony es un framework PHP basado en la arquitectura MVC (Model-View-Controller). Fue escrito desde un origen para ser utilizado sobre la versión 5 de PHP ya que hace un uso amplio de la orientación a objetos que caracteriza a esta versión y desde la versión 2 de Symfony se necesita mínimamente PHP 5.3. Fue creado por una gran comunidad liderada por [Fabien Potencier](http://fabien.potencier.org/about)<sup>1</sup>, quién a la fecha, sigue al frente de este proyecto con una visión fuertemente orientada hacia las mejores prácticas que hoy en día forman parte del estándar de desarrollo de software.

Por más que Symfony puede ser utilizado para otros tipos de desarrollos no orientados a la Web, fue diseñado para optimizar el desarrollo de aplicaciones Web, proporcionando herramientas para agilizar aplicaciones complejas y guiando al desarrollador a acostumbrarse al orden y buenas prácticas dentro del proyecto.

El concepto de Symfony es no **reinventar la rueda**, por lo que reutiliza conceptos y desarrollos exitosos de terceros y los integra como librerías para ser utilizados por nosotros. Entre ellos encontramos que integra plenamente uno de los frameworks ORM más importantes dentro de los existentes para PHP llamado [Doctrine](http://www.doctrine-project.org/)<sup>2</sup>, el cual es el encargado de la comunicación con la base de datos, permitiendo un control casi total de los datos sin importar si estamos hablando de MySQL, PostgreSQL, SQL server, Oracle, entre otros motores ya que la mayoría de las secuencias SQL no son generadas por el programador sino por el mismo Doctrine.

Otro ejemplo de esto es la inclusión del framework [Twig](http://www.twig-project.org/)<sup>3</sup>, un poderoso motor de plantillas que nos da libertad de separar el código PHP del HTML permitiendo una amplia gama de posibilidades y por sobre todo un extraordinario orden para nuestro proyecto.

Gracias al lenguaje YAML, competidor del XML, tenemos una gran cantidad de configuración totalmente separada del código permitiendo claridad como lo iremos viendo en los demás capítulos. Cabe mencionar que en caso de no querer trabajar con YAML también podemos usar estos archivos de configuración con XML o PHP.

1 <http://fabien.potencier.org/about>

2 <http://www.doctrine-project.org/>

3 <http://www.twig-project.org/>

Contamos con las instrucciones de consola denominadas tasks (tareas), que permiten ejecutar comandos en la terminal diciéndole a Symfony que nos genere lo necesario para lo que le estamos pidiendo, como por ejemplo podría ser la generación completa de los programas necesarios para crear ABMs, tarea que suele ser muy tediosa para los programadores ya que siempre implica mucho código para realizar la misma idea para diferentes tablas.

Otra de las funcionalidades más interesantes, es que contiene un subframework para trabajar con formularios. Con esto, creamos una clase orientada a objetos que representa al formulario HTML y una vez hecho esto simplemente lo mostramos y ejecutamos. Es decir que no diseñamos el formulario con HTML sino que lo programamos utilizando herramientas del framework. Esto nos permite tener en un lugar ordenados todos los formularios de nuestra aplicación incluyendo sus validaciones realizadas en el lado del servidor, ya que symfony implementa objetos validadores muy sencillos y potentes para asegurar la seguridad de los datos introducidos por los usuarios.

Contamos con un amplio soporte para la seguridad del sitio, que nos permite despreocuparnos bastante de los ataques más comunes hoy en día existentes como ser SQL Injection, XSS o CSRF. Todos estos ataques ya tienen forma de prevenir, por lo tanto, dejémosle a Symfony preocuparse por ellos y enfoquemos nuestra atención en los ataques podríamos realizar por un mal uso de nuestra lógica de negocio.

Logramos una aplicación (sitio Web) donde **todo** tiene su lugar y donde el mantenimiento y la corrección de errores es una tarea mucho más sencilla.

Contamos con un gran número de librerías, herramientas y helpers que nos ayudan a desarrollar una aplicación mucho más rápido que haciéndolo de la manera tradicional, ya que muchos de los problemas a los que nos enfrentamos ya fueron pensados y solucionados por otras personas por lo tanto ¡¡¡dediquémonos a los nuevos problemas que puedan surgir!!!

Sobre todo esto hablaremos en los siguientes capítulos pero antes entendamos un poco más sobre el concepto de la arquitectura MVC y otros muy interesantes.

**NOTA:** Si ya has trabajado con versiones anteriores de Symfony o ya entiendes el concepto de Arquitectura MVC, Frameworks ORM y Motores de Plantillas puedes ir directamente a la sección “Documentación oficial” de este capítulo para conocer más sobre la documentación de Symfony y comenzar a descargar el Framework.

# ENTENDIENDO LA ARQUITECTURA MVC

El término MVC proviene de tres palabras que hoy en día se utilizan mucho dentro del ambiente de desarrollo de software: Model - View - Controller, lo que sería en castellano Modelado, Vista y Controlador. Esta arquitectura permite dividir nuestras aplicaciones en tres grandes capas:

- ④ **Vista:** Todo lo que se refiera a la visualización de la información, el diseño, colores, estilos y la estructura visual en sí de nuestras páginas.
- ④ **Modelado:** Es el responsable de la conexión a la base de datos y la manipulación de los datos mismos. Esta capa está pensada para trabajar con los datos como así también obtenerlos, pero no mostrarlos, ya que la capa de presentación de datos es la **vista**.
- ④ **Controlador:** Su responsabilidad es procesar y mostrar los datos obtenidos por el Modelado. Es decir, este último trabaja de intermediario entre los otros dos, encargándose también de la lógica de negocio.

Veamos una imagen para tratar de entenderlo mejor:

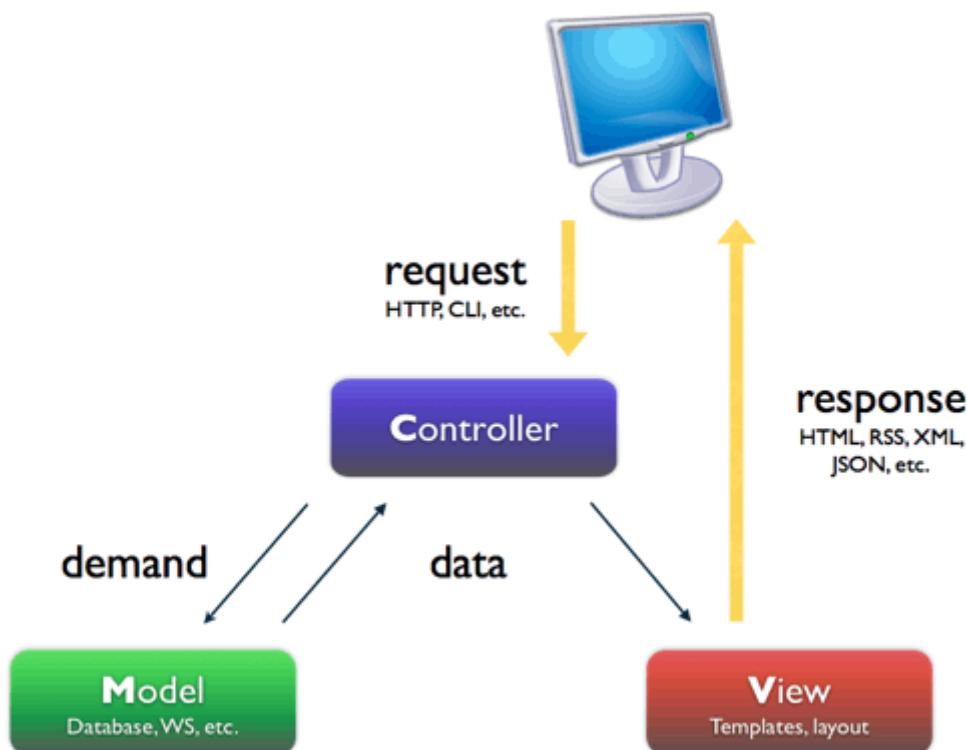


Imagen tomada del manual oficial de Symfony

El cliente envía una señal llamada REQUEST o Petición, ésta es interceptada por el Controlador quien realiza las validaciones necesarias, procesamiento de dichos datos y lógica de negocio asociadas a esa petición del cliente. El Controlador envía datos al Modelado, por ejemplo para ser guardados en una base de datos y/o los obtiene dependiendo de la solicitud del usuario para finalmente enviarlos a la Vista a fin de ser mostrador nuevamente al cliente a través de un RESPONSE o respuesta.

Symfony es un framework totalmente basado sobre la arquitectura MVC por lo que veremos poco a poco como se implementan estos conceptos.

## QUÉ ES UN FRAMEWORK ORM

La siglas ORM provienen de Object-Relational mapping o Mapeo entre Objetos y Relaciones. Este framework es el encargado de tratar con nuestra base de datos desde la conexión, generación de SQL, manipulación de datos, transacciones y desconexión. Cuando hablamos de motores de base de datos se dice que cada tabla es una relación, de ahí el nombre de base de datos relacionales, lo que implica que las tablas se encuentran relacionadas entre sí.

Cuando hablamos de una aplicación orientada a objetos decimos que tratamos con objetos y no con tablas. Cuando agregamos un registro a la tabla de personas por ejemplo, en realidad decimos que agregamos un nuevo objeto Persona. Cuando decimos que un país esta relacionado a varias personas, estamos diciendo que un objeto País contiene un colección de objetos Persona.

Para esto, lo que hacemos es crear clases que mapean cada relación de la base de datos y en lugar de hablar directamente con la base de datos, nosotros los programadores, hablamos con los objetos y Doctrine se encargará de traducir lo necesario para hablar con la base de datos.

Con esto logramos una abstracción casi del 100% con relación al motor de base de datos, sin importar cual sea, ya que hoy en día la mayoría de ellos se encuentran soportados por Doctrine. Symfony toma el framework Doctrine y lo incorpora dentro de sí mismo, proporcionándonos todo el soporte necesario para utilizarlo sin preocuparnos por la configuración del mismo.

## UTILIZANDO UN MOTOR DE PLANTILLAS

Nos hemos acostumbrado a escribir código PHP en el mismo archivo donde se encuentra la estructura HTML de la página. La idea de un motor de plantillas es justamente separar esto en dos capas. La primera sería el programa con la lógica de negocio para resolver el problema específico de esa

página, mientras que la otra sería una página que no contenga el mencionado código sino solo lo necesario para mostrar los datos a los usuarios.

Una vez que hemos solucionado la lógica necesaria, ya sea ejecutando condiciones, bucles, consultas a bases de datos o archivos, etc. tendríamos que guardar los datos que finalmente queremos mostrar en variables y dejar que el motor de plantillas se encargue de obtener la plantilla con el HTML necesario y mostrar el contenido de las variables en sus respectivos lugares.

Esto nos permite, en un grupo de desarrollo dejar la responsabilidad de la capa de diseño al diseñador y la programación de la lógica al programador.

Existen varios motores de plantillas dentro del mundo de PHP hoy en día. Ya hace un buen tiempo, Fabien Potencier, líder del proyecto Symfony, realizó pruebas con relación a los motores de plantillas existentes en el mercado y el resultado lo publicó en su blog bajo el título [Templating Engines in PHP<sup>1</sup>](#). Se puede ver ahí que tras muchas pruebas y análisis el framework de plantillas [Twig<sup>2</sup>](#) es adoptado dentro de la nueva versión de Symfony.

## DOCUMENTACIÓN OFICIAL

Symfony cuenta hoy en día con dos ramas estables. La versión 1.4 es la última de la primera generación y el 28 de julio de 2011 se ha lanzado oficialmente la versión 2 creando una nueva rama. Hay bastante diferencia entre la rama 1.4 y la 2, por lo que este manual está basado en la nueva versión como una introducción a la misma.

Es bueno tener en cuenta que gran parte de la potencia de Symfony, siempre ha sido la excelente documentación publicada, ya que esto forma parte del éxito del aprendizaje.

En el [sitio oficial<sup>3</sup>](#) encontramos 4 excelentes libros sobre Symfony que van mejorando día a día por el equipo mismo de desarrolladores y documentadores.

- ➊ [Quick Tour<sup>4</sup>](#): Guía de introducción a la versión 2 de Symfony.
- ➋ [The Book<sup>5</sup>](#): Libro oficial completo con todas las funcionalidades.

<sup>1</sup> <http://fabien.potencier.org/article/34/templating-engines-in-php>

<sup>2</sup> <http://www.twig-project.org/>

<sup>3</sup> <http://www.symfony.com/>

<sup>4</sup> [http://symfony.com/doc/2.0/quick\\_tour/index.html](http://symfony.com/doc/2.0/quick_tour/index.html)

<sup>5</sup> <http://symfony.com/doc/2.0/book/index.html>

- ➊ [The Cookbook<sup>1</sup>](#): Una recopilación de varios artículos específicos sobre ciertos puntos interesantes. Muy útil después de haber leído los dos primeros libros.
- ➋ [Glossary<sup>2</sup>](#): Una glosario de palabras que podrían ser sumamente útil entenderlas bien durante la lectura de los demás.

Si quieras dar un vistazo a los libros, los recomiendo en el orden en que los explico arriba. También existe una amplia documentación sobre Doctrine<sup>3</sup> y Twig<sup>4</sup> en sus sitios oficiales. Al igual que Symfony ambos son dependientes de la compañía SensioLabs.

## DESCARGANDO LA VERSIÓN 2 DE SYMFONY

Podremos descargar diferentes versiones de Symfony desde el sitio oficial. La primera es la versión estándar completa comprimida en formato .zip y .tgz y la otra es una versión que no contiene los vendors también en ambos formatos comprimidos. Más adelante entenderemos que son los vendors por lo que descargaremos directamente la versión estándar completa en cualquiera de los dos formatos de compresión.

---

**NOTA:** Para este manual se usará la última versión disponible de wampserver utilizando Windows como Sistema Operativo, habiéndolo instalado en C:\wamp\.

---

**NOTA:** Para este manual la última versión de Symfony se encuentra en 2.0.9.

## INSTALANDO EL FRAMEWORK

Una vez que hayamos descargado el archivo, lo descomprimiremos dentro de nuestro localhost en la carpeta C:\wamp\www\ bajo el nombre de Symfony con lo que tendremos los siguientes archivos y directorios:

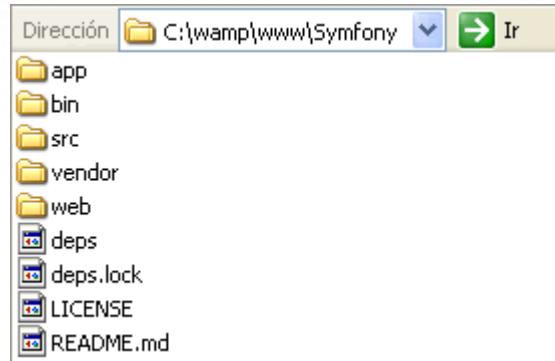
---

1 <http://symfony.com/doc/current/cookbook/index.html>

2 <http://symfony.com/doc/current/glossary.html>

3 <http://www.doctrine-project.org/>

4 <http://www.twig-project.org/>



Estructura del proyecto

Para saber si nuestro servidor cuenta con todo lo necesario para soportar el framework accedemos a la siguiente dirección <http://localhost/Symfony/web/config.php> con la cual veremos la siguiente pantalla:

Welcome!

Welcome to your new Symfony project.

This script will guide you through the basic configuration of your project. You can also do the same by editing the '`app/config/parameters.ini`' file directly.

**1 MAJOR PROBLEMS**

Major problems have been detected and **must** be fixed before continuing

1. Install and enable the **SQLite** or **PDO\_SQLite** extension.

**RECOMMENDATIONS**

Additionally, to enhance your Symfony experience, it's recommended that you fix the following :

1. Install and enable a **PHP accelerator** like APC (highly recommended).
2. Install and enable the **php\_posix** extension (used to colorize the CLI output).
3. Install and enable the **intl** extension.

[Re-check configuration >](#)

Esta pantalla nos mostrará los requerimientos mínimos y las recomendaciones para usar Symfony en nuestro server.

Los requerimientos mínimos son obligatorios solucionarlos por nuestra parte y son mostrados en la

sección marcada en rojo en la imagen anterior. En este caso nos dice que debemos tener instalada y habilitada la extensión para SQLite ya que Symfony hace uso de esta extensión por más que nosotros usemos otro motor de base de datos como MySQL.

Una vez solucionados todos los requerimientos mínimos podemos presionar sobre “Re-check configuration” hasta que no aparezcan más. Nos quedarían las recomendaciones pero ya podemos usar el framework.

Con esto ya podremos ingresar a nuestro sitio: [http://localhost/Symfony/web/app\\_dev.php](http://localhost/Symfony/web/app_dev.php) y ver la pantalla de bienvenida diciéndonos que la instalación está correcta y felicitándonos por el arduo trabajo de instalación.

The screenshot shows the Symfony welcome page. At the top left is the Symfony logo. A search bar with a magnifying glass icon and an 'OK' button are at the top right. The main heading is 'Welcome!'. Below it, a message says 'Congratulations! You have successfully installed a new Symfony application.' There are three large buttons: 'READ THE QUICK TOUR' with a book icon, 'CONFIGURE' with a video camera icon, and 'RUN THE DEMO' with a laptop and wrench icon. At the bottom, there are two sections: 'Documentation' (The book, The cookbook, Glossary) and 'Community' (IRC channel, Mailing lists, Forum). The footer contains server information: 'sf 2.0.0-DEV PHP 5.3.5 | xdebug|accel' and a URL 'app/dev/debug/4e3c06ec19c6b'. It also shows the current controller and action: 'WelcomeController::indexAction|\_welcome|html|200;text/html'.

Página de bienvenida

## RESUMEN DE CAPÍTULO

En primer lugar hablamos un poco sobre Symfony como framework de desarrollo de aplicaciones Web, viendo algunas de las muchas ventajas y aclaramos que la versión utilizada para este manual sera la versión 2.0.x.

Vimos también una introducción sobre ciertos conceptos sumamente importantes para entender la forma de trabajo que tendremos en los siguientes capítulos, hablando sobre la arquitectura Model-View-Controller, la idea de usar un framework ORM para comunicación con la base de datos y las bondades de trabajar con un motor de plantillas como Twig.

Hemos hablado también sobre que Symfony publica mucha información como documentación oficial que siempre se mantiene actualizada y por último, hemos descargado el framework y lo hemos puesto dentro de nuestro localhost para ser accedido con el navegador. Con esto ya hemos realizado la instalación correspondiente.

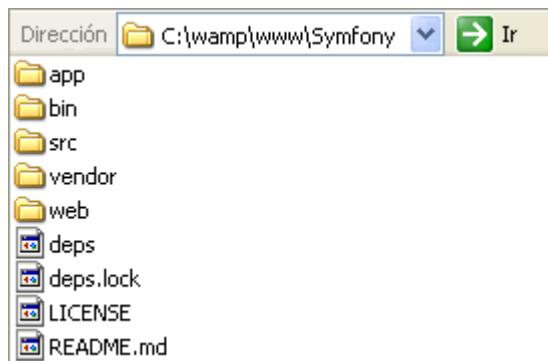
Con esta pequeña introducción daremos un salto directo al siguiente capítulo donde hablaremos de la estructura de directorios de Symfony2 y sobre los Bundles.

## CAPÍTULO 2: EL PROYECTO Y LOS BUNDLES

Ya descargamos el framework Symfony y lo hemos descomprimido en la carpeta correspondiente a nuestro localhost, lo que sería C:\wamp\www\ ya que estamos usando wampserver. Con esto hecho, realizamos nuestra instalación e ingresamos a la página de bienvenida por defecto en http://localhost/Symfony/web/app\_dev.php. Para continuar entremos en más detalles sobre la estructura de nuestro proyecto.

## ESTRUCTURA DE UN PROYECTO SYMFONY

Si vemos el contenido de nuestro proyecto en C:\wamp\www\Symfony\ vemos los siguientes archivos y carpetas:



Estructura del proyecto

- ④ **app\**: Aquí se encuentra la configuración correspondiente a todo el proyecto. Si ya has trabajado con symfony 1.x es muy importante entender que en la versión 2, por lo general debe existir una sola aplicación por proyecto. El concepto de tener varias aplicaciones en esta nueva versión es conocido por tener varios Bundles.
- ④ **bin\**: Dentro de esta carpeta tenemos el script vendors.sh que se utiliza para actualizar el framework vía consola.
- ④ **src\**: Esta es la carpeta donde irá todo nuestro código y es aquí donde residen los Bundles que básicamente son carpetas que representan nuestras aplicaciones.
- ④ **vendor\**: En esta carpeta se encuentran los archivos del framework Symfony y de las demás librerías de terceros como por ejemplo Doctrine, Twig, etc.
- ④ **web\**: En la carpeta web es donde deberán estar los archivos públicos del proyecto como los

javascripts, css, etc. También se encuentran dentro de esta carpeta los controladores frontales que se explican a continuación. Solo estos archivos deberán ser accedidos desde un navegador.

## CONTROLADORES FRONTALES

Es sumamente importante entender que los archivos que no se encuentren dentro de la carpeta web\ no pueden y no deben ser accedidos por el navegador ya que forman parte de la programación interna del proyecto. Por lo tanto nuestras páginas y programas que son guardados dentro de la carpeta src\ no son **directamente accedidos** por el navegador sino a través de los controladores frontales.



Controladores Frontales

Dentro de la carpeta web\ vemos que existen dos archivos: "app.php" y "app\_dev.php". Estos son los archivos llamados controladores frontales y son a través de ellos que accederemos a nuestras páginas. La diferencia entre ambos es que Symfony maneja **entornos**, lo que significa que a través de ambos accedemos a las mismas páginas pero con diferentes configuraciones.

Existen dos entornos configurados por defecto en un proyecto Symfony: **desarrollo y producción**. Hablaremos un poco más sobre entornos en la siguiente sección, mientras tanto sigamos entendiendo los controladores frontales.

Cualquier petición (request) que llegue a la aplicación para solicitar una página específica debe ser sobre nuestros controladores y no directamente a ellas. Esto es debido a que los controladores frontales levantan todas la utilidades necesarias del framework y luego invocan a la página solicitada.

Este es el motivo por el cual en el capítulo anterior, pudimos acceder directamente a la dirección <http://localhost/Symfony/web/config.php> para comprobar nuestro servidor ya que es una página dentro de la carpeta web\ pero, a la hora de ingresar ya a nuestra aplicación ingresamos usando el controlador frontal [http://localhost/Symfony/web/app\\_dev.php](http://localhost/Symfony/web/app_dev.php).

## ENTORNOS

Symfony ya trae configurado dos entornos muy necesarios, **desarrollo** y **producción**. La diferencia entre ambos es con relación a la configuración. El entorno de **desarrollo** está configurado para brindar ayuda al desarrollador, mientras que el entorno de **producción** está optimizado para los usuarios finales del sitio. Dicho de otra manera, mientras estemos trabajando con la construcción y programación de nuestro sitio Web accedemos a las páginas a través del entorno de desarrollo pero, una vez que lo hayamos subimos a un hosting y lo disponibilizamos a los usuarios finales las páginas deberán ser accedidas por medio del entorno de producción.

Para probarlo puedes hacer lo siguiente. Si ingresas a esta dirección `http://localhost/Symfony/web/app_dev.php` ves la página de bienvenida desde el entorno de desarrollo. Si quieres acceder a la misma desde el entorno de producción usas el controlador frontal correspondiente al mismo `http://localhost/Symfony/web/app.php`.

---

**NOTA:** En caso de que al ingresar a la URL correspondiente al entorno de producción salga un error puedes probar borrando el contenido de la carpeta `app\cache\`.

## LA CACHE DE SYMFONY

Una de las configuraciones más interesantes de ambos entornos sería con relación a que Symfony maneja una cache donde realiza una especie de pre-compilación de las páginas. Como Symfony maneja tantos archivos y formatos como YAML, XML, Twig y PHP, al momento de ingresar por primera vez al sitio, toma todos los archivos y los convierte a PHP guardándolos dentro de la carpeta `app\cache\`. Esto se hace para que no se pierda tiempo generando todo por cada página solicitada.

Una vez realizado esto, simplemente las páginas son accedidas por medio de la cache, razón por la cual la primera vez que se ingresa al sitio tardará un poco más que las siguientes y cada vez que se hayan realizado cambios sobre estos archivos debemos borrar la cache para que Symfony la vuelva a generar.

Para el entorno de **desarrollo** la cache se genera por cada petición de las páginas sin necesidad de que el **programador** tenga que borrarla a mano mientras que en el entorno de producción lo debemos hacer nosotros mismos ya que la idea es mantenerla para ganar rapidez. Para borrar la cache podemos simplemente eliminar el contenido de la carpeta `app\cache\`.

## SYMFONY PROFILER

Accediendo al entorno de desarrollo también podremos ver una barra en la parte inferior de la página llamada “Symfony profiler” que nos da información actualizada por cada request sobre varias cosas útiles para el desarrollador como parámetros del request, sentencias SQL ejecutadas, tiempos transcurridos, datos de sesión, etc. Por supuesto esta barra se encuentra deshabilitada en el entorno de producción.



## MENSAJES DE ERROR

Como un punto de seguridad también es importante saber que en el entorno de desarrollo, Symfony nos mostrará mucha más información de los errores producidos. Por ejemplo si intentamos ingresar a una página no existente en el entorno de desarrollo se nos mostrará un StackTrace completo mientras que en el entorno de producción simplemente dirá: Error 404 - Página no encontrada.

The screenshot shows the Symfony development toolbar at the top of a browser window. Below it, a modal dialog box is displayed. The dialog has a green speech bubble icon on the left containing the text "Exception detected!" and a small cartoon ghost icon below it. The main content area contains the error message: "No route found for "GET /noexiste"" followed by the stack trace: "404 Not Found - NotFoundHttpException  
1 linked Exception: NotFoundHttpException >". The stack trace details the error: "[22] NotFoundHttpException: No route found for "GET /noexiste"" and points to line 1597 of the file "C:\wamp\www\Symfony\app\cache\dev\classes-cdc01.php". The code snippet shown is:

```

1594.         if ($null !== $this->logger) {
1595.             $this->logger->err($message);
1596.         }
1597.         throw new NotFoundHttpException($message, $e);

```

Mensajes de error en entorno de Desarrollo

## Oops! An Error Occurred

The server returned a "404 Not Found".

Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.

Mensajes de error en entorno de Producción

## ¿QUÉ SON LOS BUNDLES?

Ya hemos venido nombrando esta palabra dentro de este capítulo así que hablemos un poco más detalladamente. Un Bundle es básicamente una carpeta que contiene los archivos necesarios para un grupo de funcionalidades específicas, como por ejemplo un blog, un carrito de compras o hasta el mismo frontend y backend de nuestra aplicación. La idea es que yo debería trasladar este Bundle a otro proyecto y reutilizarlo si quiero.

---

**NOTA:** Para los que hayan trabajado con la versión 1.x de Symfony, un Bundle es una mezcla entre las aplicaciones y los plugins ya que este es el motivo por el cual decíamos que a partir de la versión 2 un proyecto debería tener una sola aplicación y no varias como anteriormente era normal, debido a que para este concepto existen los Bundles. Con relación a los Plugins, estos deberán ser reescritos como Bundles.

Una aplicación en Symfony2 podrá contener todos los Bundles que queramos y necesitemos, simplemente debemos crearlos y registrarlos. Los Bundles que nosotros creemos deberán ir dentro de la carpeta src\ del proyecto mientras que los Bundles de terceros deberán ir dentro de la carpeta vendor\.

Un Bundle tiene una estructura de carpetas y archivos definidos y un nombre identificador dentro de nuestro proyecto que lo utilizaremos varias veces para hacer referencia al mismo. Como ya vimos, nuestros bundles se guardarán dentro de la carpeta src\, y dentro de esta carpeta se almacenan los bundles que podría llamarse por ejemplo FrontendBundle, BlogBundle, CarritoBundle, etc. Lo ideal es no guardar directamente los bundles dentro src\ sino dentro de una carpeta que represente a la empresa o a nosotros a la cual llamamos paquete, esto a fin de que si alguien más crea un BlogBundle no se confunda con el nuestro.

Por ejemplo, podríamos crear un bundle para nuestro manual de Maestros del Web creando un

paquete MDW\ y dentro de este un bundle con nombre DemoBundle (sufijo Bundle obligatorio). Aquí crearemos todo nuestro código de ejemplo.

La versión estándar de Symfony2 viene ya con un Bundle de ejemplo llamado AcmeBundle y es el que se ejecuta al ingresar a [http://localhost/Symfony/web/app\\_dev.php](http://localhost/Symfony/web/app_dev.php) dándonos la bienvenida. Nosotros crearemos nuestro propio Bundle pero para esto faremos una pequeña modificación en el archivo app\config\routing\_dev.yml en donde buscaremos las siguientes líneas:

```
_welcome:
  pattern: /
  defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

y las reemplazaremos por:

```
_welcome:
  pattern: /bienvenida
  defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

Con esto lo que hicimos fue liberar la dirección [http://localhost/Symfony/web/app\\_dev.php](http://localhost/Symfony/web/app_dev.php) que corresponde a la bienvenida. Ahora para ingresar a esta página se debe escribir [http://localhost/Symfony/web/app\\_dev.php/bienvenida](http://localhost/Symfony/web/app_dev.php/bienvenida) en lugar de la anterior. Esto solo lo hicimos para que podamos usar la dirección URL anterior para nuestro Bundle. Ahora ingresando a la primera dirección debería dar un error 404 ya que no existe página asignada a esa ruta.

## EL COMANDO CONSOLE

Como decíamos, un bundle es simplemente una carpeta que contiene carpetas y archivos. Para no crearlos a mano usaremos una utilidad de Symfony llamada “console”.

Abriremos un cmd y entraremos al directorio de nuestro proyecto con el siguiente comando:

```
C:\>cd wamp\www\Symfony
```

Ahora usaremos la utilidad mencionada para pedirle a Symfony que nos diga que versión del framework se está usando. Esto lo hacemos de la siguiente manera:

```
C:\wamp\www\Symfony>php app\console --version
```

Al ejecutar esto se nos mostrará un texto similar a: Symfony version 2.0.0 - app/dev/debug

**NOTA:** En caso de que no se encuentre el comando “php”, deberá agregar el directorio C:\wamp\bin\php\php5.3.5\ al PATH del Windows, carpeta que contiene el interprete de PHP (php.exe).

Es posible que necesites cerrar y volver a entrar al CMD si haces este cambio.

El archivo app\Console no es nada más que un script PHP que ejecuta varias tareas (tasks) dependiendo de los parámetros que le pasemos como por ejemplo es el parámetro “--version”, que nos devuelve la versión de nuestro framework.

Existen muchas tareas que Symfony puede hacer por nosotros. Para verlas todas simplemente puedes hacerlo ejecutando el script sin pasarle parámetros:

```
C:\wamp\www\Symfony>php app\Console
```

## CREANDO NUESTRO PROPIO BUNDLE

Para crear nuestro MDW\DemoBundle haremos uso del comando “console” de Symfony2 pasándole el parámetro “generate:bundle”. Ejecutemos en el cmd lo siguiente:

```
C:\wamp\www\Symfony>php app\Console generate:bundle
```

Con este comando se ejecutará un generador que nos hará varias preguntas para crear nuestro Bundle como se muestra a continuación:

1. Lo primero que nos pide será el namespace o carpeta contenedora del Bundle para lo que le diremos que deberá estar dentro de una carpeta MDW y el nombre de la carpeta de nuestro bundle será DemoBundle. Esto lo hacemos escribiendo: MDW\DemoBundle.
2. A continuación nos pedirá un nombre identificador del Bundle para el proyecto y nos propone entre corchetes la concatenación MDWDemoBundle. Para aceptar la propuesta daremos enter.
3. A continuación nos preguntará donde queremos que se guarde el nuevo bundle creado. Aceptaremos la propuesta.
4. Nos pide el formato de archivo que usará para las configuraciones del Bundle. Nos propone [annotations] pero le diremos que queremos que sea “yml”.
5. Luego nos pregunta si queremos que nos genere una estructura completa para el bundle y le vamos a decir que “no” ya que necesitamos solo la base.
6. Confirmamos si todo esta bien.

7. Nos pregunta si queremos registrar nuestro Bundle en el archivo app\AppKernel.php a lo que le diremos que si.
8. Nos pregunta si queremos actualizar el archivo app\config\routing.yml y le decimos que si.

```
C:\wamp\www\Symfony>php app\console generate:bundle

Welcome to the Symfony2 bundle generator

Your application code must be written in bundles. This command helps
you generate them easily.

Each bundle is hosted under a namespace <like Acme/Bundle/BlogBundle>.
The namespace should begin with a "vendor" name like your company name, your
project name, or your client name, followed by one or more optional category
sub-namespaces, and it should end with the bundle name itself
(which must have Bundle as a suffix).

Use / instead of \ for the namespace delimiter to avoid any problem.

Bundle namespace: MDW\DemoBundle 1

In your code, a bundle is often referenced by its name. It can be the
concatenation of all namespace parts but it's really up to you to come
up with a unique name (a good practice is to start with the vendor name).
Based on the namespace, we suggest MDWDemoBundle.

Bundle name [MDWDemoBundle]: 2

The bundle can be generated anywhere. The suggested default directory uses
the standard conventions.

Target directory [C:\wamp\www\Symfony\src]: 3

Determine the format to use for the generated configuration.

Configuration format (yml, xml, php, or annotation) [annotation]: yml 4

To help you getting started faster, the command can generate some
code snippets for you.

Do you want to generate the whole directory structure [no]? 5

Summary before generation

You are going to generate a "MDW\DemoBundle\MDWDemoBundle" bundle
in "C:\wamp\www\Symfony\src/" using the "yml" format.

Do you confirm generation [yes]? 6

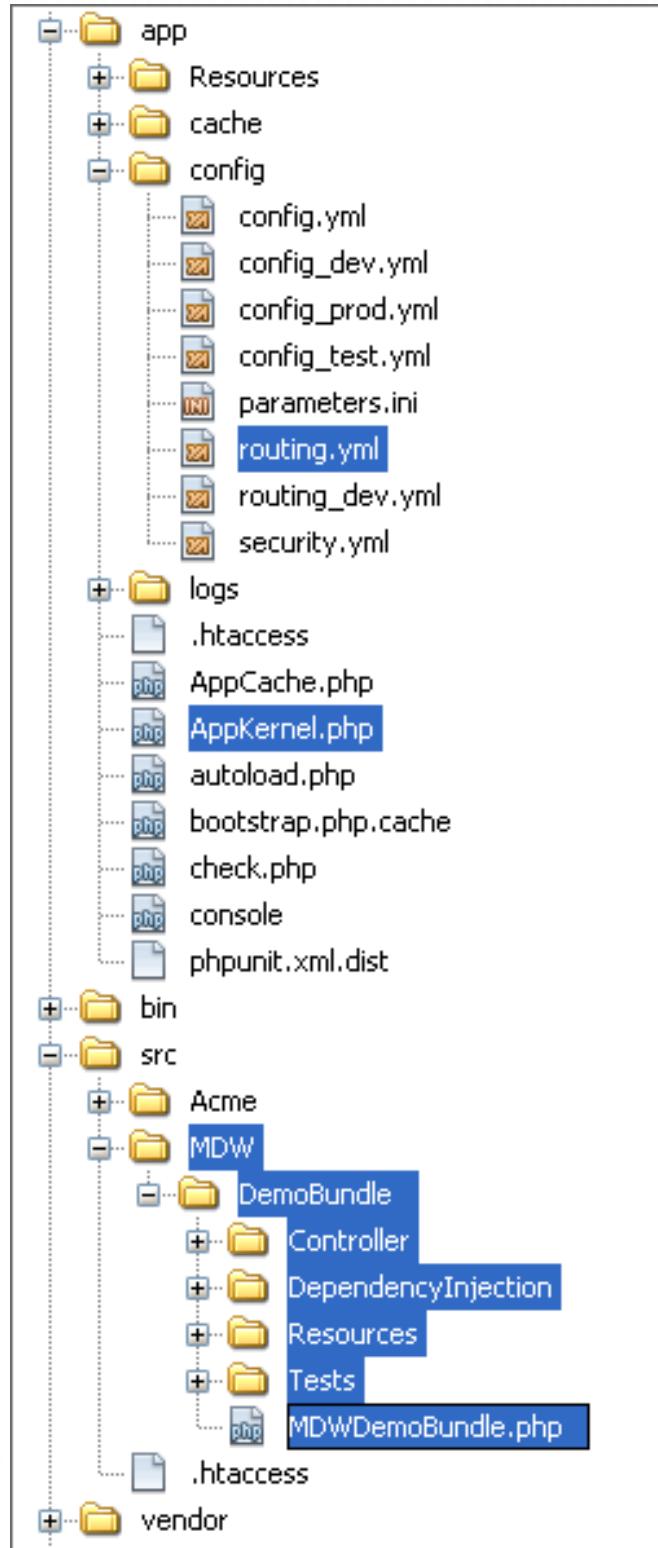
Bundle generation

Generating the bundle code: OK
Checking that the bundle is autoloaded: OK
7
Confirm automatic update of your Kernel [yes]?
Enabling the bundle inside the Kernel: OK
8
Confirm automatic update of the Routing [yes]?
Importing the bundle routing resource: OK

You can now start using the generated code!
```

Generador de Bundles de Symfony2

Con esto ya tenemos nuestro Bundle y lo deberíamos ver de la siguiente manera:



Nuevo Bundle

Ahora ya tenemos dentro de la carpeta src\ una carpeta correspondiente a la empresa llamada MDW y dentro de esta nuestro DemoBundle con sus carpetas y archivos necesarios. Tendremos que recordar que el identificador del bundle será “MDWDemoBundle”.

En el archivo AppKernel.php si lo abrimos, veremos una línea que apunta al archivo MDW\DemoBundle\MDWDemoBundle.php y lo que hace es habilitarlo para el proyecto. Todo Bundle nuevo o de terceros que incluyamos al proyecto deberán ser registrados aquí agregando esta línea.

```
$bundles = array(  
    ...  
    new MDW\ManualBundle\MDWManualBundle(),  
) ;
```

---

**NOTA:** Para los que trabajaron con Symfony 1.x, esto sería similar a habilitar plugins en el archivo ProjectConfiguration.class.php.

Por último vemos que también se agregaron las siguientes líneas al archivo app\config\routing.yml

**MDWManualBundle:**

```
resource: "@MDWManualBundle/Resources/config/routing.yml"  
prefix: /
```

Veremos más sobre el archivo routing.yml en el siguiente capítulo, pero por el momento entraremos a la página de ejemplo que se crea automáticamente al crear un Bundle con la siguiente dirección: [http://localhost/Symfony/web/app\\_dev.php/hello/minombre](http://localhost/Symfony/web/app_dev.php/hello/minombre) donde “minombre” lo podemos reemplazar por nuestro nombre como un parámetro GET. Con esto deberíamos poder ver una página en blanco con el texto Hello {minombre}.

## RESUMEN DE CAPÍTULO

En este capítulo ya hemos entrado a más bajo nivel entendiendo la estructura de un proyecto Symfony. Hemos hablado sobre los controladores frontales y los entornos de desarrollo y producción.

Entendimos uno de los conceptos más importantes de Symfony2 conocido como los Bundles y hemos usado las tareas de Symfony que se ejecutan por medio del script “console”. Por último ya hemos creado un Bundle propio para nuestro manual en donde iremos trabajando con los siguientes capítulos.

## CAPÍTULO 3: CREANDO PÁGINAS CON SYMFONY 2

En este capítulo crearemos todos los ejemplos del curso y veremos como crear las páginas con Symfony 2.

## PASOS PARA CREAR NUESTRAS PÁGINAS

Para crear una página tenemos que tener en cuenta tres pasos:

- 1. Asignación de una ruta:** Una dirección URL asignada a la página para que el controlador frontal la pueda acceder.
- 2. Creación de una acción (action):** La lógica necesaria para la página. Corresponde al Controlador en arquitectura MVC.
- 3. Creación de la plantilla (template):** La estructura de nuestra página. Corresponde a la Vista en arquitectura MVC.

Tomaremos el ejemplo que fue creado al generar nuestro MDWDemoBundle y lo estudiaremos para entender estos tres pasos:

### ASIGNACIÓN DE UNA RUTA

Las rutas se refieren a la dirección URL que utilizará el controlador frontal para acceder a nuestra página. Dichas rutas se especifican en el archivo de configuración `app/config/routing.yml`. Este paso es sumamente importante ya que de lo contrario la página existirá pero no podrá ser accedida.

Dentro de este archivo podríamos crear las rutas directamente, pero para no mezclar las rutas de nuestro Bundle con las de otros, podemos crear las rutas en un archivo dentro de nuestra carpeta `MDW\DemoBundle` y de esta manera logramos independencia y portabilidad. Para hacer esto, tendremos que importar el archivo `routing.yml` de nuestro bundle dentro del `app\config\routing.yml` que básicamente es el archivo de rutas genérico para todo el proyecto.

Al crear nuestro Bundle con el script “console”, en el paso 7 explicado en el capítulo anterior es justamente lo que se hace automáticamente agregando el siguiente código de importación en el archivo

```
app\config\routing.yml:
  MDWDemoBundle:
    resource: "@MDWDemoBundle/Resources/config/routing.yml"
    prefix: /
```

La primera línea es simplemente un texto identificador para la importación, que por convención podríamos usar el mismo identificador de nuestro Bundle. Abajo, con espacios en blanco definimos la clave “resource” que apunta a un archivo externo y haciendo uso de nuestro identificador @ `MDWDemoBundle` que apunta a `src\MDW\DemoBundle`, le decimos que use el archivo ubicado en nuestra carpeta `/Resources/config/routing.yml`.

La segunda clave a definir es el “prefix” que indica con / que a partir del controlador frontal se crean nuestras rutas.

---

NOTA: Si no conoces la forma de uso de los archivos YAML puedes ver un poco de información aquí.

Una vez que tenemos ya nuestro archivo importado lo abriremos y veremos el siguiente contenido de ejemplo que se creó al generar nuestro Bundle con el generador de Symfony:

```
MDWDemoBundle_homepage:
  pattern: /hello/{name}
  defaults: { _controller: MDWDemoBundle:Default:index }
```

El primer texto es nuevamente un texto identificador para esta ruta. Los identificadores no pueden repetirse con ninguna otra ruta del proyecto, en nuestro Bundle o en cualquier otro.

Abajo, en lugar de usar la clave `resource` para importar un archivo, definimos el “pattern” (patrón) que indica la dirección URL a usar para esta ruta. Dentro de una ruta cuando usamos llaves `{name}`, indicamos que será un parámetro. Como la ruta indica la dirección que el Controlador Frontal utilizará, estamos diciendo que podremos acceder a la página escribiendo `http://localhost/Symfony/web/app_dev.php/hello/Jhon` donde:

- ⌚ **`http://localhost`**: Dirección del servidor.
- ⌚ **`/app_dev.php`**: corresponde al controlador frontal, que podríamos utilizar el de desarrollo o producción.

- ② **/hello/Jhon:** Indica la ruta que acabamos de crear, donde {name} lo podremos reemplazar por el valor del parámetro que queramos.

La segunda clave obligatoria es “defaults” que utilizando llaves simulamos un array asociativo por lo que la clave “\_controller” indica cual será el controlador que contendrá la lógica de la página. Para no escribir la ruta completa del archivo (\src\MDW\DemoBundle\Controller\DefaultController.php) utilizamos una forma abreviada o “dirección lógica” que está compuesta de tres partes: Identificado rDelBundle:Controller:Action.

- 4. IdentificadorDelBundle:** En este caso nuestro identificador es MDWDemoBundle.
- 5. Controller:** El nombre de la clase que contendrá los actions (acciones). Estas clases se encuentran en la carpeta “Controller” de nuestros Bundles.
- 6. Action:** Representado por un método de la clase arriba mencionada. Este método contendrá la lógica de negocios para nuestra página y se ejecutará antes de mostrar la página.

Entendiendo los pasos para crear nuestras páginas, vemos que nuestra ruta de ejemplo apunta a MDWDemoBundle:Default:index, lo que indica que la acción a ejecutarse al ingresar a nuestra ruta se encuentra en MDWDemoBundle en una clase DefaultController programado en un método indexAction.

## CREACIÓN DE UN CONTROLADOR (ACCIÓN)

Así como vimos en la sección anterior, una ruta apunta a un controlador usando la clave defaults

```
defaults: { _controller: MDWDemoBundle:Default:index }
```

Si abrimos el archivo mencionado veremos lo siguiente:

```
//-- \src\MDW\DemoBundle\Controller\DefaultController.php
public function indexAction($name)
{
    return $this->render('MDWDemoBundle:Default:index.html.twig',
array('name' => $name));
}
```

Vemos que nuestro método recibe el parámetro \$name que debe coincidir con el escrito en nuestra ruta /hello/{name}.

No hay lógica para esta página ya que es un simple ejemplo, por lo que simplemente termina haciendo un return del resultado de un método render() que se encarga de llamar a la plantilla (paso siguiente) y pasarle datos por medio del segundo argumento, un array asociativo. En caso de que no necesitemos enviar variables a la vista simplemente enviamos un array vacío.

Como primer argumento del método render(), enviamos el “nombre lógico” del template MDWDemoBundle:Default:index.html.twig que indica donde está la plantilla. Las plantillas se encuentran organizadas dentro de la carpeta “Resources\views” de nuestros Bundles, en este caso src\MDW\DemoBundle\Resources\views\Default\index.html.twig.

## CREACIÓN DE LA PLANTILLA

Para la plantilla estamos utilizando el framework Twig y si abrimos el archivo mencionado vemos que simplemente utilizamos la variable \$name como si existiera, esto lo podemos hacer porque al momento de llamar a la plantilla desde el controlador enviamos esta variable para que exista mágicamente usando el método render(): array('name' => \$name)

Si abrimos el archivo src\MDW\DemoBundle\Resources\views\Default\index.html.twig vemos que solo contiene como ejemplo: Hello {{ name }}!

Como habíamos hablado en el capítulo 1, Twig permite separar el código PHP del HTML por lo que por ejemplo en lugar de escribir:

```
Hello <?php echo htmlentities($name) ?>!
```

podemos simplemente escribir:

```
Hello {{ name }}!
```

donde {{ \$var }} significa que la variable que este dentro será impresa como si utilizaramos un echo.

## CREEMOS NUESTRA PRIMERA PÁGINA DE EJEMPLO

Siguiendo las mismas instrucciones vistas en los puntos anteriores, crearemos una página de ejemplo para mostrar un listado de artículos de un blog en una tabla. Supondremos que los artículos son extraídos de una base de datos pero como todavía no hemos llegado a hablar de Doctrine los obtenemos de un array.

## EJEMPLO 1

El primer paso es pensar en una dirección URL para acceder a la página, luego crearemos el action que se procesará al ejecutarse la petición de la URL y finalmente usaremos los datos devueltos por nuestro action dentro del template para mostrar la tabla de artículos.

### ► PASO 1

---

Dentro del archivo `src\MDW\DemoBundle\Resources\config\routing.yml` de nuestro Bundle de ejemplo crearemos la ruta correspondiente agregando el siguiente código:

```
articulos:  
    pattern: /articulos  
    defaults: { _controller: MDWDemoBundle:Default:articulos }
```

Como habíamos mencionado la primera línea corresponde al identificador de la ruta al que llamaremos “articulos”. Este nombre será usado para mostrar los links a nuestras páginas pero esto lo veremos con más detalles más adelante.

A continuación definimos el pattern que será lo que agregaremos a la dirección del controlador frontal. Para este caso definimos que si nuestra dirección de acceso al controlador frontal es “`http://localhost/Symfony/web/app_dev.php`” agregaremos “/articulos” para acceder a nuestra página teniendo como URL: `http://localhost/Symfony/web/app_dev.php/articulos`.

Por último definimos el action que se ejecutará al llamar a esta ruta:

`MDWDemoBundle:Default:articulos` lo que apuntaría al método `articulosAction()` de la clase `DefaultController.php` que existe dentro de nuestro `MDWDemoBundle`.

### ► PASO 2

---

Ahora que ya tenemos nuestra ruta apuntando a nuestro action nos encargaremos de crear el método correspondiente que será ejecutado y que contendrá la lógica del negocio, que para este caso es muy simple. Dentro de nuestra clase `\src\MDW\DemoBundle\Controller\DefaultController.php` agregaremos el método (action) correspondiente que deberá llamarse `articulosAction()`:

```
public function articulosAction()  
{
```

```

    //-- Simulamos obtener los datos de la base de datos cargando los
artículos a un array
$articulos = array(
    array('id' => 1, 'title' => 'Articulo numero 1', 'created' =>
'2011-01-01'),
    array('id' => 2, 'title' => 'Articulo numero 2', 'created' =>
'2011-01-01'),
    array('id' => 3, 'title' => 'Articulo numero 3', 'created' =>
'2011-01-01'),
);
return $this->render('MDWDemoBundle:Default:articulos.html.twig',
array('articulos' => $articulos));
}

```

Una vez que tenemos los datos dentro de nuestro array, por medio del método `->render()` llamamos a nuestro template con el primer argumento y con el segundo pasamos los datos que deberán existir dentro del mismo.

## ► PASO 3

---

El último paso será usar la información que nuestro action nos provee para generar nuestra vista y mostrarla al usuario. Para esto, crearemos un archivo dentro de la carpeta \Resources\views\Default de nuestro bundle con el nombre articulos.html.twig y con la ayuda de Twig recorreremos nuestro array de artículos para mostrar nuestra tabla de una manera muy sencilla:

```

<h1>Listado de Articulos</h1>
<table border="1">
<tr>
    <th>ID</th>
    <th>Titulo</th>
    <th>Fecha de Creacion</th>
</tr>
{% for articulo in articulos %}
<tr>
    <td>{{articulo.id}}</td>
    <td>{{articulo.title}}</td>
    <td>{{articulo.created}}</td>

```

```
</tr>
{%
endfor %}
</table>
```

Twig nos provee mucha habilidad para manipular los datos sin escribir código PHP pudiendo acceder a las claves de nuestro array por medio de articulo.id o articulo['id'] indistintamente.

Con esto ya tenemos una tabla que muestra artículos haciéndolo en tres pasos bien concisos y donde cada uno tiene su responsabilidad. Para ingresar a la página podemos escribir esta URL: [http://localhost/Symfony/web/app\\_dev.php/articulos](http://localhost/Symfony/web/app_dev.php/articulos)

## EJEMPLO 2

Ahora que ya hemos visto como mostrar los datos de nuestro array supongamos que queremos otra página que reciba como parámetro GET un id de artículo y nos muestre sus datos. Para esto nuevamente sigamos los mismo pasos

### ► PASO 1

---

Para este caso crearemos una ruta llamada “articulo” que recibirá un parámetro {id} y hará que se ejecute el action MDWDemoBundle:Default:articulo:

```
articulo:
  pattern: /articulo/{id}
  defaults: { _controller: MDWDemoBundle:Default:articulo }
```

### ► PASO 2

---

Para simular la base de datos volveremos a tener nuestro array y buscaremos el id recibido. Como recibimos un parámetro GET podemos recibirlo directamente como argumento de nuestro método:

```
public function articuloAction($id)
{
  //-- Simulamos obtener los datos de la base de datos cargando los
  artículos a un array
  $articulos = array(
    array('id' => 1, 'title' => 'Articulo numero 1', 'created' => '2011-
  01-01'),
```

```
array('id' => 2, 'title' => 'Articulo numero 2', 'created' => '2011-01-01'),
array('id' => 3, 'title' => 'Articulo numero 3', 'created' => '2011-01-01'),
);
//-- Buscamos dentro del array el ID solicitado
$articuloSeleccionado = null;
foreach($articulos as $articulo)
{
    if($articulo['id'] == $id)
    {
        $articuloSeleccionado = $articulo;
        break;
    }
}
//-- Invocamos a nuestra nueva plantilla, pasando los datos
return $this->render('MDWDemoBundle:Default:articulo.html.twig',
array('articulo' => $articuloSeleccionado));
}
```

## ► PASO 3

---

Por último mostramos los datos de nuestro artículo devuelto por nuestro controlador:

```
<h1>Articulo con ID {{articulo.id}}</h1>
<ul>
    <li>Titulo: {{articulo.title}}</li>
    <li>Fecha de creacion: {{articulo.created}}</li>
</ul>
```

Para acceder a este ejemplo podemos escribir la URL: [http://localhost/Symfony/web/app\\_dev.php/articulo/1](http://localhost/Symfony/web/app_dev.php/articulo/1). Donde el parámetro “1” sería el id del artículo que queremos ver.

## RESUMEN DE CAPÍTULO

Vimos los tres pasos básicos para crear nuestras páginas y hemos llegado a la conclusión de que primeramente se accede a una URL definida por nuestras rutas donde el routing.yml deriva la información a un método de un controlador llamado action en donde se procesan los datos y se llama a la vista pasándole como variables los datos que necesitan ser mostrados. Una vez en la vista usaremos el framework Twig para facilitar la visualización de la información.

## CAPÍTULO 4: SISTEMA DE ROUTING

Una de las necesidades más comunes en el desarrollo de sitios profesionales es implementar URLs amigables, así convertimos algo como `/index.php?articulo=1` por algo más cómodo y agradable a la vista del usuario: `/blog/introduccion_symfony2.htm`.

El Routing de Symfony2 nos brinda un sistema de enrutado muy dinámico que deja concentrarnos en el desarrollo de los “caminos” hacia nuestra aplicación, además es bidireccional, lo cual nos permite cambiar la ruta `/blog/introduccion_symfony2.htm` por `/noticia/introduccion_symfony2.htm` editando sólo la definición de la ruta y así evitarnos la tarea de buscar todas las referencias internas hacia ella en nuestra aplicación para hacer el cambio.

El objetivo de éste capítulo es comprender el funcionamiento básico del sistema de Routing de Symfony2, crear e importar rutas, así como configuraciones básicas para permitir rutas flexibles. Nos enfocaremos en la configuración en formato YAML por ser simple y fácil de entender, recuerda que Symfony 2 es un Fw altamente configurable y que puedes utilizar como configuración: XML, PHP y Annotations.

## FUNCIONAMIENTO DEL ROUTING

En la arquitectura del Modelo MVC el encargado de manejar las peticiones Web es el Controlador (Controller), como cualquier aplicación consiste en varios Controllers y necesitamos un punto de acceso centralizado para distribuir las peticiones de nuestra aplicación a los Controllers indicados, a ese punto lo llamamos el Controlador frontal (Front Controller) que generalmente corresponde al archivo raíz de nuestra web, es decir el `app.php` o `index.php` (dependiendo de la configuración del servidor HTTP) y para ello necesitamos redirigir a éste todas las peticiones por medio de un `.htaccess` (en el caso de Apache):

```
# web/.htaccess
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ app.php [QSA,L]
```

En Symfony2 el **Front Controller** se encarga de cargar el kernel del Framework, el cual recibe nuestra petición HTTP (request) y pasa la URL al sistema de Routing, donde es procesada (comparada con las rutas definidas) y se ejecuta el Controller definido; este es el comportamiento básico del Routing: `empatar URL y ejecutar los Controllers.`

## CÓDIGO DEL FRONT CONTROLLER:

```
<?php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
//require_once __DIR__.'/../app/AppCache.php';
use Symfony\Component\HttpFoundation\Request;
$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
//$kernel = new AppCache($kernel);
$kernel->handle(Request::createFromGlobals())->send();
```

Notarás que Symfony2 tiene en principio 2 Front Controllers: `web/app.php` y `web/app_dev.php`, esto se debe a que Symfony2 maneja por cada controlador frontal un “Entorno” (Environment) que le permite ajustar la configuración interna según la situación en que se encuentre nuestra aplicación, `app.php` para la Producción y `app_dev.php` para el desarrollo.

De ahora en adelante utilizaremos el Front Controller de desarrollo `web/app_dev.php` debido a que desactiva el mecanismo de caché interna de Symfony, permitiéndonos comprobar los cambios que hagamos sobre la configuración de nuestra aplicación, en nuestro navegador accederíamos así: `/app_dev.php` (es decir: `http://localhost/Symfony/web/app_dev.php` en tu navegador).

## DEFINIENDO E IMPORTANDO RUTAS

Como ya sabes una ruta es una asociación o mapa de un patrón URL hacia un Controlador, las mismas se definen en el archivo routing de nuestra aplicación, el cual se encuentra en `app/config` y puede estar definido en tres formatos: YAML, XML o PHP; Symfony 2 utiliza por defecto YAML (`routing.yml`) pero puede cambiarse. Por ejemplo, tenemos una ruta (suponiendo un Bundle `MDWDemoBundle` con `DefaultController` previamente definido):

```
# app/config/routing.yml
path_hello_world:
    pattern: /hello
    defaults: { _controller: MDWDemoBundle:Default:index }
```

### Analicemos cada componente:

- ② **path\_hello\_world**: corresponde al nombre de la ruta, por ahora te parecerá irrelevante, pero es el requisito indispensable para generar las URLs internas en tu sitio que se verá más adelante, el nombre debe ser único, corto y conciso.
- ③ **pattern:/hello**: el atributo pattern define el patrón de la ruta, lo que le permite al Routing empatar las peticiones, si el patrón fuese solo "/" representaría nuestra página de inicio, más adelante se verá su uso avanzado y los comodines.
- ④ **defaults: { \_controller: MDWDemoBundle:Default:index }** dentro de defaults tenemos el parámetro especial \_controller donde se define el "Controlador", nótese que sigue un patrón definido **Bundle:Controller:Action**, esto le permite al Routing hallar el controlador especificado, automáticamente Routing resuelve la ubicación del Bundle, el controlador y la acción, sin necesidad de definir los sufijos "Controller" y "Action" correspondientes al controlador y acción.

Con la ruta anterior accederíamos al siguiente Controlador y Acción:

```
<?php
// src/MDW/DemoBundle/Controller/DefaultController.php
namespace MDW\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function indexAction()
    {
        $response = new Response('Hola mundo!!!');
        return $response;
    }
}
```

Así que ésta es la definición básica de una ruta, con ello puedo acceder desde mi navegador a /app\_dev.php/hello.

Pero según la filosofía de los Bundles en Symfony2 se debe hacer el código lo más portable posible y si defino mis rutas en la aplicación, serían parte de mi aplicación y no de mi “Bundle”, por lo cual debería definirlas en el Bundle e importarlas hacia mi aplicación, de esa forma cuando tenga la necesidad de crear otra aplicación en la cual necesite cierto Bundle, sólo tengo que importar las rutas definidas en dicho Bundle para utilizarlas en mi aplicación, con lo cual mi Bundle es verdaderamente desacoplado y portable; para ello traslado mis rutas hacia el Bundle en su propio archivo de rutas: `src/MDW/DemoBundle/Resources/config/routing.yml` y en mi archivo de rutas de la aplicación lo importo:

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

Como vemos la estructura ha cambiado, en el atributo `resource`: podemos definir la ruta completa hacia nuestro archivo de rutas del Bundle, en este caso utilizamos la forma especial `@NombreBundle` lo cual le indica al Routing que internamente resuelva la ruta hacia nuestro Bundle, haciendo la tarea muy cómoda.

También tenemos el atributo `prefix`: que nos permite definir un prefijo, con ello podemos hacer muchas cosas como diferenciar patrones similares en Bundles diferentes anteponiendo un prefijo, por ejemplo, si colocamos `prefix: /blogger` las URLs del Bundle quedarían así: `app_dev.php/blogger/hello` o `/blogger/hello` motivo por el cual el prefijo predeterminado es `"/"` es decir: **sin prefijo**.

A partir de aquí las rutas están definidas en el Bundle y en el routing de nuestra aplicación se importan, haciendo nuestro “Bundle” más portable.

## RUTAS POR DEFECTO EN EL ENTORNO DE DESARROLLO

Si revisamos bien el archivo `config_dev.yml`, utilizado para el entorno de desarrollo notamos que el archivo de rutas principalmente importado es `routing_dev.yml`, en el cual no sólo se registran las rutas del perfilador y otras que necesitas al momento de probar en el entorno de desarrollo, sino también una serie de rutas hacia el `AcmeDemoBundle`, que no es más que un Bundle de pruebas que no necesitas realmente en tu aplicación.

Como el AcmeDemoBundle sólo se registra en el entorno de Desarrollo, no afectará para nada tu aplicación cuando la ejecutes normalmente desde el entorno de Producción, pero debido a que el routing.yml normal es importado al final por éste, si intentas definir una ruta hacia "/" o alguna que coincida con dicho AcmeDemoBundle, notarás que al acceder desde el entorno de Desarrollo tomará prioridad AcmeDemoBundle y no el tuyo, afortunadamente la solución es muy sencilla donde simplemente comentas o eliminás las 3 rutas que pertenecen al AcmeDemoBundle, luego, si lo deseas eliminás dicho Bundle:

```
# app/config/routing_dev.yml
# COMENTAMOS o ELIMINAMOS estas 3 rutas: -----
-----
#_welcome:
# pattern: /
# defaults: { _controller: AcmeDemoBundle:Welcome:index }
#_demo_secured:
# resource: "@AcmeDemoBundle/Controller/SecuredController.php"
# type: annotation
#_demo:
# resource: "@AcmeDemoBundle/Controller/DemoController.php"
# type: annotation
# prefix: /demo
_assetic:
resource: .
type: assetic
_wdt:
resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
prefix: /_wdt
_profiler:
resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
prefix: /_profiler
_configurator:
resource: "@SensioDistributionBundle/Resources/config/routing/
webconfigurator.xml"
prefix: /_configurator
_main:
resource: routing.yml
```

## RESUMEN DE CAPÍTULO

Profundizamos en el comportamiento esencial del Sistema de Routing, en dónde un Controlador Frontal despacha las peticiones a los controladores reales de nuestra aplicación.

Comprendimos la importancia en la importación de las rutas, para así obtener el código correspondiente a nuestros Bundles encapsulados en los mismos, fomentando un diseño perfectamente desacoplable, adicionalmente vimos como desactivar el AcmeDemoBundle desde nuestro entorno de Desarrollo para así permitirnos navegar en nuestra aplicación por completo desde éste y así obtener las ventajas del perfilador para depurar nuestros proyectos.

## CAPÍTULO 5: DEFINICIÓN DE RUTAS CON COMODINES

No sólo de rutas estáticas se compone una aplicación web, usualmente se necesitan pasar parámetros variables por GET, es decir, por la URL y es aquí en donde el Routing saca lo mejor que tiene. Un marcador de posición o comodín es un segmento de la ruta variable, como por ejemplo: /blog/articulo\_x dónde articulo\_x es una parte variable que representa la página a consultar, en Symfony2 estos comodines se definen entre llaves "{}":

```
# src/MDW/DemoBundle/Resources/config/routing.yml
blog_mostrar:
    pattern:  /blog/{slug}
    defaults: { _controller: MDWDemoBundle:Blog:show }
```

La parte {slug} representa nuestro comodín, y como es variable cualquier URL que coincida con la expresión: /blog/\* empatará con dicha ruta, además el mismo comodín (slug) va a coincidir con el parámetro slug que definamos en el Controlador, por ejemplo, en una ruta /blog/el\_articulo\_de\_symfony en el controlador la variable (parámetro) \$slug contendrá "el\_articulo\_del\_symfony".

De forma predeterminada los comodines son requeridos, si tuvieses, por ejemplo, una ruta de paginación como esta: /blog/page/1 tendrías que definir un comodín para el número de página, pero estarías obligado siempre en agregar "1", esto se resuelve añadiendo un valor por defecto:

```
# src/MDW/DemoBundle/Resources/config/routing.yml
blog_index:
    pattern: /blog/page/{page}
    defaults: { _controller: MDWDemoBundle:Blog:index, page: 1 }
```

De esta forma añadimos el valor por defecto del comodín page: 1 como parámetro del atributo defaults.

## AGREGANDO REQUISITOS A LA RUTA

Hasta ahora tenemos dos rutas: /blog/{slug} y /blog/page/{page}, la segunda es poco práctica ¿que pasaría si la simplificamos como /blog/{slug} y /blog/{page}?, que ambos patrones asemejen con /blog/\* y el Routing solo tomará en cuenta la primera ruta coincidente. La solución a este

problema es añadir requerimientos, como por ejemplo definir que el comodín {page} acepte solo números que es lo que realmente lo diferencia del {slug}:

```
# src/MDW/DemoBundle/Resources/config/routing.yml
blog_index:
    pattern: /blog/{page}
    defaults: { _controller: MDWDemoBundle:Blog:index, page: 1 }
    requirements:
        page: \d+
blog_mostrar:
    pattern: /blog/{slug}
    defaults: { _controller: MDWDemoBundle:Blog:show }
```

Nótese que en el nuevo parámetro requirements: puedes definir expresiones regulares para cada comodín, en el caso anterior al añadirle \d+ a {page} le indicamos al Routing que dicho parámetro debe de coincidir con la expresión regular, lo que te permite definir el requerimiento de acuerdo a tus necesidades.

Es importante aclarar que debes declarar tus rutas en un orden lógico, debido a que el Routing toma en cuenta la primera ruta que coincide, por ejemplo, si defines la ruta blog\_mostrar antes de blog\_index no funcionará porque como blog\_mostrar no tiene requerimientos cualquier número coincide perfectamente con {slug} y llegar a {page} nunca será posible de esa forma.

Además de especificar requisitos para cada comodín, existen otros de mucha ayuda:

- ② **\_method:** [GET | POST] como lo indica su nombre, permite establecer como restricción que la ruta solo coincida si la petición fue POST o GET, muy útil cuando nuestra acción del controlador consista en manipulación de datos o envío de forms.
- ② **\_format:** es un parámetro especial que permite definir el content-type de la Respuesta (Response), puede ser utilizado en el patrón como el comodín {\_format} y de esta forma pasarlo al Controlador.

Con todo esto ya somos capaces de crear rutas dinámicas con Symfony2, pero el sistema de Routing es tan flexible que permite crear rutas avanzadas, veamos:

```
# src/MDW/DemoBundle/Resources/config/routing.yml
blog_articulos:
```

```

pattern: /articulos/{culture}/{year}/{titulo}.{_format}
defaults: { _controller: MDWDemoBundle:Blog:articulos, _format: html }
requirements:
    culture: en|es
    _format: html|rss
    year: \d{4}+

```

En este ejemplo podemos apreciar que {culture} debe de coincidir con en o es, {year} es un número de cuatro dígitos y el {\_format} por defecto es html, así que las siguientes rutas empatan con nuestra anterior definición:

- ⇒ /articulos/es/2000/patron\_mvc
- ⇒ /articulos/es/2000/patron\_mvc.html
- ⇒ /articulos/es/2000/patron\_mvc.rss

## GENERANDO RUTAS

Como anteriormente indicamos, el Routing es un sistema bidireccional, en el cual nos permite generar URLs desde las mismas definiciones de rutas, el objetivo es obvio: si deseas cambiar el patrón de la ruta lo haces y ya. No necesitas buscar los links internos hacia esa ruta dentro de tu aplicación si utilizas el generador de rutas de Symfony2.

La forma más común de utilizar el generador de rutas es desde nuestras plantillas (Vistas/Views) y para ello solo necesitamos acceder al objeto “router”, ejemplos:

con **TWIG** como gestor de plantillas usamos path para obtener la url relativa:

```

<a href="{{ path('blog_mostrar', { 'slug': 'mi-articulo' }) }}>
    lee el artículo.
</a>

```

O url si queremos una url absoluta:

```

<a href="{{ url('blog_mostrar', { 'slug': 'mi-articulo' }) }}>
    lee el artículo.
</a>

```

con **PHP** como gestor de plantillas vemos que en realidad accedemos al helper router para obtener

la url relativa:

```
<a href="<?php echo $view['router']->generate('blog_mostrar', array('slug' => 'mi-articulo')) ?>">
    lee el artículo
</a>
```

Si quisieramos una url absoluta solo se debe especificar true en el tercer parámetro de la función generate:

```
<a href="<?php echo $view['router']->generate('blog_mostrar', array('slug' =>'mi-articulo'), true) ?>;>
    lee el artículo
</a>
```

Así de sencillo. Si en nuestras vistas usamos el generador de rutas no tendremos que preocuparnos por la eventualidad de cambiar las URLs de nuestra aplicación, además de que también podemos acceder desde nuestros controladores gracias a la inyección de dependencias(DI) donde obtendremos el servicio “router”:

```
// src/MDW/DemoBundle/Controller/DefaultController.php
class BlogController extends Controller
{
    public function showAction($slug)
    {
        // ...
        $url = $this->get('router')->generate('blog_mostrar', array('slug' => 'mi-articulo'));

        // Atajo si extiendes la clase Controller:
        $url = $this->generateUrl('blog_mostrar', array('slug' => 'mi-articulo'));
    }
}
```

Esto es todo lo básico que debes saber para generar rutas dinámicas y concisas en Symfony2.

## RESUMEN DE CAPÍTULO

El sistema de rutas de Symfony2 no sólo nos permite definir caminos claros y concisos para nuestros usuarios, sino que también nos hace más fácil su implementación, con lo cual al cambiar el patrón de una ruta no tenemos que buscar todas las llamadas de la misma en la aplicación.

Además aprendimos a definir rutas condicionadas, entendimos la importancia del orden de declaración de las mismas y vimos como utilizar el sistema de routing bidireccional en nuestras vistas y controladores.

## CAPÍTULO 6: EL CONTROLADOR

En el capítulo anterior aprendimos a crear las rutas hacia nuestros controladores, como algunos ya saben el “Controlador” (o Controller) es la piedra angular de la arquitectura MVC, es lo que nos permite enlazar el modelo de la vista y básicamente donde se construye la aplicación.

### ¿QUÉ HACE EXACTAMENTE UN CONTROLADOR?:

Bajo la filosofía de Symfony2 el concepto es simple y preciso: “El controlador recibe la petición (Request) y se encarga de crear y devolver una respuesta (Response)”; a simple vista pensarás “¡el controlador hace más que eso!” sí, pero a Symfony2 sólo le importa eso, lo que implementes dentro de él depende de tu lógica de aplicación y negocio\*, al final un controlador puede:

1. Renderizar una vista.
2. Devolver un contenido tipo XML, JSON o simplemente HTML.
3. Redirigir la petición (HTTP 3xx).
4. Enviar mails, consultar el modelo, manejar sesiones u otro servicio.

Te darás cuenta de que todas las funciones anteriores terminan siendo o devolviendo una “**Respuesta**” al cliente, que es la base fundamental de una aplicación basada en el ciclo Web (Petición -> acción -> Respuesta). En este capítulo nos concentraremos en la estructura y funciones básicas que puedes hacer en el controlador.

---

NOTA: \*Las buenas prácticas indican que la lógica de negocios debe residir en el modelo (modelos gordos, controladores flacos).

## DEFINIENDO EL CONTROLADOR

Los controladores en Symfony2 se declaran como archivos con el sufijo Controller “Micontrolador-Controller.php” en el directorio Mibundle\Controller dentro de nuestro Bundle, quedando así nuestro ejemplo: “src/MDW/DemoBundle/Controller/DemoController.php”, veamos el código:

```
<?php  
// src/MDW/DemoBundle/Controller/DefaultController.php
```

```
namespace MDW\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function indexAction()
    {
        // $response = new \Symfony\Component\HttpFoundation\Response('Hola
        mundo!!!');
        return new Response('Hola mundo!!!');
    }
}
```

Vemos que el controlador es una clase que extiende de la clase base Controller (Symfony\Bundle\FrameworkBundle\Controller\Controller), pero en realidad esta clase es un paquete de controladores, ¡sí, es confuso!: nuestros verdaderos controladores son en realidad las funciones (acciones) que contiene esta clase, cuyo sufijo siempre sea **Action**.

En la mayoría de los FW el Controlador contiene varias Acciones y si nos damos cuenta es **en las Acciones en donde aplicamos la verdadera lógica del controlador**, consideralo como un cliché; además Symfony2 no requiere que la clase controlador extienda de Controller, la clase base Controller simplemente nos provee de atajos útiles hacia las herramientas del núcleo del framework, como el DI (inyección de dependencias), por eso se recomienda su uso.

Como primera norma tenemos que asignar el “**namespace**” a nuestro controlador (línea #3), para asegurar que el mecanismo de autoload pueda encontrar nuestro controlador y con ello el Routing (entre otros) pueda hallarlo eficientemente, luego vemos una serie de declaraciones **use** que nos sirven para importar los diferentes espacios de nombres y clases que necesitamos para trabajar.

**Namespaces:** Es sabido que para algunos este cambio es brusco, pero tiene su objetivo claro el cual otros FW siguen y dentro de poco otros seguirán, entre ellos evitar conflictos con los nombres de clases, pero bueno ¿te frustra escribir tanto código?: tranquilo, que si usas un IDE inteligente como Netbeans 7 con solo hacer nombre a la clase puede generarte automáticamente la cadena de refe-

rencia completa (como puedes notar en la línea comentada #12), claro está que utilizar use es más elegante y forma parte de las buenas prácticas.

Al final vemos que simplemente creamos nuestra respuesta (objeto Response) y lo devolvemos, eso es lo básico de nuestro controlador, más adelante expondremos los detalles.

## RENDERIZANDO (GENERANDO) VISTAS

Prácticamente es la tarea más común en cada controlador en la cual hacemos la llamada al servicio de Templating, Symfony2 por defecto utiliza Twig. Nos concentraremos en la forma de renderizar nuestras vistas, para eso añadimos la siguiente acción en nuestro BlogController que nos corresponderá a la ruta “blog\_index” definida en el capítulo anterior:

```
// src/MDW/DemoBundle/Controller/BlogController.php
// ...
public function showAction($slug)
{
    $articulo = $slug; // (suponiendo que obtenemos el artículo del modelo
    con slug)
    return $this->render('MDWDemoBundle:Blog:show.html.twig', array(
        'articulo' => $articulo
    ));
}
// ...
```

La función `$this->render()` acepta 2 parámetros: el primero es un patrón para encontrar la plantilla `[Bundle]:[Controller]:template.[_format].template_engine` y el segundo el array para pasar variables a la plantilla, devolviendo un objeto `Response()`.

Como apreciarás en el patrón, Symfony busca la correspondiente plantilla en el directorio `MyBundle/Resoruces/views`, según su propia estructura interna detallada en el próximo capítulo de la Vista. Como indicamos en el capítulo anterior sobre Routing, los parámetros de nuestro Controlador coinciden con los comodines definidos en la ruta con el cual obtendremos eficientemente estos datos.

En realidad la función `$this->render()` es un atajo de nuestra clase base Controller que renderiza la plantilla y crea el Response, al final tenemos estas diferentes formas para hacer el mismo proceso:

## Otras formas de renderizar plantillas:

```
// 2da forma- atajo que obtiene directamente el contenido renderizado:  
$content = $this->renderView('MDWDemoBundle:Blog:show.html.twig',  
array('articulo' => $articulo));  
return new Response($content);  
  
// 3ra forma- obteniendo el servicio templating por DI (inyección de  
dependencias):  
$templating = $this->get('templating');  
$content = $templating->render('MDWDemoBundle:Blog:show.html.twig',  
array('articulo' => $articulo));  
return new Response($content);
```

Como vemos, los atajos de la clase base Controller son de mucha ayuda y si necesitaras mayor flexibilidad puedes optar por la 2da y 3ra forma.

## OBteniendo DATOS DE LA PETICIÓN (REQUEST)

Ya sabes que los parámetros del Controlador coinciden con los comodines definidos en las Rutas, pero existe otra forma de acceder a las variables POST o GET pasadas a nuestra petición y como es obvio lo proporciona el objeto `Request()`, existen varias formas de obtener el objeto Request, incluso como en Symfony1 en Symfony2 lo puedes pasar como parámetro del controlador, como el sistema de Routing empata éstos con los comodines el orden no importa:

```
// src/MDW/DemoBundle/Controller/BlogController.php  
// ...  
use Symfony\Component\HttpFoundation\Request; //es necesario para importar  
la clase  
//...  
public function showAction(Request $peticion, $slug)  
{  
    $articulo = $peticion->get('slug'); // otra forma para obtener  
comodines, GET o POST  
    $metodo = $peticion->getMethod(); //obtenemos si la petición fue por  
GET o POST  
    return $this->render('MDWDemoBundle:Blog:show.html.twig', array(  
        'articulo' => $articulo
```

```
));
}
```

Vemos que con el objeto Request podemos obtener por completo los datos de nuestra petición, además la clase base Controller dispone de un atajo para obtener el Request sin necesidad de pasarlo como parámetro y de esa forma nos ahorraremos la importación por use:

```
$peticion = $this->getRequest();
//lo que es lo mismo que con DI:
$peticion = $this->get('request');
```

## REDIRECCIONES

En diversas oportunidades nos vemos obligados en hacer una redirección, ya sea por haber procedido un POST o dependiendo de la lógica que apliquemos en el Controlador, por ejemplo, si tuviéramos que redirigir a HTTP 404 por un registro en el modelo no encontrado, etc.

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

Como notarás la función `$this->redirect()` acepta como primer parámetro una string Url (la función `$this->generateUrl()` nos permite generar la url desde una ruta definida en Routing) y por defecto realiza una redirección HTTP 302 (temporal), en el segundo parámetro puedes modificarla para hacer una 301 (permanente)

En el caso de redirigir a HTTP 404, podremos causar una Excepción de esta forma que Symfony2 intercepta internamente:

```
public function indexAction()
{
    $producto = false;// suponiedo que nuestro modelo devuelva false al no encontrarlo
    if (!$producto) {
        throw $this->createNotFoundException('No existe el producto');
    }
    return $this->render(...);
```

```
}
```

En ocasiones necesitarás pasar la petición a otro controlador internamente sin necesidad de redirecciones HTTP, en este caso el **Forwarding** es tu alternativa:

```
public function indexAction($name)
{
    $respuesta = $this->forward('MDWDemoBundle:Blog:index', array(
        'slug' => $name
    ));
    // puedes modificar directamente la respuesta devuelta por el
    controlador anterior.
    return $respuesta;
}
```

Como podrás notar, en el primer parámetro de la función `$this->forward()` acepta el mismo formato que utilizamos en el Routing para hallar el Controlador y como segundo parámetro un array con los parámetros de dicho Controlador, cabe mencionar que el mismo ha de ser un controlador normal y no es necesario que tenga una ruta en Routing definida.

## RESUMEN DEL CAPÍTULO

Como apreciamos en éste capítulo, los Controladores en Symfony2 requieren el sufijo Controller para ser hallados por el FW, además de resaltar que la verdadera lógica del controlador reside en sus Acciones las cuales son las funciones internas de los mismos declaradas con el sufijo Action, además de ello vimos las funcionalidades básicas del controlador, como lo son acceder a las variables de la petición “Request”, Redireccionar, pasar a otros controladores sin redirecciones (Forwarding) y el renderizado de las vistas.

## CAPÍTULO 7: LA VISTA Y TWIG

Dentro del patrón o arquitectura MVC la vista es la encargada de proporcionar la verdadera “interfaz” a nuestros usuarios, para ello en Symfony2 podemos usar el servicio de Templating (plantillas) y como ya sabrás Twig es el motor por defecto, aunque si lo prefieres puedes usar PHP.

¿Por qué Twig?: porque las plantillas en Twig son muy fáciles de hacer y resultan muy intuitivas en el caso de que contemos con maquetadores o Diseñadores Frontend, además de todo ello su sintaxis corta y concisa es muy similar (por no decir idéntica) a la de otros famosos FW como django, Jinja, Ruby OnRails y Smarty; además Twig implementa un novedoso mecanismo de herencia de plantillas y no tendrás que preocuparte por el peso que conlleva el interpretar todo ello debido a que Twig cachea en auténticas clases PHP todo el contenido de las mismas, para acelerar el rendimiento de nuestra aplicación.

En este capítulo nos concentraremos en las nociones básicas de Twig y verás lo fácil que es interactuar con él.

### LO BÁSICO DE TWIG

```
{# comentario #}  
{ { mostrar_algo } }  
{ % hacer algo %}
```

Como puedes ver con Twig mostrar el contenido de una variable es tan simple como usar las dobles llaves {{ variable }}, sin necesidad de echo ni etiquetas de apertura de PHP (<? php ?>), además de eso **Twig es un lenguaje de plantillas**, lo que nos permite hacer condicionales y estructuras de control muy intuitivas y funcionales:

```
<ul>  
{ % for usuario in usuarios %}  
    <li>{{ usuario.nombreusuario | upper }}</li>  
{ % else %}  
    <li><em>no hay usuarios</em></li>  
{ % endfor %}  
</ul>
```

`{for .. else?}`: sí, esto en realidad no existe en PHP, como puedes notar Twig internamente hace una comprobación de la colección (u array) antes de iterarlo, lo que hace a la plantilla más fácil de escribir y es intuitivo para maquetadores.

`{{ usuario.nombreusuario | upper }}` Twig dispone de una serie de filtros, los cuales puedes anidar hacia la derecha con el operador “pipe” (|) de esta forma con el filtro `upper` nos imprime el valor de la variable `usuario.nombreusuario` en mayúsculas, al final del capítulo mostraremos los filtros más comunes.

## NOMENCLATURA Y UBICACIÓN DE PLANTILLAS

Symfony 2 dispone de dos principales lugares para contener a las plantillas:

- ⌚ en nuestros Bundles: `src/Vendor/MyBundle/Resources/views`
- ⌚ en la Aplicación: `app/Resources/views` y para reemplazo: `app/Resources/VendorMyBundle/views`.

De esta forma el servicio de templating de Symfony busca primero las plantillas en la Aplicación y luego en el mismo Bundle, permitiéndonos un mecanismo simple para reemplazar plantillas en Bundles de terceros. Cuando renderizamos las vistas (es decir plantillas) desde el Controlador utilizamos un patrón definido: `[Bundle] : [Controller] : template[._format].template_engine`

El parámetro `template` representa el nombre de archivo de nuestra plantilla (como norma se recomienda el mismo nombre de la Acción del Controlador), el parámetro `_format` es requerido y se usa para poder diferenciar el formato real que representará la plantilla y el último parámetro `template_engine` que representa la extensión real de nuestra plantilla le indica al servicio de templating el motor a utilizar, así que un archivo de plantillas quedaría de la siguiente forma:

`show.html.twig` o `show.rss.twig`

Con respecto a la primera parte los parámetros `[Bundle]:[Controller]` nos permiten ubicar la plantilla dentro de un Bundle, como puedes notar `Controller` indica que existe otra carpeta (`views/Controller`), en donde buscara el template, si se omite buscará nuestro template en `views`, ejemplo:

## MDWDemoBundle:Blog:show.html.twig =

- ⌚ src/MDW/DemoBundle/Resources/views/Blog/show.html.twig
- ⌚ app/Resources/MDWDemoBundle/views/Blog/show.html.twig

## MDWDemoBundle::show.html.twig =

- ⌚ src/MDW/DemoBundle/Resources/views/show.html.twig
- ⌚ app/Resources/MDWDemoBundle/views/show.html.twig

Nótese que de omitirse el Controller se busca en la raíz de views, pero también podemos omitir el Bundle, de ésta forma indicamos que queremos una plantilla global de nuestra Aplicación, con el cual no dependería del Bundle:

```
:Blog:show.html.twig = app/Resources/views/Blog/show.html.twig
::show.html.twig = app/Resources/views/show.html.twig
```

De esta forma podremos indicar con precisión de donde queremos nuestras plantillas y no sólo desde los Controladores, éste patrón también se aplica dentro de las mismas plantillas, tanto para incluir otras como para el mecanismo de herencia de plantillas que veremos más adelante.

## HERENCIA DE PLANTILLAS

Este mecanismo está pensado para simular el mismo comportamiento de la POO en nuestras plantillas, de modo que tengamos “plantillas base” que contienen los elementos comunes como el Layout, de esta forma las plantillas pueden heredar de las Plantillas Base, proporcionando un Decorado de plantillas multi-nivel; Twig provee esto por medio de Bloques (block) y utilizando el método extends para heredar la plantilla, vemos un ejemplo básico de una plantilla que herede de otra:

### PLANTILLA BASE POR DEFECTO DE SYMFONY2:

```
{# app/Resources/views/base.html.twig #-}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"
    />
        <title>{% block title %}Welcome!{% endblock %}</title>
```

```

    {% block stylesheets %}{% endblock %}
    <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
</head>
<body>
    {% block body %}{% endblock %}
    {% block javascripts %}{% endblock %}
</body>
</html>

```

## PLANTILLA LAYOUT DE NUESTRO BUNDLE:

```

{-# src/MDW/DemoBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}
{% block stylesheets %}
    {{ parent() }}
    <link href="{{ asset('bundles/mdwdemo/css/main.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}
{% block body %}
    <div id="layout">
        <div id="header">
            <h1>Encabezado</h1>
        </div>
        <div id="content" class="clean" >
            <div id="content_left">
                {% block content %}
                {% endblock %}
            </div>
            <div id="content_right">
                columna
            </div>
        </div>
        <div id="footer">
            <p>Pie {{ "now" | date("m/d/Y") }}</p>
        </div>
    </div>
{% endblock %}

```

Como vemos, en la Plantilla base (base.html.twig) se define la estructura básica de todo documento HTML (en este caso HTML5) y dentro de la misma se definen los bloques **title, stylesheets, body y javascript**, de ese modo la plantilla que herede de ella puede reemplazar cada uno de estos bloques facilitando el decorado, en la plantilla layout.html.twig de nuestro Bundle se aprecia:

- ⌚ **{% extends '::base.html.twig' %}**: extiende o hereda (herencia simple) de la plantilla base, fíjate que la nomenclatura es igual a la expresada en el enunciado anterior.
- ⌚ **{% block stylesheets %}**: redefinir el bloque stylesheets indica que se reemplaza para incluir una hoja de estilos propia de nuestro bundle y dentro del bloque notarás `{} parent()` esto permite que el contenido original del mismo bloque en la plantilla base sea agregado, con el cual podremos conservar las hojas CSS declaradas en la plantilla base.

Además podemos definir nuevos bloques como `{% block content %}`, de esta forma en las plantillas que extiendan de ésta puedan reemplazar solo el bloque estratégico, esto con el fin de introducir el modelo de Herencia a tres niveles propuesto por Symfony2 que brinda una manera flexible de independizar nuestro layout del Bundle del de la Aplicación, con ello nuestras plantillas finales tendrían este aspecto:

```
# src/MDW/DemoBundle/Resources/views/Blog/show.html.twig #
{% extends 'MDWDemoBundle::layout.html.twig' %}
{% block content %}
    El artículo solicitado es: {{articulo}}
{% endblock %}
```

De ésta manera al renderizar la plantilla show.html.twig extiende del layout.html.twig del Bundle que extiende del base.html.twig de la Aplicación y en la plantilla solo necesitamos reemplazar el Bloque correspondiente al contenido.

## REUTILIZANDO PLANTILLAS

Una de las necesidades más comunes que pueden presentarse es la de fragmentar el código común y repetitivo en nuestras plantillas, con el objetivo de reutilizarlo en más de una plantilla, para ello en Symfony2 podemos crear tales fragmentos en archivos de plantillas separados y utilizando el helper include de twig podemos incluir dinámicamente el contenido de dicha plantilla, además de permitirnos pasar datos a la misma:

```
{% include 'MDWDemoBundle:Blog:articuloDetalles.html.twig' with
{'articulo': articulo} %}
```

Como puedes notar el patrón de acceso a la plantilla es el mismo que utilizas al renderizar cualquier plantilla, además puedes añadir un array de variables pasadas a la misma después de la cláusula `with`.

## REUTILIZANDO CONTROLADORES

En algunos casos en nuestras plantillas incluidas necesitamos acceder a datos del modelo que de otra forma sólo sería posible si desde el controlador añadimos la consulta hacia el modelo (por ejemplo un banner con los 3 títulos de artículos más recientes), por lo cual tendríamos un doble trabajo al volver a pasar las variables generadas al `include` de dicha plantilla, una mejor solución es crear otro controlador que realice dicha tarea por separado y renderizarlo o incrustarlo directamente desde la plantilla, para ello utilizamos el helper `render`:

```
{% render "MDWDemoBundle:Articulo:articulosRecientes" with {'max': 3} %}
```

De esta forma puedes crear un controlador `articulosRecientes` que realice la correspondiente consulta al modelo y pasar variables específicas como la cantidad que quieras consultar, así no es necesario incluir consultas adicionales en tu controlador principal, conservando la semántica de cada controlador y un código más limpio y ordenado.

## INCORPORANDO ACTIVOS (ASSETS)

Toda aplicación web contiene un conjunto de Activos (Assets) que son básicamente archivos JavaScript, Hojas de estilo CSS, Imágenes y demás; generalmente estos archivos deben de publicarse dentro del árbol de publicación del sitio para que sean accesibles por el navegador, pero el hecho de disponer de “Url Amigables en la aplicación” nos obliga a generar rutas hacia éstos desde la raíz del sitio anteponiendo un slash (/), por ejemplo “/images/logo.gif”, eso si la aplicación está disponible desde la raíz del servidor, en el caso de tenerla en una carpeta compartida hay que añadirla: “**/miaplicacion/web/images/logo.gif**” lo que resulta en un problema, afortunadamente el **Helper asset** nos permite hacer la aplicación más portable, con ello solo necesitamos:

```

```

De esta forma no importa en donde viva tu aplicación, el Helper asset de Symfony 2 se encarga de generar la URL correcta para que no tengas que preocuparte por ello.

# LISTADO DE FILTROS TWIG Y HELPERS MÁS COMUNES

## RAW Y ESCAPADO DE VARIABLES (XSS)

De forma predeterminada Twig **escapa** todos los caracteres especiales HTML de las variables, lo que permite una protección frente a XSS, de igual forma si tenemos un contenido que disponga de código HTML seguro, podremos evitar este mecanismo con el filtro raw:

```
{{ variable | raw }} {# evita el escapado de variables #}
{# fuerza el escapado de variables (opción por defecto en Symfony2) #}
{{ variable | escape }}
```

Default (valores por defecto) y detectando variables no declaradas

Si una variable no es pasada a la plantilla twig devolverá una excepción, esto es un inconveniente para cuando necesitemos reutilizar plantillas en las cuales no todas las variables necesiten ser pasadas, afortunadamente en estos casos podemos definir un valor por defecto:

```
{{ variable | default('valor por defecto') }}
```

Pero en oraciones no necesitamos que twig nos devuelva un valor por defecto, sino saber si la variable fue declarada o no, para interactuar en un condicional por ejemplo, allí usamos `is defined`:

```
{% if variable is defined %}
    {# aplicar operaciones si no se ha declarado la variable #}
{% endif %}
```

En el caso de querer comprobar si la variable está declarada pero tiene un valor null usamos `variable is null`

## CAPITALIZACIÓN, MAYÚSCULAS Y MINÚSCULAS

Con estos sencillos filtros podremos capitalizar o convertir a mayúsculas / minúsculas una variable cadena:

```
{{ variable | capitalize }} {# capitaliza el primer carácter de la cadena #}
{{ variable | lower }} {# convierte a minúsculas la cadena #}
{{ variable | upper }} {# convierte a mayúsculas la cadena #}
{{ variable | title }} {# capitaliza cada palabra de la cadena #}
```

y por si fuera poco, podremos aplicar el filtro a un gran bloque de código HTML anidándolo dentro de un bloque filter de twig:

```
{% filter upper %}
    Todo el texto de aquí será convertido a mayúsculas
{% endfilter %}
```

## DATE (FORMATANDO FECHAS)

El filtro date es una forma rápida de aplicar formato a nuestras variables con fechas y lo mejor de todo es que internamente aplica las mismas convenciones de la función date() nativa de PHP, además como parámetro adicional podemos establecer la zona horaria:

```
{{ variable | date("m/d/Y", "Europe/Paris") }}
```

En ciertas ocasiones necesitamos simplemente la fecha/hora actual, por lo que no es necesario declarar en el controlador una variable y asignarle el timestamp actual, colocando como fecha "now" en twig es realmente simple:

```
{# "now" nos devuelve la fecha/hora actual #}
{{ "now" | date("m/d/Y") }}
```

## NUMBER\_FORMAT (FORMATO NUMÉRICO)

Twig realmente nos ofrece con este filtro un atajo a la función nativa number\_format de PHP:

```
{{ 2500.333 | number_format(2, ',', '.') }}
```

## (CONCATENACIÓN EN TWIG) Y DECLARANDO VARIABLES DESDE PLANTILLA (~)

En ciertas oraciones necesitamos concatenar una cadena estática con una o más variables, para aplicarlo como parámetro de un filtro o para asignarlo a una variable declarada desde plantilla para reutilizarla, un ejemplo práctico es crear una variable con el "share url" para compartir una entrada de blog en varias redes sociales así evitamos el repetir constantemente el llamado al helper path:

```
{% set url_share = 'http://maycolalvarez.com' ~ path('blog_article', {
    'year' : (article.created|date('Y')),
    'month' : (article.created|date('m')),
    'slug' : article.slug })
}

<!-- Coloca esta etiqueta donde quieras que se muestre el botón +1. --&gt;</pre>

```

```
<g:plusone size="medium" href="{{ url_share }}"></g:plusone>
<a href="https://twitter.com/share" class="twitter-share-button" data-
url="{{ url_share }}" data-lang="es">Twittear</a>
<div class="fb-like" data-href="{{ url_share }}" data-send="false" data-
layout="button_count" data-width="100" data-show-faces="true"></div>
```

Como ves, con `set variable` = podemos declarar una variable, asignamos una cadena estática entre comillas simples y concatenamos el resultado del helper `path` con el operador (~), con ello podemos usar `{{ url_share }}` en la url de cada botón de red social, en el ejemplo Google+, Twitter y Facebook (consulte su api para más detalles).

## RESUMEN DE CAPÍTULO

Manejar las Vistas con Twig nos permite tener código mucho más limpio e intuitivo, no sólo abstractea la verdadera lógica de una vista, sino que nos proporciona de herramientas útiles, no sólo a diseñadores sino a programadores también como la herencia de plantillas, lo cual separa de forma limpia el layout, además de incluir plantillas y modularizar el contenido de las vistas y llamar controladores para obtener datos del modelo sin romper el esquema MVC.

Con esto culminamos el capítulo de la vista, reitero que puedes usar PHP como motor de plantillas si lo deseas, pero si eliges quedarte con twig no olvides revisar su documentación (<http://twig.sensiolabs.org/documentation>) para extender tus conocimientos y mejorar tu desempeño.

## CAPÍTULO 8: CONFIGURANDO NUESTRA BASE DE DATOS

En este capítulo entraremos al mundo de Doctrine usándolo dentro de nuestro proyecto para así tener ya los datos dinámicamente almacenados en nuestra base de datos y manipularlos.

Como ya lo habíamos hablado en el capítulo 1, Doctrine es un ORM (Object-Relational Mapping). Cuando hablamos de relaciones en conceptos de base de datos, estamos refiriéndonos a las tablas diciendo entonces que existe una vinculación entre las tablas y objetos. Al usar un ORM mapeamos cada tabla con objetos dentro de nuestras aplicaciones, por ejemplo si tenemos una tabla de personas en la base de datos, tendremos un objeto Persona en la aplicación que conoce cuales son sus campos, tipos de datos, índices, etc. logrando con esto que la **aplicación** conozca el **modelo de los datos** desde un punto de vista orientado a objetos, es decir representado con Clases y Objetos.

Doctrine nos permitirá, conociendo nuestras tablas como hablamos anteriormente, crear las sentencias SQL por nosotros ya que toda la información necesaria para crear estos queries se encuentra “mapeada” en código PHP.

Como si esto fuera poco, la mayor de las ventajas de contar con un ORM será que nos permite como desarrolladores, abstraernos de que motor de base de datos estemos usando para el proyecto y nos permitirá, con solo un poco de configuración, cambiar toda nuestra aplicación por ejemplo de una base de datos MySQL a PostgreSQL o a cualquier otra soportada por el framework Doctrine.

## CONFIGURACIÓN DE LA BASE DE DATOS

Para configurar los datos de la conexión del servidor de base de datos se deberán ingresar los valores en el archivo app\config\parameters.ini dentro de las variables ya definidas:

- ⇒ database\_driver = pdo\_mysql
- ⇒ database\_host = localhost
- ⇒ database\_port =
- ⇒ database\_name = blog
- ⇒ database\_user = maestros
- ⇒ database\_password = clavesecreta

Estos datos serán usados por Doctrine para conectarse al servidor y trabajar con la base de datos. No hay necesidad de ingresar el puerto si se está usando el que figura por defecto para la base de datos que usemos, para nuestro caso MySQL.

Una vez que ya tenemos configurada nuestra base de datos, podemos usar el comando “console” para decirle a nuestro proyecto que se encargue de crear la base de datos en el servidor ejecutándolo de la siguiente manera:

```
C:\wamp\www\Symfony>php app\console doctrine:database:create  
Created database for connection named <comment>blog</comment>
```

Esto nos retornará un mensaje diciéndonos, si los permisos estaban correctos, que la base de datos **blog** fue creada. Podemos revisar ahora nuestra base de datos por medio del phpMyAdmin y veremos que la base de datos ya existe.

---

**Nota:** En caso de necesitar borrar la base de datos usaremos el comando “doctrine:database:create --force”, donde tendremos que pasarle el parámetro especial --force para confirmar que ejecute la acción.

El concepto que suele ser muy utilizado en frameworks de este tipo es ir **creando la base de datos desde la aplicación**, es decir, que la aplicación se encargue de crear la base de datos, las tablas, relaciones, índices y los datos de ejemplo. Este concepto lo iremos ahondando durante estos capítulos.

## CREANDO LAS TABLAS Y CONOCIENDO LAS ENTIDADES

Como ejemplo usaremos dos tablas para nuestro manual simplemente para ver como trabajar con ellas usando el framework. Crearemos una tabla de Artículos y una de Comentarios para tener como ejemplo el concepto ya bien conocido de un blog.

A continuación tenemos el SQL de las tablas con sus campos:

```
CREATE TABLE IF NOT EXISTS `articles` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `title` varchar(255) NOT NULL,  
  `author` varchar(255) NOT NULL,  
  `content` longtext NOT NULL,  
  `tags` varchar(255) NOT NULL,
```

```

`created` date NOT NULL,
`updated` date NOT NULL,
`slug` varchar(255) NOT NULL,
`category` varchar(255) NOT NULL,
PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
CREATE TABLE IF NOT EXISTS `comments` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`author` varchar(255) NOT NULL,
`content` longtext NOT NULL,
`reply_to` int(11) NOT NULL,
`article_id` int(11) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `IDX_A6E8F47C7294869C`(`article_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

```

Para crear estas tablas en la base de datos primeramente lo haremos en código PHP para que Doctrine conozca perfectamente las tablas para interactuar con ellas. Es aquí donde conocemos el concepto de las Entidades (Entities). Una entidad es la representación “orientada a objetos” de nuestras tablas, es decir que crearemos una clase por cada una de nuestras tablas. Para no tener que escribirlas a mano, Symfony nos provee un generador de Entidades que es invocado con nuestro comando console de la siguiente manera:

```
C:\wamp\www\Symfony>php app\console doctrine:generate:entity
```

Al darle enter nos preguntará el nombre que usaremos para nuestra entidad y hay que entender que sigue el siguiente formato: **IdentificadorDelBundle:NombreEntidad**. Para nuestro caso y como ya vimos en los capítulos anteriores, nuestro identificador del Bundle es **MDWDemoBundle** y primero crearemos la entidad para nuestra tabla de artículos por lo tanto escribiremos lo siguiente:

```
The Entity shortcut name: MDWDemoBundle:Articles
```

Al darle enter nos preguntará que formato queremos usar para agregar los metadatos que mapearan a la tabla proponiéndonos [annotation]. Aceptaremos la propuesta presionando Enter.

```
Configuration format (yml, xml, php, or annotation) [annotation]:
```

Luego empezará a preguntarnos cuales serán las columnas de nuestra tabla y por cada columna el tipo de dato que arriba nos deja como ejemplo cuales podemos elegir. En caso de que sea un tipo

de datos que necesite longitud también nos la pedirá y cuando ya no queramos ingresar columnas simplemente daremos un enter dejando vacío el valor.

---

**Nota:** No será necesario ingresar el campo id de la tabla ya que Symfony lo creará automáticamente con la idea de ser usado como clave primaria autonumérica.

Una vez finalizada la carga de los campos, nos preguntará si queremos crear un repositorio vacío a lo que contestaremos Si para luego, como último paso preguntarnos si confirmamos la creación automática de la Entidad a lo que también diremos que Si:

```
Instead of starting with a blank entity, you can add some fields now.  
Note that the primary key will be added automatically (named id).  
Available types: array, object, boolean, integer, smallint,  
bigint, string, text, datetime, datetimetz, date, time, decimal, float.  
New field name (press <return> to stop adding fields): title  
Field type [string]:  
Field length [255]:  
New field name (press <return> to stop adding fields): author  
Field type [string]:  
Field length [255]:  
New field name (press <return> to stop adding fields): content  
Field type [string]: text  
New field name (press <return> to stop adding fields): tags  
Field type [string]:  
Field length [255]:  
New field name (press <return> to stop adding fields): created  
Field type [string]: date  
New field name (press <return> to stop adding fields): updated  
Field type [string]: date  
New field name (press <return> to stop adding fields): slug  
Field type [string]:  
Field length [255]:  
New field name (press <return> to stop adding fields): category  
Field type [string]:  
Field length [255]:  
New field name (press <return> to stop adding fields):  
Do you want to generate an empty repository class [no]? yes
```

**Summary before generation**

You are going to generate a “MDWDemoBundle:Articles” Doctrine2 entity using the “annotation” format.

Do you confirm generation [yes]?

Entity generation

Generating the entity code: OK

You can now start using the generated code!

Una vez finalizado el generador, tendremos dos archivos creados dentro de la carpeta `src\MDW\DemoBundle\Entity` ya que como prefijo de nuestro nombre de Entidad usamos el identificador de nuestro Bundle (`MDWDemoBundle`). El primer archivo será nuestra entidad `Articles.php` y el otro será el repositorio de esa entidad `ArticlesRepository.php` del cual hablaremos en el siguiente capítulo.

Si revisamos el contenido de estos archivos podremos ver que es una Clase con el nombre `Articles` y que contiene como propiedades los datos que hemos cargado de las columnas de la tabla incluyendo sus métodos getters y setters. También podremos ver que la información que mapea los datos de la tabla se encuentra como parte de bloques de comentarios sobre cada propiedad usando el concepto de Anotaciones, concepto muy conocido en el lenguaje Java. Esta información será utilizada en runtime para conocer datos sobre la entidad. Comentemos algunos más importantes:

- ⌚ **@ORM\Entity:** Indica que esta tabla se comportará como una entidad, es decir que mapeará una tabla de la base de datos.
- ⌚ **@ORM\Table:** Doctrine usará el nombre de nuestra entidad para crear una tabla en la base de datos con el mismo nombre y esta anotación nos permitirá especificar un nombre diferente si agregamos el parámetro `name="nombre_tabla"`.
- ⌚ **@ORM\Column:** Indica que esta propiedad mapea una columna de la tabla y como argumentos recibe datos de la misma como por ejemplo el tipo de dato y el largo en el caso de ser string. Doctrine se encargará de crear estas columnas con el tipo de dato correspondiente para cada motor de base de datos.
- ⌚ **@ORM\Id:** Indica que esta propiedad/columna será la Clave Primaria de la tabla.
- ⌚ **@ORM\GeneratedValue:** Indica que este campo numérico se irá autoincrementando usando la estrategia que pasemos por parámetro. En este caso AUTO hará que elija la mejor forma según el motor de base de datos que usemos, por ejemplo si usamos MySQL creará un índice autonómico mientras que si es PostgreSQL usará un dato de tipo SERIAL.

Ahora creamos la entidad para la tabla de comentarios:

```
C:\wamp\www\Symfony>php app\console doctrine:generate:entity
  Welcome to the Doctrine2 entity generator
  This command helps you generate Doctrine2 entities.
  First, you need to give the entity name you want to generate.
  You must use the shortcut notation like AcmeBlogBundle:Post.
  The Entity shortcut name: MDWDemoBundle:Comments
  Determine the format to use for the mapping information.
  Configuration format (yml, xml, php, or annotation) [annotation]:
  Instead of starting with a blank entity, you can add some fields now.
  Note that the primary key will be added automatically (named id).
  Available types: array, object, boolean, integer, smallint,
  bigint, string, text, datetime, datetimetz, date, time, decimal, float.
  New field name (press <return> to stop adding fields): author
  Field type [string]:
  Field length [255]:
  New field name (press <return> to stop adding fields): content
  Field type [string]: text
  New field name (press <return> to stop adding fields): reply_to
  Field type [string]: integer
  New field name (press <return> to stop adding fields):
```

Una vez que tengamos estas 2 entidades creadas podemos crear las tablas en la base de datos con el siguiente comando:

```
C:\wamp\www\Symfony>php app\console doctrine:schema:create
```

Ahora revisando la base de datos con el phpMyAdmin veremos como ambas tablas fueron creadas por nuestro proyecto y nos daremos cuenta como Doctrine, por medio de nuestras Entidades, conoce perfectamente como crearlas.

## MODIFICANDO LA ESTRUCTURA DE LAS TABLAS

Ahora que ya tenemos nuestras Entidades tenemos que crear la relación que existe entre ellas. Para este caso decimos que un Artículo puede llegar a tener varios comentarios relacionados, mientras que un comentario pertenece a un artículo específico, por lo que en la tabla de comentarios deberíamos agregar una Clave Foránea apuntando a la tabla de artículos. Para esto, agregaremos la siguiente propiedad con sus respectivos getter y setter a nuestra entidad Comment:

```
/**
 * @ORM\ManyToMany(targetEntity="Articles", inversedBy="comments")
 * @ORM\JoinColumn(name="article_id", referencedColumnName="id")
 * @return integer
 */
private $article;
public function setArticle(\Mdw\BlogBundle\Entity\Articles $article)
{
    $this->article = $article;
}
public function getArticle()
{
    return $this->article;
}
```

Con este código estamos definiendo una relación entre ambas tablas y nuevamente las anotaciones permiten a Doctrine especificar como se define la FK:

- ➊ **@ORM\ManyToMany**: Esta anotación le dice que desde esta tabla de comentarios existe una relación de muchos a uno con la entidad `Articles`.
- ➋ **@ORM\JoinColumn**: Especifica las columnas que se usarán para hacer el join. Localmente se usará una campo `article_id` y en la tabla de referencia se usará la propiedad `id`.

Con esto hemos modelado la FK desde el punto de vista de la entidad Comment, iremos a la entidad Articles a escribir la relación desde su punto de vista agregando el siguiente código:

```
/**
 * @ORM\OneToMany(targetEntity="Comments", mappedBy="article")
 */
private $comments;
```

```

public function __construct()
{
    $this->comments = new \Doctrine\Common\Collections\ArrayCollection();
}
public function addComments(\Mdw\BlogBundle\Entity\Comments $comments)
{
    $this->comments[] = $comments;
}
public function getComments()
{
    return $this->comments;
}

```

Agregando la propiedad `$comments` estamos creando la referencia a la otra tabla ya que un artículo puede tener varios comentarios usamos la anotación inversa a la que vimos anteriormente `@ORM\OneToMany` y podremos ver que agregamos un constructor que inicializa la propiedad con un objeto del tipo  `ArrayCollection`, que nos permitirá que un artículo contenga varios comentarios para así obtenerlos todos a través del método `getComments()`.

Con estas modificaciones realizadas volvamos a generar nuestras tablas, pero como ya hemos creado ambas tablas, ejecutemos primeramente para borrarlas el siguiente comando:

```
C:\wamp\www\Symfony>php app\console doctrine:schema:drop --force
```

Para luego volver a crearlas usando el comando ya conocido:

```
C:\wamp\www\Symfony>php app\console doctrine:schema:create
```

La otra manera de actualizar nuestras tablas, en caso de no poder o no querer borrarlas es usando el comando de update donde vemos la gran potencia que nos provee Doctrine enviando solo el SQL necesario para actualizar las tablas:

```
C:\wamp\www\Symfony>php app\console doctrine:schema:update --force
```

En cualquiera de los tres casos `doctrine:schema:create`, `doctrine:schema:drop` y `doctrine:schema:update` podemos usar el parámetro especial “`--dump-sql`” para que en lugar de ejecutar el SQL necesario solo nos lo muestre en la pantalla para poder controlarlo:

```
C:\wamp\www\Symfony>php app\console doctrine:schema:create --dump-sql
```

ATTENTION: This operation should not be executed in a production environment.

```
CREATE TABLE Articles (id INT AUTO_INCREMENT NOT NULL,
title VARCHAR(255) NOT NULL, author VARCHAR(255) NOT NULL,
content LONGTEXT NOT NULL, tags VARCHAR(255) NOT NULL,
created DATE NOT NULL, updated DATE NOT NULL,
slug VARCHAR(255) NOT NULL, category VARCHAR(255) NOT NULL,
PRIMARY KEY(id)) ENGINE = InnoDB;
CREATE TABLE Comments (id INT AUTO_INCREMENT NOT NULL,
author VARCHAR(255) NOT NULL, content LONGTEXT NOT NULL,
reply_to INT NOT NULL, PRIMARY KEY(id)) ENGINE = InnoDB
C:\wamp\www\Symfony>php app\console doctrine:schema:update --dump-sql
ALTER TABLE comments ADD article_id INT DEFAULT NULL;
ALTER TABLE comments ADD CONSTRAINT FK_A6E8F47C7294869C FOREIGN KEY
(article_id)
    REFERENCES Articles(id);
CREATE INDEX IDX_A6E8F47C7294869C ON comments (article_id)
C:\wamp\www\Symfony>php app\console doctrine:schema:drop --dump-sql
ALTER TABLE comments DROP FOREIGN KEY FK_A6E8F47C7294869C;
DROP TABLE articles;
DROP TABLE comments
```

## RESUMEN DE CAPÍTULO

Como vemos, el framework Doctrine, al tener los datos de la base de datos y de las tablas, nos proporciona un soporte muy potente para trabajar con ellas y eso que solo hemos visto la parte de creación, borrado y modificación de tablas. En los siguientes capítulos trabajaremos manipulando los datos de las tablas y le proporcionaremos aún más información a nuestras entidades para ayudarnos a validar los datos ingresados en nuestros campos.

## CAPÍTULO 9: MANIPULANDO DATOS CON DOCTRINE

En el capítulo anterior ya hemos configurado nuestra base de datos y mostrado como, a partir de nuestra aplicación, crear la base de datos y las tablas. En este capítulo nos concentraremos en acceder a los datos de las tablas para consultarlos, insertarlos, actualizarlos y borrarlos. Al tener nuestro ORM bien configurado y nuestros Entities mapeando las tablas veremos como fácilmente tenemos acceso a los datos de las mismas.

De ahora en más cuando nos refiramos a una tabla dentro de Symfony hablaremos de un entity o entidad ya que este último es la forma en que Symfony ve a las tablas es decir objetos. Recordemos que nuestras entidades se encuentran dentro de nuestro Bundle en de carpeta `src\MDW\DemoBundle\Entity\` y tenemos mapeadas las tablas “articles” y “comments”.

El acceso a nuestras entidades se hará por medio de un objeto de Doctrine llamado **EntityManager** que vamos a decir que sería como el administrador de las entidades y quién sabrá como interactuar con ellas. Para obtener este objeto simplemente, dentro de nuestro action lo invocamos de la siguiente manera:

```
$em = $this->getDoctrine()->getEntityManager();
```

Con esto ya tenemos la referencia a este objeto dentro de una variable “\$em”. Ahora bien, para trabajar con los datos necesitaremos de un **repositorio**. Este repositorio sería el objeto que nos permite solicitar y actualizar datos, para obtenerlo simplemente usamos el nombre lógico de nuestra entidad de esta manera:

```
$em->getRepository('MDWDemoBundle:Articles');
```

## OBteniendo DATOS

Para obtener datos de las tablas tenemos varios métodos realmente mágicos:

- ⇒ **findAll()**: Obtiene todos los registros de la tabla. Retorna un array.
- ⇒ **find()**: Obtiene un registro a partir de la clave primaria de la tabla.
- ⇒ **findBy()**: Obtiene los registros encontrados pudiendo pasar como argumentos los valores que irían dentro del WHERE. Retorna un array.

- ④ **findOneBy()**: obtiene un registro pudiendo pasar como argumentos los valores que irían dentro del WHERE.

Veamos unos ejemplos de la utilización de estos métodos:

```
$em = $this->getDoctrine()->getEntityManager();
//-- Obtenemos todos los artículos de la tabla
$articulos = $em->getRepository('MDWDemoBundle:Articles')->findAll();
//-- Obtenemos el artículo con el id igual a 5
$articulo = $em->getRepository('MDWDemoBundle:Articles')->find(5);
//-- Obtenemos el artículo cuyo slug sea "articulo-1"
$articulos = $em->getRepository('MDWDemoBundle:Articles')-
>findOneBy(array('slug' => 'articulo-1'));
//-- Obtenemos todos los artículos de autor John Doe que sean de la
categoría "Symfony"
$articulos = $em->getRepository('MDWDemoBundle:Articles')->findBy(
array(
    'author' => 'John Doe',
    'category' => 'Symfony'
)
);
```

## CASO DE EJEMPLO

Para realizar nuestros ejemplos, crearemos unas páginas para nuestras pruebas. Primeramente como ya vimos en el capítulo 3 tenemos que crear nuestras rutas en el archivo `src\MDW\DemoBundle\Resources\config\routing.yml`, por lo tanto agreguemos el siguiente código:

```
articulo_listar:
    pattern: /articulos/listar
    defaults: { _controller: MDWDemoBundle:Articulos:listar }
articulo_crear:
    pattern: /articulos/crear
    defaults: { _controller: MDWDemoBundle:Articulos:crear }
articulo_editar:
    pattern: /articulos/editar/{id}
    defaults: { _controller: MDWDemoBundle:Articulos:editar }
```

```
articulo_visualizar:  
    pattern: /articulos/visualizar/{id}  
    defaults: { _controller: MDWDemoBundle:Articulos:visualizar }  
articulo_borrar:  
    pattern: /articulos/borrar/{id}  
    defaults: { _controller: MDWDemoBundle:Articulos:borrar }
```

Como segundo paso crearemos un nuevo controlador dentro de nuestro Bundle con el nombre `ArticulosController.php`. El archivo lo crearemos dentro de `src\MDW\DemoBundle\Controller` y contendrá inicialmente el siguiente código con 5 actions para nuestras pruebas, uno por cada ruta creada anteriormente:

```
<?php  
namespace MDW\DemoBundle\Controller;  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
use MDW\DemoBundle\Entity\Articles;  
class ArticulosController extends Controller  
{  
    public function listarAction()  
    {  
    }  
    public function crearAction()  
    {  
    }  
    public function editarAction($id)  
    {  
    }  
    public function borrarAction($id)  
    {  
    }  
}
```

Ahora crearemos dos plantillas para usarlas como visualización de respuesta de nuestros actions. Los archivos los crearemos en `src\MDW\DemoBundle\Resources\views\Articulos\` con los nombres `listar.html.twig` y `articulo.html.twig`.

## ▶ LISTAR.HTML.TWIG

---

Conforme un array de artículos crea una tabla para mostrar datos:

```
<h1>Listado de Articulos</h1>
<table border="1">
  <tr>
    <th>ID</th>
    <th>Titulo</th>
    <th>Fecha de Creacion</th>
  </tr>
  {% for articulo in articulos %}
  <tr>
    <td>{{ articulo.id }}</td>
    <td>{{ articulo.title }}</td>
    <td>{{ articulo.created | date('d-m-Y') }}</td>
  </tr>
  {% endfor %}
</table>
```

## ▶ ARTICULO.HTML.TWIG

---

Conforme a un artículo muestra sus datos:

```
<h1>Articulo con ID {{ articulo.id }}</h1>
<ul>
  <li>Titulo: {{ articulo.title }}</li>
  <li>Fecha de creacion: {{ articulo.created | date('d-m-Y') }}</li>
</ul>
```

Ahora ya tenemos nuestro código inicial y así que comenzemos a trabajar con los datos.

## MANIPULANDO DATOS

### ▶ 1. INSERCIÓN DE DATOS

---

Primeramente trabajaremos en el método `crearAction()` de nuestro `ArticulosController` en donde usaremos la entidad `Articles` para insertar registros. Para esto es bueno notar que en las

primeras líneas de nuestro controlador estamos importando el namespace de la entidad con la palabra reservada “use”.

Crearemos un objeto nuevo de la manera tradicional:

```
$articulo = new Articles();
$articulo->setTitle('Artículo de ejemplo 1');
$articulo->setAuthor('John Doe');
$articulo->setContent('Contenido');
$articulo->setTags('ejemplo');
$articulo->setCreated(new \DateTime());
$articulo->setUpdated(new \DateTime());
$articulo->setSlug('articulo-de-ejemplo-1');
$articulo->setCategory('ejemplo');
```

Usando los setters insertamos los datos y pongamos atención en que no usamos el `setId()` ya que le dijimos a nuestra entidad que se encargue de generarlo por medio de la anotación `@ORM\Generate  
dValue(strategy="AUTO")`. También notemos que para asignar las fechas de creación y modificación cargamos un objeto `DateTime` nuevo agregándole una barra invertida adelante para especificar que es una clase del CORE de php ya que si no la ponemos va a esperar que importemos un namespace tal cual como lo hicimos con la clase `Articles`.

Una vez que tenemos nuestro objeto creado por medio del `EntityManager` le diremos que sea insertado con el siguiente código:

```
$em = $this->getDoctrine()->getEntityManager();
$em->persist($articulo);
$em->flush();
```

Con el método `persist()` le decimos que el objeto pasado por argumento sea guardado para ser insertado y la inserción en sí se realizará cuando ejecutemos el método `flush()`. Con esta orden Doctrine generará la sentencia `INSERT` necesaria. Finalmente vamos a invocar a nuestra plantilla a quién le pasaremos el artículo para que muestre sus datos:

```
return $this->render('MDWDemoBundle:Articulos:articulo.html.twig',
array('articulo' => $articulo));
```

## ► 2. ACTUALIZANDO DATOS

---

La actualización de datos es exactamente igual a la inserción con la diferencia que estamos modificando un objeto Articles ya existente, para este ejemplo crearemos el siguiente código en el método `editarAction($id)`:

```
$em = $this->getDoctrine()->getEntityManager();

$articulo = $em->getRepository('MDWDemoBundle:Articles')->find($id);

$articulo->setTitle('Articulo de ejemplo 1 - modificado');
$articulo->setUpdated(new \DateTime());

$em->persist($articulo);
$em->flush();

return $this->render('MDWDemoBundle:Articulos:articulo.html.twig',
array('articulo' => $articulo));
```

El método obtiene el id que llega por GET y por medio del `EntityManager` obtiene el registro y nos devuelve el objeto al cual, usando los setters asignamos los cambios y de la misma forma que la inserción ejecutamos el `persist()` y el `flush()`. Esto creará un update en lugar de un insert ya que doctrine puede identificar perfectamente que el artículo ya existe en la base de datos porque ya fue persistido con anterioridad.

## ► 3. MOSTRANDO UN LISTADO DE ARTÍCULOS

---

Para obtener todos los artículos de una entidad ya vimos el método `findAll()` por lo que tendremos el siguiente código dentro del método `listarAction()` para luego enviar el array de `Articles` a la vista que se encargará de mostrarlos en una tabla.

```
$em = $this->getDoctrine()->getEntityManager();

$articulos = $em->getRepository('MDWDemoBundle:Articles')->findAll();

return $this->render('MDWDemoBundle:Articulos:listar.html.twig',
array('articulos' => $articulos));
```

## ► 4. ELIMINAR UN ARTÍCULO

Para eliminar un artículo simplemente lo obtenemos e invocamos el método `remove()` que marcará el artículo para ser borrado hasta que ejecutemos el método `flush()`. Este código lo pondremos en el método `borrarAction()`:

```
$em = $this->getDoctrine()->getEntityManager();
$articulo = $em->getRepository('MDWDemoBundle:Articles')->find($id);
$em->remove($articulo);
$em->flush();
return $this->redirect(
    $this->generateUrl('articulo_listar')
);
```

Una vez que lo borramos, redirigimos al usuario a la ruta “`articulo_listar`”.

## FORMAS ALTERNATIVAS DE OBTENER DATOS

### 1.EXTENSIÓN DE LOS MÉTODOS FINDBY() Y FINDONEBY()

Los métodos `findBy()` y `findOneBy()` tienen otros métodos similares conocidos como `findBy*()` y `findOneBy*()` donde el asterisco representa cualquier propiedad de nuestra entidad:

```
//-- Obtenemos todos los artículos de la categoría ‘Symfony’
$articulos = $em->getRepository('MDWDemoBundle:Articles')-
>findByCategory('Symfony');

//-- Obtenemos el artículo con el slug ‘artículo-1’
$articulo = $em->getRepository('MDWDemoBundle:Articles')-
>findOneBySlug('articulo-1');
```

### 2.UTILIZANDO LAS CLAVES FORÁNEAS

Otra de las formas de obtener datos es por medio de las claves foráneas las cuales fueros configuradas en nuestra entidad por medio de los annotatios `@ManyToOne`, `@OneToMany`. Una vez que obtenemos por ejemplo un artículo podríamos obtener todos sus comentarios de la siguiente manera:

```
$em = $this->getDoctrine()->getEntityManager();
$articulo = $em->getRepository('MDWDemoBundle:Articles')-
>findOneBySlug('articulo-de-ejemplo-1');
```

```
$comentarios = $articulo->getComments();
foreach($comentarios as $c)
{
    echo $c->getContent();
}
```

Al utilizar la clave foránea configurada en nuestra entidad invocando al getter getComments(), doctrine se encargará de generar la sentencia SELECT necesaria para obtener todos los comentarios.

### 3.GENERANDO DQL

Por si las formas de obtener datos que ya vimos nos quedan cortas, cosa que por lo general es así, Doctrine nos permite trabajar con algo muy parecido al SQL estándar al que estamos acostumbrados a trabajar solo que como estamos trabajando con el ORM se llama [DQL](#)<sup>1</sup> es decir Doctrine Query Language.

El DQL es realmente muy parecido al SQL con la diferencia que en lugar de hacer queries contra registros de las tablas, los hacemos sobre objetos de tipo Entity, por ejemplo un select bien sencillo:

```
SELECT * FROM articles
en DQL sería:
select a from MDWDemoBundle:Articles a
```

donde la “a” es nada más que un simple alias que podemos llamar como queramos. El cambio principal se nota en que en lugar de hacer referencia a la tabla articles estamos haciendo referencia a la entidad `MDWDemoBundle:Articles`. Con esta sintaxis estamos dejando que doctrine se encargue de la traducción al SQL necesario para el motor de base de datos utilizado y configurado inicialmente.

También es posible pedir solo algunos campos y no un SELECT \* poniendo los nombres de las propiedades del objeto usando el alias:

```
select a.id, a.title, c.author from MDWDemoBundle:Articles a
```

Para decirle a Doctrine que ejecute este DQL lo hacemos a través del EntityManager de la siguiente manera:

```
$em = $this->getDoctrine()->getEntityManager();
```

<sup>1</sup> <http://www.doctrine-project.org/docs/orm/2.1/en/reference/dql-doctrine-query-language.html>

```
$dql = "select a from MDWDemoBundle:Articles a";
$query = $em->createQuery($dql);
$articulos = $query->getResult();
```

Con el código anterior utilizamos el DQL para generar un objeto de Doctrine llamado “Doctrine\_Query” representado por \$query y luego a este objeto le pedimos que nos devuelva los resultados invocando al getResult() lo que nos devolverá un array de objetos Articles y para acceder a sus datos simplemente utilizamos los getters del objeto. Por ejemplo si quisieramos recorrer el array de articulos y obtener el id lo haríamos así ya que siguen siendo objetos metidos dentro de un array:

```
foreach($articulos as $articulo)
{
    $id = $articulo->getId();
    $title = $articulo->getTitle();
}
```

En caso de trasladar filtros para el WHERE, lo hacemos usando el método setParameter() del objeto Doctrine\_Query de la siguiente manera:

```
$em = $this->getDoctrine()->getEntityManager();
$dql = "select a from MDWDemoBundle:Articles a where a.author=:author and
a.title like :title";
$query = $em->createQuery($dql);
$query->setParameter('author', 'John Doe');
$query->setParameter('title', '%ejemplo 1%');
$articulos = $query->getResult();
```

Con la utilización del setParameter() ya no nos preocupamos de poner, por ejemplo, comillas a los filtros que no son numéricos ya que Doctrine ya sabe de que tipo de dato es cada columna por medio de la definición que hicimos de la entidad.

También tenemos por supuesto soporte para unir entidades por medio de la cláusula JOIN por lo que este SQL estándar lo podríamos convertir a DQL de la siguiente manera:

```
$em = $this->getDoctrine()->getEntityManager();
$dql = "select a.id, a.title, c.author
       from MDWDemoBundle:Comments c
       join c.article a
       where a.author=:author"
```

```

        and a.title like :title";
$query = $em->createQuery($dql);
$query->setParameter('author', 'John Doe');
$query->setParameter('title', '%ejemplo 1%');
$articulos = $query->getResult();

```

Hay una diferencia a la hora de obtener los datos. Ya que estamos obteniendo una mezcla de datos de artículos y comentarios, el método `getResult()` nos devuelve todo ya directamente en un array como siempre estuvimos acostumbrados a trabajar con PDO por lo tanto la estructura del array devuelto sería la siguiente:

```

Array
(
    [0] => Array
        (
            [id] => 4
            [title] => Articulo de ejemplo 1
            [author] => Autor 1
        )
    [1] => Array
        (
            [id] => 4
            [title] => Articulo de ejemplo 1
            [author] => Autor 2
        )
)

```

## 4. UTILIZANDO EL REPOSITORIO

En el capítulo anterior cuando nos ocupamos de crear nuestras entidades `Articles` y `Comments` con el generador `doctrine:entity:create`, se nos hizo una pregunta sobre si queríamos que el generador nos cree un repositorio vacío para la entidad a crear y hemos dicho que sí. Esto hizo que se cree un segundo archivo a parte de la entidad llamado `NombreEntidadRepository.php`. Para nuestro ejemplo hemos creado el `ArticlesRepository.php` y el `CommentsRepository.php`.

Estos archivos se utilizan para organizar sentencias DQL de una entidad en cuestión. Por ejemplo, en lugar de tener todos los códigos de DQL escritos más arriba esparcidos por nuestros Controla-

dores, podríamos (y deberíamos para mantener el código más ordenado y mantenible) tener todas las consultas relacionadas con los artículos dentro de nuestro repositorio `ArticlesRepository.php`. Esto es muy útil ya que desde el primer capítulo hablamos que Symfony intenta mantener todas las cosas en su lugar y es realmente útil.

En este mismo momento nuestro repositorio de artículos se encuentra de la siguiente manera:

```
<?php
namespace MDW\DemoBundle\Entity;
use Doctrine\ORM\EntityRepository;
/**
 * ArticlesRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class ArticlesRepository extends EntityRepository
{}
```

Dentro de esta clase crearemos métodos que serán cada una de nuestras consultas. Fijémonos que la clase hereda de `EntityRepository` lo cual ya nos da ciertas ventajas por ejemplo que para obtener el `EntityManager` simplemente tenemos que invocarlo como `$this->getEntityManager()`. Así que podríamos tener por ejemplo un método para obtener artículos de un autor con un cierto contenido en el título creando el siguiente método:

```
public function findArticlesByAuthorAndTitle($author, $title)
{
    $em = $this->getEntityManager();
    $dql = "select a.id, a.title, c.author
            from MDWDemoBundle:Comments c
            join c.article a
            where a.author=:author
            and a.title like :title";
    $query = $em->createQuery($dql);
    $query->setParameter('author', $author);
    $query->setParameter('title', '%' . $title . '%');
    $articulos = $query->getResult();
```

```
    return $articulos;  
}
```

Una vez que tengamos nuestros métodos en el repositorio lo accedemos de la misma forma que los métodos `find()` o `findAll()` ya vistos dentro de nuestros actions en los controladores:

```
$em = $this->getDoctrine()->getEntityManager();  
$articulos = $em->getRepository('MDWDemoBundle:Articles')->findArticlesByAu  
thorAndTitle('John Doe', 'ejemplo 1');
```

En cada repositorio podemos tener la cantidad de métodos que necesitemos y hasta podríamos hacer que métodos genéricos que reutilicen otros métodos de la misma clase lo cual, si pensamos en una aplicación que puede llegar a utilizar muchas sentencias SQL lograríamos tener un código mucho más ordenado y por supuesto, también tenemos identificado donde pueden ocurrir los errores ya que cada cosa está en su lugar.

## RESUMEN DE CAPÍTULO

Si pensáramos nuevamente en la arquitectura Model-View-Controller (MVC) estaríamos viendo que la **presentación de los datos (View)** se encuentran en las plantillas, la programación **del lado de datos y trabajo con los mismos (Model)** se encuentran en las Entidades y Repositorios.

Nos queda la **lógica de la aplicación (Controller)** que se mantendrá dentro de los actions programados dentro de cada Controlador. Esto al mirarlo desde un enfoque de arquitectura de software demuestra un proyecto muy ordenado y por sobre todo con facilidades de mantener y escalar a futuro, con un equipo de desarrollo donde cada quién maneja su parte. Algo que siempre suelo decir es que si una página llega a tener más de 20 líneas de código es porque nos estamos olvidando de modularizar y es probable que una de las partes del MVC no se encuentre en el lugar correcto.

## CAPÍTULO 10: VALIDACIÓN DE DATOS Y CREACIÓN DE FORMULARIOS

Ahora que ya hemos trabajado con la base de datos y con los datos en sí vamos a tocar dos temas que serán de mucha ayuda. El primero es como el framework nos ayuda a validar los datos que vamos a grabar en nuestra base de datos y el segundo es como crear los formularios para que los usuarios finales ingresen los datos en la aplicación.

### VALIDACIÓN DE DATOS

Como ya hemos visto cada tabla de nuestra base de datos es representada por medio de un Entity en el que usamos annotations para definir los metadatos. Para usar estos annotations importamos el paquete “use Doctrine\ORM\Mapping as ORM;” arriba del archivo y usamos los mismos por medio del alias ORM de esta manera “@ORM\Id”.

Para la validación de los datos importaremos otro namespace: “use Symfony\Component\Validator\Constraints as Assert;” y por medio del alias Assert, utilizando annotations definiremos los validadores para los campos que queramos. El listado completo de validadores o también llamados constraints los puedes ver en la [documentación oficial<sup>1</sup>](#).

Algo que resulta muy importante entender es que la definición de los metadatos que escribimos usando el alias @ORM no tiene el mismo propósito que cuando usamos el @Assert. Por ejemplo, en el caso de nuestro Entity Article, hemos definido que la propiedad (campo/columna) \$title no permite nulos. Esto lo hicimos porque dejamos su mapeo como @ORM\Column(name="title", type="string", length=255) donde por defecto es not null pero esto no implica la validación de los datos ya que lo que acabamos de escribir es que para la creación de la tabla se debe tener en cuenta que no es nulo y esto sirve para generar correctamente el CREATE TABLE necesario.

Para asegurarnos de que no sea nulo a la hora de ingresar los datos debemos usar el @Assert\NotNull() cuyo objetivo es realmente decirle al validador de datos que efectivamente al intentar grabar datos por medio del entity este debe ser validado como que no permite valores nulos.

Estos annotations tienen la misma forma que los anteriores. Son llamadas a métodos que pueden

<sup>1</sup> <http://symfony.com/doc/current/book/validation.html#constraints>

recibir parámetros opcionales. Por ejemplo, si ponemos:

```
/**
 * @var string $title
 *
 * @ORM\Column(name="title", type="string", length=255)
 * @Assert\NotNull()
 */
private $title;
```

estamos diciendo que la propiedad title no debe permitir valores nulos y al momento de validarlos saldrá una mensaje diciendo eso en inglés, si queremos cambiar el mensaje por defecto lo hacemos agregando el argumento a la invocación de esta manera:

```
/**
 * @var string $title
 *
 * @ORM\Column(name="title", type="string", length=255)
 * @Assert\NotNull(message="Debe escribir un titulo")
 */
private $title;
```

Si quisiéramos validar que un campo debe ser de tipo email usaremos el annotation

`@Assert\Email()` de esta manera:

```
/**
 * @Assert\Email(
 *     message = "El mail '{{ value }}' ingresado no tiene el formato
 * correcto.",
 * )
 */
protected $email;
```

Haciendo referencia a `{{ value }}` va a mostrar el valor ingresado como parte del mensaje.

Como último ejemplo, si quisiéramos validar la máxima cantidad de caracteres ingresados, podríamos usar el `@Assert\MaxLength()`:

```
/**
```

```

 * @var string $title
 *
 * @ORM\Column(name="title", type="string", length=255)
 * @Assert\NotNull(message="Debe escribir un titulo")
 * @Assert\MaxLength(255)
 */
private $title;

```

Y si quisieramos además de la máxima cantidad de caracteres, controlar la mínima cantidad simplemente lo agregamos también:

```

 /**
 * @var string $title
 *
 * @ORM\Column(name="title", type="string", length=255)
 * @Assert\NotNull(message="Debe escribir un titulo")
 * @Assert\MaxLength(255)
 * @Assert\MinLength(5)
 */
private $title;

```

Con esto ya estamos controlando que mínimamente debemos escribir 5 caracteres en el título y como máximo 255.

Nota: Cuando usamos el `@Assert\MaxLength()`, la cantidad de caracteres que permitimos debe ser menor o igual al `length` definido en el `@ORM\Column()` ya que de lo contrario la aplicación dejaría pasar valores mayores y al llegar a la base de datos nos devolvería un error pero del motor de datos.

Como ya había mencionado más arriba, en la [documentación oficial<sup>1</sup>](#) tenemos los constraints soportados y si damos click sobre cada uno de ellos veremos como se utilizan con un ejemplo. Entre ellos encontrarán `NotNull`, `NotBlank`, `Email`, `MinLength` y `MaxLength` (para cantidad de caracteres), `Max` y `Min` (para valores numéricos), `Date`, `DateTime`, `Time`, `Choice` (para campos que serán ingresados por medio de un combo de valores por ejemplo).

Al escribir los Asserts en realidad estamos configurando las validaciones que queremos tener pero para validar los datos debemos invocar al validador. Para esto usemos como base el ejemplo que

<sup>1</sup> <http://symfony.com/doc/current/book/validation.html#constraints>

teníamos para la inserción de artículos del capítulo anterior:

```
public function crearAction()
{
    $articulo = new Articles();
    $articulo->setTitle('Articulo de ejemplo 1');
    $articulo->setAuthor('John Doe');
    $articulo->setContent('Contenido');
    $articulo->setTags('ejemplo');
    $articulo->setCreated(new \DateTime());
    $articulo->setUpdated(new \DateTime());
    $articulo->setSlug('articulo-de-ejemplo-1');
    $articulo->setCategory('ejemplo');
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($articulo);
    $em->flush();
    return $this->render('MDWDemoBundle:Articulos:articulo.html.twig',
array('articulo' => $articulo));
}
```

En código anterior, sin validar nada y pasando por alto los constraints de nuestro Entity intenta grabar los datos y si por ejemplo no cargamos un dato obligatorio como el título, el error devuelto será el que la misma base de datos valida ya que la columna fue creada como not null pero lo que queremos es obtener la validación en la aplicación, antes que llegue el insert a la base de datos y esto lo haremos por medio del validador agregando el siguiente código antes de la invocación al EntityManager:

```
$errores = $this->get('validator')->validate($articulo);
```

Por medio del objeto `$this->get('validator')` le decimos que valide la entidad `$articulo`, quien ya sabe como validarse por sí misma ya que los annotations están dentro de la misma. Este método `validate()` nos devolverá un array de errores que podemos iterar y obtenerlos por medio del método `getMessage()`:

```
public function crearAction()
{
    $articulo = new Articles();
    //-- No cargamos el dato para title
    $articulo->setAuthor('John Doe');
```

```
$articulo->setContent('Contenido');
$articulo->setTags('ejemplo');
$articulo->setCreated(new \DateTime());
$articulo->setUpdated(new \DateTime());
$articulo->setSlug('articulo-de-ejemplo-1');
$articulo->setCategory('ejemplo');
$errores = $this->get('validator')->validate($articulo);
if(!empty($errores))
{
    foreach($errores as $error)
        echo $error->getMessage();
    return new Response();
}
$em = $this->getDoctrine()->getEntityManager();
$em->persist($articulo);
$em->flush();
return $this->render('MDWDemoBundle:Articulos:articulo.html.twig',
array('articulo' => $articulo));
}
```

En el código de arriba hemos borrado la línea del `setTitle()`. Esto nos mostrará en pantalla el mensaje “Debe escribir un título” y si tenemos más errores los mensajes por cada error.

Ahora bien, realmente no tiene mucho sentido mostrar de esta manera los mensajes de errores ya que finalmente ni siquiera cargamos los datos a mano como lo estamos haciendo, sino que son ingresados por un formulario y es aquí donde pasamos al siguiente tema, la creación de formularios.

## CREACIÓN DE FORMULARIOS

Los formularios no se escriben en HTML sino que son programados como objetos y el mismo framework se encarga de hacer render del HTML necesario y asegurándonos que serán escritos de la mejor manera posible incluso utilizando las validaciones de [HTML5<sup>1</sup>](#).

Un formulario siempre debería ser representado por un objeto que se lo conoce como **Type**. Este objeto hace referencia a otro que puede ser un Entity (del que ya hablamos en los capítulos anteriores).

<sup>1</sup> <http://www.maestrosdelweb.com/guias/#guia-html5>

res) o un POPO (Plain Old PHP Object).

---

**Nota:** Un POPO (Plain Old PHP Object) es simplemente una clase con propiedades y métodos tradicionales, es decir que es muy parecido a los Entities pero sin los annotations. Por ejemplo en caso de ser una clase que representará a un formulario para contacto donde tendremos simplemente una propiedad asunto, email, nombre y texto con sus respectivos setters y getters.

Un objeto **Type** se debe tomar como la definición del formulario. Este objeto recibirá cual es el Entity o POPO en el cual se almacenan los datos cargados en el formulario. Podríamos tener más de un Type para un mismo objeto ya que dependiendo de ciertos perfiles por ejemplo, podríamos mostrar algunos campos u otros dependiendo de que el usuario sea operador normal o administrador.

## DEFINICIÓN DE NUESTRO FORMULARIO

Para nuestro ejemplo tomaremos en cuenta el Entity Article que venimos usando y crearemos un objeto Type para representar a este formulario. Los formularios se deben crear dentro de nuestro Bundle en una carpeta `Form` por lo que crearemos el archivo `ArticleType` dentro de nuestra carpeta `src/MDW/DemoBundle/Form`:

```
<?php
namespace MDW\DemoBundle\Form;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
class ArticleType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('title')
            ->add('author')
            ->add('created');
    }
    public function getName()
    {
        return 'article_form';
    }
}
```

Como podemos ver en el ejemplo, el nombre de la clase esta formado por un nombre que yo he elegido concatenado con el sufijo Type y debe heredar de AbstractType para contener las funcionalidades base.

En el método `buildForm()`, por medio del `$builder`, que se recibe como argumento, agregamos los campos que vamos a usar. Estos campos son los nombres de los campos que tendrá el formulario y deben coincidir con las propiedades de nuestro Entity Article aunque todavía no hemos dicho que el formulario representará a Article ya que eso lo hacemos en la invocación desde el controller. El argumento `options` nos servirá para crear el formulario con otras opciones de personalización.

El método `getName()` deberá retornar el identificador de nuestro formulario y este `String` puede tomar el nombre que queramos siempre y cuando sea único. Debemos tomar en cuenta que este nombre será usado para los atributos “name” de los componentes de formularios. Por ejemplo vemos que tenemos un componente llamado “title” que hemos agregado en el método `buildForm()` por lo que la etiqueta creada será:

```
<input type="text" name="article_form[title]" />
```

Hay que notar que esta es la sintaxis para cargar datos en un array (usando los corchetes) por lo que “`article_form`” será simplemente un array con una clave asociativa “`title`” que contendrá el valor ingresado por el usuario. Esta sintaxis nos permite tener en un array todos los datos del formulario al hacer el submit.

Con esto lo que hemos hecho es crear la representación básica de nuestro formulario, diciéndole cual es el identificador del formulario y los campos que deberá contener.

---

**Nota:** Escribiendo los objetos Type NO definimos como será visualmente el formulario sino como será **CONCEPTUALMENTE**.

## INVOCACIÓN Y VISUALIZACIÓN DEL FORMULARIO

Para mostrar el formulario HTML en nuestra página debemos invocar a nuestra clase `ArticleType` desde nuestro controlador o más específicamente desde el action que llama a nuestra página, para esto vamos a crear un action nuevo dentro de nuestro `ArticulosController` al que vamos a llamar `newAction`.

Primeramente creamos nuestra ruta en el archivo `routing.yml`

```
articulo_new:
    pattern: /articulo/new
    defaults: { _controller: MDWDemoBundle:Articulos:new }
```

Una vez creada nuestra ruta iremos a crear nuestro `newAction` en `src\MDW\DemoBundle\Controller\ArticulosController.php` (`MDWDemoBundle:Articulos:new`). Para esto agregamos el siguiente código:

```
//-- Al estar utilizando la clase ArticleType dentro de nuestro método no
debemos olvidar importar el namespace al principio del archivo
use MDW\DemoBundle\Form\ArticleType;
//-- Agregar este método como uno nuevo
public function newAction()
{
    $articulo = new Articles();
    $form = $this->createForm(new ArticleType(), $articulo);
    return $this->render('MDWDemoBundle:Articulos:new.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

---

**Nota:** Un dato importante es que en el código de arriba hemos creado un nuevo `$articulo` desde un objeto vacío lo cual hará que el formulario se muestre vacío. Si queremos, por ejemplo en un formulario de modificación de registro, mostrar ya los datos del artículo a modificar esto simplemente implicaría obtener los datos desde la base de datos utilizando un DQL o el método `find()` que vimos en el capítulo anterior antes de pasarlo al método `createForm()`.

El código que contendrá nuestro `action` es muy sencillo. Primeramente creamos un objeto `Article` y luego, por medio del método `$this->createForm()` invocamos a nuestro objeto `ArticleType` pasándole nuestro objeto recién creado `$articulo`, devolviéndonos un objeto de tipo formulario. Finalmente invocamos a la vista como siempre hacemos y pasamos como parámetro el resultado de ejecutar `$form->createView()`.

Con esto ya seremos capaces de ver el código de nuestra vista `MDWDemoBundle:Articulos:new.html.twig` que de acuerdo a este nombre lógico debemos crear el archivo `new.html.twig` dentro de

la carpeta `src/MDW/DemoBundle/Resources/views/Articulos/` con el siguiente código:

```
<form action="{{ path('articulo_new') }}" method="post">
    {{ form_widget(form) }}
    <input type="submit" />
</form>
```

La creación de la etiqueta formulario la hacemos normalmente así como también el botón de submit. Lo único importante aquí es que el action del form debe apuntar a la misma página por lo que creamos el link por medio de `path('articulo_new')`.

La parte mágica está en `{{ form_widget(form) }}` donde, por medio de `form_widget` y Twig, pasamos como argumento la variable que nuestro action nos ha enviado y se imprime en la página el código necesario para nuestro formulario. Es decir que veremos el formulario al ingresar a la dirección: [http://localhost/Symfony/web/app\\_dev.php/articulo/new](http://localhost/Symfony/web/app_dev.php/articulo/new)

The screenshot shows a web form with the following fields:

- Title:** A text input field.
- Author:** A text input field.
- Created:** A date input field with dropdown menus for year (2007), month (01), and day (01).
- Enviar consulta:** A blue submit button.

Si miramos el código HTML veremos lo siguiente:

```
<form action="/Symfony/web/app_dev.php/articulo/new" method="post">
    <div id="article_form">
        <input type="hidden" id="article_form_token" name="article_form[_token]" value="62bc1a503b32de46b8755e9a5f5d8855bc8eb877" />
        <div>
            <label for="article_form_title" class="required">Title</label>
            <input type="text" id="article_form_title" name="article_form[title]" required="required" maxlength="20" />
        </div>
        <div>
            <label for="article_form_author" class="required">Author</label>
```

```

label>
    <input type="text" id="article_form_author" name="article_
form[author]" required="required" maxlength="255" />
</div>
<div>
    <label class=" required" >Created</label>
    <div id="article_form_created">
        <select id="article_form_created_year" name="article_
form[created][year]" required="required">
            <option value="2007">2007</option>
            <option value="2008">2008</option>
            ...
        </select>
    </div>
</div>
<input type="submit" />
</form>

```

**Nota:** Muy importante es notar que a parte de los campos que hemos agregado para que sean mostrados en el \$builder, también se muestra un campo `article_form[_token]` con un valor aleatorio. Esto lo hace automáticamente para luchar contra uno de los ataques más usados por los hackers llamado [CSRF<sup>1</sup>](#). Con eso ya vemos como Symfony nos propone ya un estándar de seguridad. A esta seguridad también se suma que por medio de Doctrine tenemos validado los problemas de SQL Injection.

Si vemos el código notamos los atributos “name” como los explicamos arriba y también vemos que mágicamente el campo “created” se muestra como un campo para seleccionar una fecha. Esto es debido a que el framework reconoce el tipo de input a mostrar ya que sabe, por medio del objeto Articles, que esa propiedad es una fecha.

Esto es tremadamente útil ya que muchas veces podría ya reconocer que type agregarle a las etiquetas input, pero si necesitamos definir por nosotros mismos el atributo type lo hacemos agregando un segundo argumento al momento de agregar el campo al \$builder:

```
public function buildForm(FormBuilder $builder, array $options)
```

<sup>1</sup> [http://es.wikipedia.org/wiki/Cross\\_Site\\_Request\\_Forgery](http://es.wikipedia.org/wiki/Cross_Site_Request_Forgery)

```
{
    $builder->add('title')
        ->add('author', 'checkbox')
        ->add('created');
}
```

Mientras que si necesitamos hacer que un campo no sea obligatorio lo hacemos enviando un array como tercer argumento ya que por defecto todos los campos son puestos como requeridos con validaciones HTML5:

```
public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('title')
        ->add('author', 'text', array('required' => false))
        ->add('created');
}
```

Como vemos el formulario HTML es impreso directamente en la página usando el `{} form_widget(form) {}` incluyendo divs que nos ayudarán a formatear por medio de CSS y mejorar la estructura de mismo pero en caso de querer crear formularios más complejos en diseño también se cuentan con las siguientes opciones:

- ⌚ **form\_errors(form)**: Renderiza los errores que se encuentren en el formulario.
- ⌚ **form\_rest(form)**: Renderiza los campos de formulario que no hayan sido agregados manualmente con el `form_row`.
- ⌚ **form\_row(form.field)**: Renderiza un campo específico dentro de un div.
- ⌚ **form\_errors(form.field)**: Renderiza el error para un campo específico.
- ⌚ **form\_label(form.field)**: Renderiza la etiqueta label para un campo específico.
- ⌚ **form\_widget(form.field)**: Renderiza un campo específico.

## PROCESAMIENTO DEL FORMULARIO

Ahora que ya vimos como mostrar el formulario en la página y habiendo dicho el `action` de un `form` va al mismo `action` para ser procesado, entremos en detalle de las modificaciones que tenemos que tener en cuenta en el código original dentro del método `newAction()`.

Lo primero que tenemos que pensar es que si para procesar el formulario llamamos al mismo action, ¿Cómo sabemos cuándo mostrar el formulario y cuándo procesarlo?. La respuesta es bien sencilla, cuando el request fue de tipo GET lo deberíamos de mostrar pero en caso de que se haya dado click en el botón submit se ejecuta un request de tipo POST y por lo tanto se debería procesar. Veamos el código modificado de nuestro newAction():

```
public function newAction()
{
    //--- Obtenemos el request que contendrá los datos
    $request = $this->getRequest();
    $articulo = new Articles();
    $form = $this->createForm(new ArticleType(), $articulo);
    //--- En caso de que el request haya sido invocado por POST
    //    procesaremos el formulario
    if($request->getMethod() == 'POST')
    {
        //--- Pasamos el request el método bindRequest() del objeto
        //    formulario el cual obtiene los datos del formulario
        //    y los carga dentro del objeto Article que está contenido
        //    también dentro del objeto Type
        $form->bindRequest($request);
        //--- Con esto nuestro formulario ya es capaz de decirnos si
        //    los dato son válidos o no y en caso de ser así
        if($form->isValid())
        {
            //--- Procesamos los datos que ya están automáticamente
            //    cargados dentro de nuestra variable $articulo, ya sea
            //    grabándolos en la base de datos, enviando un mail, etc
            //--- Finalmente, al finalizar el procesamiento, siempre es
            //    importante realizar una redirección para no tener el
            //    problema de que al intentar actualizar el navegador
            //    nos dice que los datos se deben volver a reenviar. En
            //    este caso iremos a la página del listado de artículos
            return $this->redirect($this->generateURL('articulos'));
        }
    }
    return $this->render('MDWDemoBundle:Articulos:new.html.twig', array(
```

```
    'form' => $form->createView(),
);
}
```

Como vemos en las explicaciones del código casi todo es automáticamente realizado por el objeto `ArticleType` quién al conocer el request ya nos devuelve el mismo objeto original `$articulo` que le fue entregado en el `createForm(new ArticleType(), $articulo);`

En caso de que los datos no sean válidos y el método `isValid()` retorne `false` seguirá hasta mostrar nuevamente el formulario llamando al método `$this->render()` y el `\{\{ form_widget(form) \}\}` puesto en nuestra misma vista se encargará de mostrar los errores de validación.

---

**Nota:** Symfony2 agrega las validaciones de los formularios en HTML5 y del lado del servidor. Si el navegador no soporta las validaciones por medio de HTML5 el método `isValid()` lo valida en el servidor y al retornar la respuesta por el método `render()` se mostrarán los mensajes de validación del servidor. Puede que tu navegador ya acepte las validaciones HTML5 por lo que al intentar enviar los datos no notes la validación del lado del servidor aunque lo mismo se están realizando.

Por ejemplo el campo `$title` está puesto como `<input type="text" id="article_form_title" name="article_form[title]" required="required" maxlength="255" pattern=".{10,255}" />` donde se puede ver que las validaciones de HTML5 fueron ya puestas.

Si no tienes un navegador que NO soporte HTML5 para probar como se muestran los mensajes de validación del servidor puedes, utilizando el Firebug del Firefox, eliminar el texto `required="required" maxlength="255" pattern=".{10,255}"` de la etiqueta `input` y luego presionar el botón de `submit`:-)

Como verás, los hackers que quieren usar esta técnica también serán detenidos por las validaciones del servidor.

## RESUMEN DE CAPÍTULO

Hemos trabajado muchísimo viendo dos temas sumamente importantes: la validación de los Entities y los formularios.

Para las validaciones hemos hablado sobre los `@Asserts`, simples anotaciones que realizan validaciones poderosas con poco código y vemos que Symfony2 ya nos provee de la gran mayoría que necesitaremos usar.

Hablando sobre los formularios hemos notado la gran diferencia de diseñar los formularios y programar los formularios por medio de clases. Me gusta decir que en Symfony, el concepto de un formulario NO es simplemente introducción de texto sino introducción de texto VÁLIDO para la aplicación libre de los problemas que hoy se tienen al crear un formulario a mano y tener que recordar pelear con ataques CSRF, XSS, SQL Injection y cambios en caliente con herramientas como Firebug.

El sub-framework de formularios es uno de los que más me hicieron sentir la diferencia entre usar un framework y no hacerlo y todavía hay muchas otras herramientas que nos permite usar como los formularios embebidos.

En el primer capítulo del curso hablamos sobre que uno de los objetivos de Symfony es plantear que cada cosa debe ir en su lugar, respetando el concepto del MVC. Con esto vemos que no solo podríamos tener un equipo de desarrollo, con personas expertas en cada área, trabajando con el modelado, otras con los controladores y a los diseñadores en la vista, sino que también podríamos hablar de personas que trabajen netamente en la creación de los formularios de la aplicación.

## CAPÍTULO 11: INTEGRANDO AJAX

En la actualidad muchos de nuestros proyectos están orientados hacia la [web 2.0<sup>1</sup>](#) y es por esto que la necesidad de implementar [AJAX<sup>2</sup>](#) (XmlHttpRequest) es cada vez más grande, desde su versión 1.3 el proyecto Symfony a optado por no apoyar (ni integrar) ningún framework de desarrollo Frontend (básicamente en lo que se refiere Javascript), pero eso no impide que puedas utilizar cualquier framework Javascript en tu proyecto, de hecho Symfony2 es compatible con Assetic y te permite, entre otras cosas, utilizar el YUI-compressor para optimizar los assets de tu proyecto web.

En este capítulo nos concentraremos en las herramientas básicas que provee Symfony2 para manipular y detectar peticiones AJAX, las cuales son completamente transparentes para el cliente web (o framework que utilices), de esta forma podrás implementar AJAX de la forma que quieras y con el Framework JS que deseas.

### DETECTANDO PETICIONES XMLHTTPREQUEST DESDE EL CONTROLADOR

La clase Request contiene un función para verificar si la petición HTTP fue enviada por AJAX, es decir por medio del XMLHttpRequest:

```
// retorna true o false  
$this->getRequest()->isXmlHttpRequest();
```

¡Así de simple!, con ello puedes comprobar de forma efectiva desde tus controladores si la petición fue enviada por AJAX, lo que te permite, entre otras cosas, renderizar una plantilla específica, crear un objeto Response y controlar una salida personalizada como por ejemplo un JSON o XML.

### DETECTANDO PETICIONES AJAX DESDE LA VISTA

Controlar AJAX desde el controlador es muy efectivo, pero a veces por la estructura interna de nuestras plantillas (las que usas como layout y las que heredan un layout con la estructura HTML) puede resultar tedioso modificar cada controlador para devolver una plantilla específica, y ¿si dicha plantilla extiende un layout HTML (<html>, <head> y <body>)?: sabemos muy bien que en el caso de AJAX

1 <http://www.maestrosdelweb.com/editorial/web2/>

2 <http://www.maestrosdelweb.com/editorial/ajax/>

solo necesitamos el fragmento HTML específico, no un árbol HTML completo, por suerte en TWIG podremos hacer esto:

```
{% extends app.request.isXmlHttpRequest ? "MDWDemoBundle::layout_ajax.html.twig" : "::base.html.twig" %}
```

De esta forma Twig nos permite comprobar si la petición es AJAX y de este modo tener un layout específico para cada situación.

En el caso de plantillas con PHP, simplemente podemos acceder al objeto Request gracias al contenedor de Inyección de Dependencias (DI) en nuestras vistas:

```
$this->container->get('request')->isXmlHttpRequest()
```

Note que se hace una llamada a `$this->container` esto motivado a que en las plantillas `$this->get()` **intentará cargar los Helpers**, con ello especificamos que queremos cargar explícitamente un servicio, en nuestro caso el “Request”.

## CAPÍTULO 12: INTEGRANDO JQUERY

Como mencioné anteriormente Symfony2 es un Framework PHP orientado al desarrollo en el servidor, por su parte AJAX es una técnica que se implementa desde el JavaScript (JS) cliente, cuyo único objetivo es realizar las peticiones HTTP desde JavaScript y obtener la respuesta para manipularla directamente, sea añadiéndola al DOM o lo que quieras con JavaScript, por lo cual en dicho caso tanto Symfony2 como PHP sólo pueden detectar si la petición fue realizada por dicha técnica, razón por la cual su implementación es realmente simple; también aclaramos que desde su versión 1.3 el proyecto Symfony ha optado por no apoyar ni integrar ningún Framework JS, debido a que ello queda a elección del programador.

En este capítulo utilizaremos a jQuery como el Framework JS a utilizar para los ejemplos, por ser uno de los más populares y fáciles de implementar, reiteramos que puedes usar el FW JS que deseas y queda bajo tu absoluta elección, además aclaro que los ejemplos se concentran en el uso de AJAX con jQuery y que para conservar la facilidad con la que implementen los ejemplos no se usaron modelos reales de doctrine, en su caso arrays multidimensionales, si quieren complementarlos con Doctrine.

### EJEMPLO 1: EJECUTANDO UNA LLAMADA AJAX CON JQUERY

Descargamos el script de la página de jQuery y la guardamos en el directorio **web\js\**. Para este caso la versión actual es jquery-1.7.2.min.js.

Importamos el archivo dentro del template base que se encuentra en `app\Resources\views\base.html.twig`:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"
    />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
        <script src="{{ asset('js/jquery-1.7.2.min.js') }}"></script>
```

```

<script>
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>

```

Con esto ya tenemos el soporte para jQuery en todas nuestras páginas que hereden mediante Twig a este template.

Ya teníamos la página `http://localhost/Symfony/web/app_dev.php/articulos/listar` que nos mostraba una lista de artículos de la base de datos por lo que podemos hacer otra que simplemente llame por ajax a esta página y muestre los artículos. Para esto creamos la página ver-articulos que simplemente tendrá una link para cargar por ajax el contenido de la página que contiene la lista de artículos. Como cada vez que necesitamos crear una página hacemos los 3 pasos:

## ► 1. CREAMOS LA RUTA

---

Agregamos a nuestro archivo `routing.yml` las siguientes líneas para crear la nueva ruta:

```

ver_articulos:
    pattern: /ver-articulos
    defaults: { _controller: MDWDemoBundle:Articulos:verArticulos }

```

## ► 2. CREAMOS EL ACTION

---

```

//Dentro del controlador src\MDW\DemoBundle\Controller\
ArticulosController.php agregamos el siguiente action
public function verArticulosAction()
{
    return $this->render('MDWDemoBundle:Articulos:ver_articulos.html.
twig', array());
}

```

Como vemos lo único que hace es llamar a la vista que creamos a continuación.

## ► 3. CREAMOS LA VISTA

---

Creamos el archivo ver\_articulos.html.twig dentro de la carpeta

```
src\MDW\DemoBundle\Resources\views\Articulos
{% extends '::base.html.twig' %}
{% block title %}Symfony - AJAX{% endblock %}
{% block body %}
<div id="articulos"></div>
<a id="link_articulos" href="#">Cargar artículos</a>
{% endblock %}
{% block javascripts %}
<script>
$(document).ready(function(){
    $('#link_articulos').click(function(){
        $('#articulos').load('{{ path('articulo_listar') }}');
    });
});
</script>
{% endblock %}
```

En esta página hemos heredado el contenido de template base con el extend y hemos incluído los bloques. Vemos que en el body lo único que tenemos es un div vacío con id “articulos” y link con id “link\_articulos” para obtenerlo utilizamos jQuery. La idea es que al ingresar a la página solo nos muestre el link y no así los artículos. Al dar click sobre el link “Cargar Artículos” se ejecutará una llamada ajax mediante jQuery y cargaremos asincrónicamente la ruta {{ path('articulo\_listar') }} que sería la página que ya tenemos lista y que vimos en el capítulo 9.

## ► ENTENDAMOS EL CÓDIGO JQUERY:

---

```
<script>
$(document).ready(function(){
    $('#link_articulos').click(function(){
        $('#articulos').load('{{ path('articulo_listar') }}');
    });
});
</script>
```

Primeramente registramos el evento ready para que ejecute la función anónima una vez que todo el DOM sea cargado.

Una vez ejecutado el evento ready, agregamos una acción al evento click de nuestro link que ejecutará la función que carga la segunda página:

```
$('#link_articulos').click(function(){
    $('#articulos').load('{{ path('articulo_listar') }}');
});
```

La función load() de jQuery se ejecuta sobre el div vacío para que el resultado de la respuesta Ajax se cargue dentro de este div y como argumento del método pasamos la dirección en donde, para no escribir la URL a mano, usamos la función Twig {{ path('articulo\_listar') }} para que los genere la ruta relativa de la ruta articulo\_listar.

Con esto ya podemos acceder a la página [http://localhost/Symfony/web/app\\_dev.php/ver-articulos](http://localhost/Symfony/web/app_dev.php/ver-articulos) donde veremos un link “Cargar artículos”. Presionando el link veremos como se obtiene, sin recargar la página, el listado de artículos de la base de datos.

## EJEMPLO 2: GESTIONANDO LLAMADAS AJAX CON JQUERY, TWIG Y HERENCIA EN 3 NIVELES

En el ejemplo anterior apreciamos lo sencillo que es implementar una llamada AJAX por medio del FW jQuery y comprendimos que en realidad Symfony2 interviene prácticamente en nada porque su alcance se limita al desarrollo del lado del servidor, en este ejemplo práctico haremos uso de las facilidades que brinda Symfony para detectar peticiones AJAX y modificar la respuesta en función de las necesidades.

Para este ejemplo crearemos una sección o módulo de noticias, en donde nos aparece un listado principal de noticias de las cuales al hacer click nos redirigirá al detalle de la noticia como tal. La idea es conservar un enlace directo a cada noticia, con el cual un motor de búsqueda pueda indexar y a su vez cargar el detalle de la noticia en una capa por medio de AJAX (con load jQuery) con el objetivo de que el usuario pueda cargar las noticias sin recargar la página y si en caso llega desde un buscador pueda apreciar la noticia y el resto de contenidos que ofrezca nuestro layout principal.

Para ello creamos primero nuestro layout:

```

{# /src/MDW/DemoBundle/views/layout.html.twig #}
{% extends '::base.html.twig' %} {# extendemos del layout por defecto #}
{% block javascripts %}
{# añadimos la CDN de jQuery o en su defecto lo descargamos e incluimos
con:
<script src="{{ asset('js/jquery-1.7.2.min.js') }}"></script>
#}
<script type="text/javascript" src="http://code.jquery.com/jquery-
1.7.2.min.js"></script>
{% endblock javascripts %}
{# creamos una estructura para el layout general #}
{% block body %}
<div>
    <h1>*** Web de Noticias ***</h1>
    {# según el patrón de 3 niveles, creamos el bloque de contenido #}
    {% block content %}{% endblock content %}
</div>
{% endblock body%}

```

Procedemos ahora a crear las rutas hacia nuestro controlador, abre el `src/MDW/DemoBundle/Resources/config/routing.yml` y agrega:

```

# /src/MDW/DemoBundle/Resources/config/routing.yml
MDWDemoBundle_noticias:
    pattern:  /noticias
    defaults: { _controller: MDWDemoBundle:Notice:index }
MDWDemoBundle_noticeView:
    pattern:  /leerNoticia/{notice_id}
    defaults: { _controller: MDWDemoBundle:Notice:noticeView }

```

Procedemos ahora a crear el controlador, en este ejemplo utilizaremos como Modelo un array de noticias, para enfocarnos en el uso de AJAX:

```

<?php
// src/MDW/DemoBundle/Controller/NoticeController.php
namespace MDW\DemoBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
class NoticeController extends Controller

```

```

{
    // tenemos un array con los datos básicos
    private $array_notice = array(
        array(
            'title' => 'Titulo de noticia 0',
            'content' => 'Contenido de noticia 0'
        ),
        array(
            'title' => 'Titulo de noticia 1',
            'content' => 'Contenido de noticia 1'
        ),
    );
    public function indexAction()
    {
        // suponiendo que obtenemos del modelo el listado de noticias
        return $this->render('MDWDemoBundle:Notice:index.html.twig',
array(
            'notices' => $this->array_notice
       ));
    }
    public function noticeViewAction($notice_id)
    {
        //obtenemos la noticia del modelo, en este ejemplo proviene de
        //un array
        $notice = $this->array_notice[$notice_id];
        return $this->render('MDWDemoBundle:Notice:noticeView.html.
twig', array(
            'notice' => $notice
       ));
    }
}

```

Procedemos ahora a crear las vistas principales:

```

"># src/MDW/DemoBundle/Resources/views/Notice/index.html.twig #
{% extends 'MDWDemoBundle::layout.html.twig' %}

{% block content %}

```

```
<div>
    <p>Noticias recientes</p>
    <ol>
        {% for index,notice in notices %}
            <li><a href="{{ path('MDWDemoBundle_noticeView', {'notice_id': index}) }}>{{notice.title}}</a></li>
        {% endfor %}
    </ol>
    <div id="notice_viewer">
        {# en esta capa serán cargadas las noticias por ajax #}
    </div>
</div>
{% endblock content %}
{# extendemos el bloque javascript #}
{% block javascripts %}
{{parent()}} {# incluimos las declaraciones de script del layout, como
jQuery #}
<script type="text/javascript">
{# añadimos una función al evento click de todos los enlaces a.notice_
link, para
usar AJAX en vez de su comportamiento por defecto #}
$(document).ready(function(){
    $('a.notice_link').click(function(event){
        event.preventDefault(); //cancela el comportamiento por defecto
        $('#notice_viewer').load($(this).attr('href')); //carga por ajax a
        la capa "notice_viewer"
    });
});
</script>
{% endblock javascripts %}
```

Como puede notar en index.html.twig se ha extendido el bloque JavaScript para añadir la carga por medio de jQuery.load (AJAX) hacia la capa DIV “notice\_viewer”, esto con el objetivo de que si un buscador indexa nuestra página pueda hallar los links íntegros de las noticias sin afectar al SEO, además de que nos permite cargar por AJAX el contenido de nuestras noticias directamente en la capa asignada.

```
{# src/MDW/DemoBundle/Resources/views/Notice/noticeView.html.twig #}
{# note que en este caso utilizamos el layout de ajax para así no cargar
todo el contenido del layout general #}
{% extends app.request.isXmlHttpRequest ? "MDWDemoBundle::layout_ajax.
html.twig" : "MDWDemoBundle::layout.html.twig" %}
{% block content %}
<div>
    <h2>{{notice.title}}</h2>
    <p>{{notice.content}}</p>
</div>
{% endblock content %}
```

Vemos ahora que en noticeView.html.twig se hace una bifurcación de los layouts para cuando se trata de una petición AJAX, en la cual se utiliza el layout principal cuando el enlace es accedido directamente (origen de un buscador, o de un usuario con Javascript desactivado) y al contrario si proviene de AJAX se utiliza un layout especial:

```
{# /src/MDW/DemoBundle/views/layout_ajax.html.twig #}
{# como puede apreciar, el layout para ajax sólo debe incluir el bloque
contenido,
adicionalmente podemos añadir extras #}
<div>
    <strong>Visor de Noticias</strong>
    {% block content %}{% endblock content %}
</div>
```

De esta forma con Symfony podemos adaptar las respuestas en función de si la petición es AJAX o no y en este caso devolver sólo el contenido necesario, debido a que en una petición AJAX no es necesario devolver la estructura completa HTML como en una petición normal, sino el fragmento de código que nos interesa.

## RESUMEN DE CAPÍTULO

Como bien se explica, las interacciones Ajax no son parte del framework Symfony ya que para esto usamos JavaScript mientras que Symfony es un framework PHP. Existiendo tantas librerías bien robustas para manejo de Ajax como por ejemplo jQuery, incluimos la que más nos guste y ejecutamos la llamada al ajax. La manera de trabajar con librerías JavaScript en Symfony es simplemente incluirlo como un asset de la misma forma que trabajaríamos con las librerías <http://www.tinymce.com> o <http://lightbox.com>. Incluimos el archivo y la programación que hacemos para usarla ya es JavaScript y no PHP.

## CAPÍTULO 13: INSTALANDO BUNDLES DE TERCEROS

En reiteradas ocasiones existe la necesidad de implementar librerías de terceros en nuestros proyectos con el objetivo primordial de aprovecharlas, reutilizar código y mejorar nuestros tiempos de entrega; como ya saben en Symfony 2 todo se distribuye en forma de “Bundles” y las librerías de terceros no son la excepción, además en [symfony2bundles.org](http://symfony2bundles.org) (actualmente <http://knpbundles.com/>) podrás encontrar miles de bundles que podrías necesitar.

En este capítulo nos concentraremos en la instalación de uno de los Bundles más atractivos para incluir en nuestros proyectos, se trata del [StofDoctrineExtensionsBundle](#)<sup>1</sup> por Christophe Coevoet el cual hace una implementación del complemento [Doctrine2 behavioral extensions](#)<sup>2</sup> creado por Gediminas Morkevicius, cuyo propósito es proveer de los aclamados Comportamientos (behaviors) de Doctrine; en general explicaremos la configuración de 3 comportamientos ampliamente utilizados, como lo son Timestampable, Sluggable y Loggable, reiteramos que el objetivo del capítulo es la instalación de Bundles de Terceros en Symfony2 y no el de profundizar en todos los comportamientos que provee el StofDoctrineExtensionsBundle.

### ► PASO 1 – INSTALANDO EL BUNDLE

En Symfony 2 existen varias formas para la instalación de los bundles de terceros, entre ellas (y la más práctica) es la instalación por medio de submodule en GIT, si no sabes qué es GIT te recomiendo visitar este enlace (<http://progit.org/book/es/>) y tratar de instalarlo en tu sistema.

Comenzamos instalando la librería principal de Doctrine Behavioral Extensions, para ello accedemos a nuestra consola, nos ubicamos en la raíz del proyecto de symfony y ejecutamos:

```
~$ git submodule add git://github.com/l3pp4rd/DoctrineExtensions.git  
vendor/gedmo-doctrine-extensions
```

Si tienes problemas puedes descargar manualmente el paquete (<https://github.com/l3pp4rd/DoctrineExtensions>), sólo debes descomprimir su contenido y copiarlo al directorio /ruta\_hacia\_proyecto/vendor/gedmo-doctrine-extensions.

<sup>1</sup> <https://github.com/stof/StofDoctrineExtensionsBundle>

<sup>2</sup> <https://github.com/l3pp4rd/DoctrineExtensions>

## Ahora si añadimos el StofDoctrineExtensionsBundle:

```
~$ git submodule add git://github.com/stof/StofDoctrineExtensionsBundle.
git vendor/bundles/Stof/DoctrineExtensionsBundle
```

Si tienes problemas puedes descargar manualmente el paquete (<https://github.com/stof/StofDoctrineExtensionsBundle>), sólo debes descomprimir su contenido y copiarlo al directorio /ruta\_hacia\_proyecto/vendor/bundles/Stof/DoctrineExtensionsBundle.

Registramos los Namespaces en nuestro app/autoload.php:

```
<?php

use Symfony\Component\ClassLoader\UniversalClassLoader;
use Doctrine\Common\Annotations\AnnotationRegistry;

$loader = new UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony'      => array(__DIR__.'/../vendor/symfony/src', __
DIR__.'/../vendor/bundles'),
    'Sensio'        => __DIR__.'/../vendor/bundles',
    'JMS'           => __DIR__.'/../vendor/bundles',
    'Doctrine\\Common' => __DIR__.'/../vendor/doctrine-common/lib',
    'Doctrine\\DBAL'   => __DIR__.'/../vendor/doctrine-dbal/lib',
    'Doctrine'       => __DIR__.'/../vendor/doctrine/lib',
    'Monolog'        => __DIR__.'/../vendor/monolog/src',
    'Assetic'         => __DIR__.'/../vendor/assetic/src',
    'Metadata'        => __DIR__.'/../vendor/metadata/src',
    // Aquí registramos:
    'Stof'           => __DIR__.'/../vendor/bundles',
    'Gedmo'          => __DIR__.'/../vendor/gedmo-doctrine-extensions/lib',
));
// ... resto del archivo
```

Añadimos el Bundle a nuestro app/AppKernel.php:

```
<?php

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

```
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new Symfony\Bundle\MonologBundle\MonologBundle(),
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
            new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
            new Symfony\Bundle\AsseticBundle\AsseticBundle(),
            new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
            new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
            new MDW\BlogBundle\MDWBlogBundle(),
            new MDW\DemoBundle\MDWDemoBundle(),
            // Aquí Añadimos:
            new Stof\DoctrineExtensionsBundle\
StofDoctrineExtensionsBundle(),
        );
        // ... resto del archivo
    }
}
```

Básicamente es todo lo que se realiza para incluir un Bundle.

## ► PASO 2 - CONFIGURANDO EL BUNDLE: STOFDOCTRINEEXTENSIONSBundle

En nuestro caso el `StofDoctrineExtensiosBundle` para funcionar requiere agregar configuración adicional al archivo `app/config/config.yml` de la aplicación (la mayoría de los bundles pueden detallar tales configuraciones en su documentación), para ello agregamos estos segmentos:

En la sección Doctrine Configuration, añadimos al final el mapping para `StofDoctrineExtensionsBundle`:

```
# Doctrine Configuration
doctrine:
    dbal:
        driver:    %database_driver%
        host:      %database_host%
        port:      %database_port%
        dbname:   %database_name%
        user:      %database_user%
        password: %database_password%
        charset:  UTF8
    orm:
        auto_generate_proxy_classes: %kernel.debug%
        auto_mapping: true
        # Añadimos el Mapping para StofDoctrineExtensiosBundle: -----
        ---
        mappings:
            StofDoctrineExtensionsBundle: ~
    # ... resto del archivo
```

Luego añadimos **al final del mismo archivo** la siguiente configuración:

```
# Añadimos las configuraciones específicas para el
StofDoctrineExtensiosBundle
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            sluggable: true
            timestampable: true
            loggable: true
            #demás behaviors para activar
```

Donde **default**: representa la configuración para todos los entornos, eso quiere decir que puedes añadir una configuración específica para cada entorno.

## ► PASO 3 – UTILIZANDO LOS COMPORTAMIENTOS (BEHAVIORS) EN LOS MODELOS

En este tutorial nos concentraremos en los comportamientos sluggable, timestampable y loggable, para hacer las cosas más fáciles se recomienda añadir el siguiente Namespace a cada una de nuestras entidades en donde queramos añadir los comportamientos:

```
// añadimos luego del Namespace de la Entidad:  
use Gedmo\Mapping\Annotation as Gedmo;
```

Algunos Behaviors como Loggable disponen de Entidades propias que requieren crearse en base de datos, para garantizar ello solo debemos ejecutar en consola el comando siguiente, de este modo Doctrine creará las tablas necesarias para almacenar los datos generados por los comportamientos que lo requieran:

```
~$ php app/console doctrine:schema:update --force
```

**Sluggable:** Permite que Doctrine cree automáticamente el “slug” o la típica cadena optimizada para buscadores utilizada comúnmente al indexar artículos de un blog. Para definir el slug debemos tener un campo destino que es donde se almacenará (el Slug) y uno o más campos origen (los Sluggable) de los cuales se construirá el slug y es tan simple como agregar los siguientes **Metadatos** a nuestros campos del modelo:

```
// ... dentro de una Entidad  
// Campo origen:  
/**  
 * @var string $title  
 *  
 * @ORM\Column(name="title", type="string", length=255)  
 * @Gedmo\Sluggable()  
 */  
private $title;  
  
// ... otras variables  
  
// Campo Destino:  
/**  
 * @var string $slug  
 *
```

```

 * @ORM\Column(name="slug", type="string", length=255)
 * @Gedmo\Slug(style="camel", separator="_", updatable=false, unique=true)
 */
private $slug;
// ... dentro de una Entidad

```

Note que el campo desde donde se creará el slug (`$title`) tiene el Metadato `@Gedmo\Sluggable()`, de hecho puede definir más de uno. En cambio el campo de destino (`$slug`) tiene el Metadato `@Gedmo\Slug(...)` y por convención debe ser uno solo, los argumentos style, separator, updatable y unique son opcionales y se detallan en la documentación propia del autor, en este ejemplo se tiene una forma básica de configuración.

Cada vez que se cree un registro de la entidad, Doctrine automáticamente generará el slug y lo aplicará al campo destino, en el caso de modificaciones depende del valor del argumento `updatable`.

**Timestampable:** permite que Doctrine gestione la actualización del Timestamp en campos específicos al realizar operaciones de inserción y/o actualización. Para definir un campo con `Timestampable` solo debemos añadir el Metadato `Gedmo\Timestampable(on="action")`, donde action puede ser `created` o `updated` respectivamente:

```

// ... dentro de una Entidad
// Campo created:
/**
 * @var date $created
 *
 * @ORM\Column(name="created", type="date")
 * @Gedmo\Timestampable(on="create")
*/
private $created;
// Campo updated:
/**
 * @var datetime $updated
 *
 * @ORM\Column(name="updated", type="datetime")
 * @Gedmo\Timestampable(on="update")
*/
private $updated;

```

```
// ... dentro de una Entidad
```

Doctrine automáticamente aplicará un nuevo Date al campo definido on="create" al crear un nuevo registro de la entidad y actualizará el Timestamp del campo definido on="update" al actualizar el registro de la entidad.

**Loggable:** permite que Doctrine lleve un control de Versiones sobre los campos indicados, permitiendo consultar las versiones y revertir hacia una versión anterior.

Para crear campos con log (control de versión) solo debemos añadir a cada campo el Metadato @Gedmo\Versioned(), además de añadir el Metadato @Gedmo\Loggable() a la Entidad correspondiente:

```
// ... encabezados del archivo
// definimos el Metadato @Gedmo\Loggable() a la Entidad:
/***
 * MDW\BlogBundle\Entity\Articles
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="MDW\BlogBundle\Entity\ArticlesRepository")
 * @Gedmo\Loggable()
 */
class Articles
{
    // ... dentro de una Entidad
    // Campo $content será Versionable:
    /**
     * @var text $content
     *
     * @ORM\Column(name="content", type="text")
     * @Gedmo\Versioned()
     */
    private $content;
    // ... demás contenido de la entidad
```

Doctrine automáticamente supervisará los updates hacia la entidad y llevará un control de versiones en la Entidad (Stof\DoctrineExtensionsBundle\Entity\LogEntry) y gracias al Repositorio de

dicha entidad (`\Gedmo\Loggable\Entity\Repository\LogEntryRepository`) podremos consultar las Versiones e incluso Revertir los cambios (función `$logRepositoryInstance->revert($Entity, $version);`), aquí apreciamos un ejemplo de un controlador que lista los cambios:

```
// Ejemplo dentro de un Controller:
public function updateArticleAction($id) {
    $em = $this->getDoctrine()->getEntityManager();

    $article = $em->getRepository('MDWBlogBundle:Articles')-
>findOneBy(array('id' => $id));

    $article->setContent('editado');
    $em->persist($article);
    $em->flush();

    $content = '';
    // ver cambios
    $log = $em->getRepository('Stof\DoctrineExtensionsBundle\Entity\'
LogEntry');
    /* @var $log \Gedmo\Loggable\Entity\Repository\LogEntryRepository */

    $query_changues = $log->getLogEntriesQuery($article); //use
$log->getLogEntries() para un result directo
    $changues = $query_changues->getResult();
    /* @var $version Stof\DoctrineExtensionsBundle\Entity\LogEntry */
    foreach ($changues as $version) {
        $fields = $version->getData();
        $content.= ' fecha: ' .
            $version->getLoggedAt()->format('d/m/Y H:i:s') .
            ' accion: "' . $version->getAction() . '"';
            ' usuario: "' . $version->getUsername() . '"';
            ' objeto: "' . $version->getObjectClass() . '"';
            ' id: "' . $version->getObjectId() . '"';
            ' Version: "' . $version->getVersion() . '"';
            ' datos:<br />';
        foreach ($fields as $field => $value) {
            $content.= '-- ' . $field . ': ' . $value . '<br />';
        }
    }
}
```

```
    }  
}  
// generamos una salida básica  
$r = new \Symfony\Component\HttpFoundation\Response();  
$r->setContent($content);  
return $r;  
}
```

De esta forma podemos aprovecharnos de los comportamientos de Doctrine, reutilizar código y automatizar tareas en nuestros modelos.

## RESUMEN DE CAPÍTULO

Como pudimos apreciar con Symfony2 disponemos de una amplia variedad de bundles de terceros a incluir para extender las capacidades de nuestras aplicaciones, aprovechar y reutilizar código mejorando considerablemente el tiempo en el desarrollo de nuestros proyectos; aprendimos que existen diversas formas de incluir nuestros bundles y que en dado caso podemos hacer instalaciones a mano, también de lo importante que es seguir la documentación de cada bundle para agregarlo en el Auto-load o Kernel según corresponda y aplicar las configuraciones requeridas por el mismo.

## CAPÍTULO 14: SEGURIDAD DE ACCESO

Uno de los aspectos más importantes en el desarrollo de cualquier aplicación es la Seguridad de acceso, para ello Symfony 2 dispone de una moderna librería que se encarga de las validaciones de acceso y seguridad.

En este capítulo nos dispondremos a crear un ejemplo básico de seguridad que nos permita hacer el “login” de los usuarios y a su vez bloquear el acceso a determinados usuarios según su rol, para ello tendremos que adentrarnos en como funciona la librería de seguridad de Symfony2, luego crearemos las estructuras necesarias para definir un mini-backend donde crearemos unos CRUD’s para usuarios y roles utilizando Doctrine como proveedor de usuarios.

## AUTENTICACIÓN (FIREWALLS) VS AUTORIZACIÓN (ACCESS\_CONTROL)

Representan los 2 conceptos más fundamentales de seguridad en Symfony, el primero se encarga de verificar si el usuario en cuestión está Autenticado (logeado) y se le conoce como “Firewall”, el segundo verifica si el usuario tiene los permisos o “roles” necesarios y se le conoce como “access\_control”.

El primer paso es verificar si el usuario está o no autenticado, en tal caso lo deja pasar y el segundo paso es verificar si el usuario tiene el rol necesario para dicha acción, para comprenderlo mejor veamos un ejemplo básico del archivo de configuración:

```
# proyecto/app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:      ^
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"
    access_control:
```

```

- { path: ^/admin, roles: ROLE_ADMIN }

providers:
    in_memory:
        users:
            usuario: { password: user, roles: 'ROLE_USER' }
            admin: { password: kitten, roles: 'ROLE_ADMIN' }

encoders:
    Symfony\Component\Security\Core\User\User: plaintext

```

Vemos como primer elemento definido los conjuntos de firewall y en él una “secured\_area” donde:

- ⌚ **pattern:** es una expresión regular para hacer empatar la URL, toda ruta que empate con ello obligará al mecanismo de firewall que verifique si el usuario está autenticado, si no lo está procederá a re-dirigirlo al formulario de autenticación (en el caso anterior mostrar el diálogo nativo de autenticación HTTP del navegador).
- ⌚ **anonymous:** ~ :indica que permite usuarios anónimos, no se debe aplicar en caso de backends.
- ⌚ **http\_basic:** indica que utilice la autenticación HTTP.

Por su parte el Mecanismo de Autorización “access\_control” actúa de forma diferente, porque aunque dependa de que el usuario esté autenticado verifica si el mismo dispone de los permisos (roles) necesarios para determinada operación:

- ⌚ **- { path: ^/admin, roles: ROLE\_ADMIN }:** indica una regla básica para autorización, donde en toda ruta que coincida con /admin al principio el usuario debe de tener dicho rol “ROLE\_ADMIN” indicado.

---

**Nota:** puedes añadir tantos access\_control como necesites.

Providers simplemente define el proveedor de usuario, que en este caso es en memoria y Encoders define el codificador de la contraseña, el cual debe ser de algún tipo de HASH como SHA512, en el ejemplo se usa texto plano.

# CONFIGURACIONES DEL CONTROL DE ACCESO

El control de acceso no sólo se limita a controlar que el usuario cumpla con un rol determinado para un patrón de ruta determinada, permite cierta flexibilidad con el que podrás adaptarte a las necesidades de seguridad de tu aplicación.

## PROTEGIENDO POR IP

Tan simple como añadir el parámetro ip con la misma puedes obligar a que una ruta solo se pueda acceder desde dicha ip:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip:
127.0.0.1 }
```

## PROTEGIENDO POR CANAL

Si dispones de un certificado SSL puedes obligar a que la ruta solo esté disponible desde https, especificando requires\_channel:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY,
requires_channel: https}
```

## PROTEGIENDO UN CONTROLADOR

En ocasiones necesitamos que el controlador se encargue del control de acceso, de forma que podemos flexibilizarlo según nuestro modelo de negocios, para ello podremos acceder al contexto de seguridad desde nuestros controladores:

```
//dentro de un controlador
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...
```

```

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }
    // ...
}

```

También puede optar por instalar y utilizar el Bundle opcional `JMSSecurityExtraBundle`, el cual puede asegurar su controlador usando anotaciones:

```

//dentro de un controlador
use JMS\SecurityExtraBundle\Annotation\Secure;
/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}

```

Para más información, consulte la documentación de `JMSSecurityExtraBundle`. Si estás usando la distribución estándar de Symfony, este paquete está disponible de forma predeterminada. Si no es así, lo puedes descargar e instalar.

## CONTROLANDO EL ACCESO DESDE PLANTILLA

Incluso puedes verificar si el usuario tiene acceso desde la misma plantilla, útil para ocultar segmentos a roles específicos:

### DESDE TWIG:

```

{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}

```

## DESDE PHP:

```
<?php if ($view['security']->isGranted('ROLE_ADMIN')): ?>
    <a href="...">Delete</a>
<?php endif; ?>
```

## RECUPERANDO DEL OBJETO USUARIO

Desde un controlador, puedes acceder fácilmente a la instancia del usuario actual utilizando el Mecanismo de inyección de dependencias:

```
//dentro de un controlador
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

## TUTORIAL: MINI-BACKEND DE USUARIOS CON DOCTRINE

Realmente el ejemplo anterior es demasiado básico como para llevarlo a una aplicación real y una de las opciones más tentadoras es utilizar Doctrine como proveedor de los Usuarios, con el cual podamos crear Roles y Usuarios desde CRUD's elaborados por el mismo framework y crear nuestro propio esquema de seguridad, debo resaltar que existen muchos Bundles prefabricados como el FOSUserBundle que facilitan enormemente ésta tarea, pero si quieres profundizar puedes seguir el siguiente tutorial para conocer a fondo como se hace desde 0 con Doctrine .

### ► PASO 1: CREA LAS ENTIDADES BÁSICAS

---

Antes de empezar debemos definir las entidades básicas para ser utilizadas como proveedor de usuarios y roles en Sf2, dichas entidades User y Role deben implementar las interfaces Symfony\Component\Security\Core\User\UserInterface y Symfony\Component\Security\Core\Role\RoleInterface respectivamente, así que añade estas 2 entidades a tu directorio “proyecto/src/MDW/BlogBundle/Entity”:

## USER.PHP:

```
<?php
// proyecto/src/MDW/BlogBundle/Entity/User.php
namespace MDW\BlogBundle\Entity;
```

```
use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="admin_user")
 */
class User implements UserInterface
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length="255")
     */
    protected $username;

    /**
     * @ORM\Column(name="password", type="string", length="255")
     */
    protected $password;

    /**
     * @ORM\Column(name="salt", type="string", length="255")
     */
    protected $salt;

    /**
     * se utilizó user_roles para no hacer conflicto al aplicar ->toArray
     * en getRoles()
     * @ORM\ManyToMany(targetEntity="Role")
     * @ORM\JoinTable(name="user_role",
     *      joinColumns={@ORM\JoinColumn(name="user_id",
     * referencedColumnName="id")},
     *      inverseJoinColumns={@ORM\JoinColumn(name="role_id",
     *
```

```
referencedColumnName="id")}

    *
    */

protected $user_roles;
public function __construct()
{
    $this->user_roles = new \Doctrine\Common\Collections\ArrayCollection();
}

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set username
 *
 * @param string $username
 */
public function setUsername($username)
{
    $this->username = $username;
}

/**
 * Get username
 *
 * @return string
 */
public function getUsername()
{
    return $this->username;
}
```

```
/**
 * Set password
 *
 * @param string $password
 */
public function setPassword($password)
{
    $this->password = $password;
}

/**
 * Get password
 *
 * @return string
 */
public function getPassword()
{
    return $this->password;
}

/**
 * Set salt
 *
 * @param string $salt
 */
public function setSalt($salt)
{
    $this->salt = $salt;
}

/**
 * Get salt
 *
 * @return string
 */
public function getSalt()
{
    return $this->salt;
}
```

```
/**
 * Add user_roles
 *
 * @param Maycol\BlogBundle\Entity\Role $userRoles
 */
public function addRole(\Maycol\BlogBundle\Entity\Role $userRoles)
{
    $this->user_roles[] = $userRoles;
}

public function setUserRoles($roles) {
    $this->user_roles = $roles;
}

/**
 * Get user_roles
 *
 * @return Doctrine\Common\Collections\Collection
 */
public function getUserRoles()
{
    return $this->user_roles;
}

/**
 * Get roles
 *
 * @return Doctrine\Common\Collections\Collection
 */
public function getRoles()
{
    return $this->user_roles->toArray(); //IMPORTANTE: el mecanismo de
seguridad de Sf2 requiere ésto como un array
}

/**
 * Compares this user to another to determine if they are the same.
 *
 * @param UserInterface $user The user
 * @return boolean True if equal, false otherwise.

```

```

        */
    public function equals(UserInterface $user) {
        return md5($this->getUsername()) == md5($user->getUsername());
    }
    /**
     * Erases the user credentials.
     */
    public function eraseCredentials() {
    }
}

```

## ROLE.PHP

```

<?php
// proyecto/src/MDW/BlogBundle/Entity/Role.php
namespace MDW\BlogBundle\Entity;
use Symfony\Component\Security\Core\Role\RoleInterface;
use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="admin_roles")
 */
class Role implements RoleInterface
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;
    /**
     * @ORM\Column(name="nombre", type="string", length="255")
     */
    protected $name;
    /**
     * Get id
     *

```

```
* @return integer
*/
public function getId()
{
    return $this->id;
}

/**
 * Set name
 *
 * @param string $name
 */
public function setName($name)
{
    $this->name = $name;
}

/**
 * Get name
 *
 * @return string
 */
public function getName()
{
    return $this->name;
}

public function getRole() {
    return $this->getName();
}

public function __toString() {
    return $this->getRole();
}
}
```

Una vez creadas nuestras entidades, accedemos a la consola de symfony y generamos las tablas en Base de Datos:

```
~$ app/console doctrine:schema:update --force
```

## ► PASO 2: GENERANDO LOS CRUD'S

Una vez creadas las entidades en DB, procedemos a crear los CRUD's desde la consola de symfony:

```
~$ app/console doctrine:generate:crud
```

Seguimos los pasos colocando MDWBlogBundle:Role, luego nos solicita si deseamos crear las opciones de escritura, le decimos y, formato del CRUD: annotation, y finalmente en el Routes prefix colocamos /admin/role, este paso es importante porque a la ruta le asignamos el prefijo /admin para que nos permita empatar luego con el access\_control, confirmamos y aparecerá el mensaje "You can now start using the generated code!"

Procedemos a aplicar lo mismo pero en este caso con [MDWBlogBundle:User](#) y en Routes prefix colocamos /admin/user.

Ahora añadiremos las rutas a nuestro archivo de rutas (`proyecto/src/MDW/Bundle/Resources/Config/routing.yml`), porque al crearlas como Anotaciones las mismas no se añaden automáticamente:

```
#proyecto/src/MDW/Bundle/Resources/Config/routing.yml
# final del archivo:
MDWAnnotations:
    resource: "@MDWBlogBundle\Controller/"
    prefix:   /
    type:     annotation
```

De ésta forma añadiremos todas las rutas definidas por anotaciones del directorio Controller, ésta técnica forma parte del SensioFrameworkExtraBundle y nos permite definir las rutas directamente en nuestros controladores.

Ya con esto podemos acceder a nuestros crud's desde `localhost/proyecto/web/app_dev.php/admin/user`, pero aún debemos modificar ciertos aspectos en el controlador User para codificar el hash de contraseña.

Primero añadiremos la siguiente función en el controlador de usuarios:

```
// proyecto/src/MDW/Bundle/Controller/UserController.php
// añadimos esta función
private function setSecurePassword(&$entity) {
```

```

$entity->setSalt(md5(time()));
$encoder = new \Symfony\Component\Security\Core\Encoder\MessageDigestPasswordEncoder('sha512', true, 10);
$password = $encoder->encodePassword($entity->getPassword(), $entity->getSalt());
$entity->setPassword($password);
}

```

Luego modificamos las funciones de las acciones correspondientes a create y update, añadiendo la llamada a la función anterior para establecer el hash de la contraseña con el algoritmo SHA512:

```

// proyecto/src/MDW/BlogBundle/Controller/UserController.php
//funcion createAction:
public function createAction()
{
    $entity = new User();
    $request = $this->getRequest();
    $form = $this->createForm(new UserType(), $entity);
    $form->bindRequest($request);
    if ($form->isValid()) {
        //establecemos la contraseña: -----
        $this->setSecurePassword($entity);
        $em = $this->getDoctrine()->getEntityManager();
        $em->persist($entity);
        $em->flush();
        return $this->redirect($this->generateUrl('admin_user_show',
array('id' => $entity->getId())));
    }
    return array(
        'entity' => $entity,
        'form'   => $form->createView()
    );
}
//...
//funcion updateAction:
public function updateAction($id)
{

```

```

$em = $this->getDoctrine()->getEntityManager();
$entity = $em->getRepository('MDWBlogBundle:User')->find($id);
if (!$entity) {
    throw $this->createNotFoundException('Unable to find User
entity.');
}
$editForm = $this->createForm(new UserType(), $entity);
$deleteForm = $this->createDeleteForm($id);
$request = $this->getRequest();
//obtiene la contraseña actual -----
$current_pass = $entity->getPassword();
$editForm->bindRequest($request);
if ($editForm->isValid()) {
    //evalua si la contraseña fue modificada: -----
    --
    if ($current_pass != $entity->getPassword()) {
        $this->setSecurePassword($entity);
    }
    $em->persist($entity);
    $em->flush();
    return $this->redirect($this->generateUrl('admin_user_edit',
array('id' => $id)));
}
return array(
    'entity'      => $entity,
    'edit_form'   => $editForm->createView(),
    'delete_form' => $deleteForm->createView(),
);
}

```

Por último sólo nos queda eliminar del formulario (src/MDW/BlogBundle/Form/UserType.php) el campo salt el cual no debe ser modificado por el usuario:

```

<?php
// proyecto/src/MDW/BlogBundle/Form/UserType.php
namespace MDW\BlogBundle\Form;
use Symfony\Component\Form\AbstractType;

```

```

use Symfony\Component\Form\FormBuilder;
class UserType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('username')
            ->add('password')
            //->add('salt') //No necesitamos que salt sea mostrado
            ->add('user_roles')
        ;
    }
    public function getName()
    {
        return 'mdw_blogbundle_usertype';
    }
}

```

Ahora puedes proceder a registrar usuarios y roles, **es muy importante que al menos crees los roles “ROLE\_ADMIN” y “ROLE\_USER” y dos usuarios** (uno con un rol diferente) antes de que procedas en aplicar el esquema de seguridad, de lo contrario no tendrás usuario con queloguearte .

## ► PASO 3: CREANDO EL ESQUEMA DE SEGURIDAD

---

Ahora procedemos a sobreescibir nuestro esquema de seguridad (proyecto/app/config/security.yml), recomiendo que antes de hacerlo guardes una copia del security.yml.

```

# proyecto/app/config/security.yml
security:
    encoders:
        MDW\BlogBundle\Entity\User:
            algorithm: sha512
            encode-as-base64: true
            iterations: 10
    providers:
        user_db:
            entity: { class: MDW\BlogBundle\Entity\User, property:

```

```

username }

firewalls:
    dev:
        pattern:  ^/(_(profiler|wdt)|css|images|js)/
        security: false
    login:
        pattern:  ^/admin/login$
        security: false
    secured_area:
        pattern:  ^/admin/
#        http_basic:
#            realm: "Introduzca Usuario y Contraseña"
        form_login:
            login_path: /admin/login
            check_path: /admin/login_check
        logout:
            path: /admin/logout
            target: /
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

```

Como vemos en “**encoders**” se ha definido un codificador específico para la entidad User, utilizando el algoritmo SHA512, además codificándolo en Base64 con 10 iteraciones, tal cual se apreció en la función `setSecurePassword` del controlador.

En “**providers**” se estableció nuestra entidad User de Doctrine, especificando el campo correspondiente al `username`, es cual es el mismo `username` en nuestra entidad.

En “**firewalls**” se ha añadido la nueva regla (o firewall) **login** desde la cual se aplica el parametro `security: false` lo que permite acceder a la misma sin autenticarse, de lo contrario el formulario de login nunca lo podremos visualizar.

Además en “**secured\_area**” se ha eliminado `anonymous`, se ha establecido “**form\_login**” donde definimos la ruta para el login del sistema y se definió una ruta personalizada para el “**log\_out**”, donde en “**target**” podemos definir el path hacia donde redirigir cuando los usuarios cierren sesión.

Para culminar sólo necesitamos crear el controlador y vista para nuestro login, por lo que debes de crear el archivo `SecurityController.php` en el directorio (`proyecto/src/MDW/BlogBundle/Controller`):

## SECURITYCONTROLLER.PHP

```
<?php  
// proyecto/src/MDW/BlogBundle/Controller/SecurityController.php  
namespace MDW\BlogBundle\Controller;  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;  
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;  
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;  
use Symfony\Component\Security\Core\SecurityContext;  
/**  
 * Security controller.  
 *  
 * @Route("/admin")  
 */  
class SecurityController extends Controller  
{  
    /**  
     * Definimos las rutas para el login:  
     * @Route("/login", name="login")  
     * @Route("/login_check", name="login_check")  
     */  
    public function loginAction()  
    {  
        $request = $this->getRequest();  
        $session = $request->getSession();  
        // obtiene el error de inicio de sesión si lo hay  
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {  
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);  
        } else {  
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);  
        }  
    }  
}
```

```

    }

    return $this->render('MDWBlogBundle:Security:login.html.twig',
array(
    // el último nombre de usuario ingresado por el usuario
    'last_username' => $session->get(SecurityContext::LAST_
USERNAME),
    'error'           => $error,
));
}

?>

```

Ahora crea el directorio “Security” dentro de (proyecto/src/MDW/BlogBundle/Resources/views) y procede a crear el archivo de vista:

## LOGIN.HTML.TWIG

```

<# proyecto/src/MDW/BlogBundle/Resources/views/Security/login.html.twig #>
{% if error %}

    <div>{{ error.message }}</div>

{% endif %}
<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_
username }} " />
    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />
    <input type="submit" name="login" />
</form>

```

Y con ello ya puedes intentar acceder a localhost/proyecto/web/admin/user y probar el sistema de seguridad de Symfony2 (vaciar la caché en el caso de entrar al entorno de producción), **si creaste previamente 2 usuarios**, intenta acceder con el usuario que no tiene el rol “ROLE\_ADMIN” y verás como te niega el acceso, en cambio si pruebas con un usuario con dicho rol, puedes entrar perfectamente.

## RESUMEN DEL CAPÍTULO

En esta ocasión apreciamos el complejo sistema de seguridad de Symfony2, en donde un **firewall** verifica si el usuario está o no logueado y un **access control** vigila que dicho usuario no pueda acceder a contenido del cual no se le ha dado acceso, también conocimos que se pueden definir providers diferentes para contener a nuestros usuarios y **encoders** para personalizar el HASH de la contraseña.

Además interactuamos con dicho sistema a través de un “rápido” tutorial que nos permitió resolver las inquietudes más directas en cuanto a creación de un básico RBAC (Role-based Access Control), reitero que no es la única forma de hacerlo y que existen muchos Bundles Prefabricados como el FOSUserBundle que nos facilita enormemente ésta tarea, pero si no se conoce debidamente la base puede resultar una verdadera caja negra el usar un Bundle sin el previo conocimiento de como Symfony2 implementa tales mecanismos.

## MÁS GUÍAS DE MAESTROS DEL WEB



### ADICTOS A LA COMUNICACIÓN

Utiliza las herramientas sociales en Internet para crear proyectos de comunicación independientes.

[Visita Adictos a la comunicación](http://mdw.li/guiacomunica)

<http://mdw.li/guiacomunica>



### GUÍA STARTUP

Aprende las oportunidades, retos y estrategias que toda persona debe conocer al momento de emprender.

[Visita la Guía Startup](http://mdw.li/gkTDom)

<http://mdw.li/gkTDom>

## MÁS GUÍAS DE MAESTROS DEL WEB



**LOS MAESTROS DEL WEB**  
Personas y proyectos que nos inspiran

**CURSO  
ANDROID**  
Desarrollo de  
aplicaciones móviles

CATEGORÍA: PROGRAMACIÓN  
NIVEL: INTERMEDIO

### LOS MAESTROS DEL WEB

Una serie de perfiles de personas y proyectos que nos inspiran a permanecer en el medio, aprender y seguir evolucionando.

[Visita Los Maestros del Web](#)

<http://j.mp/spAncK>

### CURSO ANDROID

Actualiza tus conocimientos con el curso sobre Android para el desarrollo de aplicaciones móviles.

[Visita el Curso Android](#)

<http://mdw.li/lmlydX>

## MÁS GUÍAS DE MAESTROS DEL WEB



### GUÍA ASP.NET

ASP.NET es un modelo de desarrollo Web unificado creado por Microsoft para el desarrollo de sitios y aplicaciones web dinámicas con un mínimo de código. ASP.NET

[Visita la Guía ASP.NET](http://mdw.li/guiaaspnet)  
http://mdw.li/guiaaspnet



### GUÍA ZEND

Zend Framework es un framework de código abierto para desarrollar aplicaciones y servicios web con PHP 5.

[Visita la Guía Zend](http://mdw.li/guiazend)  
http://mdw.li/guiazend