SYMFONY 2

# INTRODUCCIÓN A SYMFONY 2

SYMFONY 2 (FRAMEWORK PHP ORIENTADO A OBJETOS)

* Uso para version php 5.3.3 o mas
* Creado por comunidad de Fabien Potencier
* Symfony = No reinventar la rueda
* Arquitectura MVC(Model, view, controller) – Modelado, Vista y Controlador (Capas)
* Vista : Visualizacion de información, diseño, colores, estilos y estructura visual.
* Modelado: Conexión de DB y manipulación de datos
* Controlador: Procesa y muestra datos obtenidos por el modelado.



Funcionamiento: El cliente envía un REQUEST o petición, la recibe el **controlador** q realiza validaciones necesarias, procesa datos y lógica del negocio asociada a la petición. El **controlador**  envía datos al **modelado,** y se envían a la **vista** para ser mostrados al cliente en un RESPONSE o respuesta.

**Framewrks ORM- Object-relational mapping (Mapeo entre objetos y relaciones)**

* Doctrine = Control de datos
  + Framework encargado de trato con la DB (conexión, SQL, manipulación, transaccio y desconexion)
  + NO TABLAS = OBJETOS
  + Ejemplo
    - agregar una persona a la tabla usuario = agregar objeto persona
    - un país esta relacionado a varias personas = objeto país contiene un colección de objetos persona.
    - Se crean clases que mapean cada relación de la base de datos , se programan los objetos y **DOCTRINE** traduce lo necesario para hablar con la DB
* MOTOR DE PLANTILLAS
  + Twig = Motor de plantillas (Separa HTML con PHP) – Lenguaje YAML, aunque se puede usar XML o PHP
  + Instrucciones de consola (Tasks - Tareas)
    - Permite ejecutar comandos en la terminal para generar lo pedido como ejemplo crear ABMs
  + Los formularios se crean programando una herramienta del framework = Se guardan todos los formularios con sus respectivas validaciones en un solo lugar.
  + Separar completamente el código PHP con el HTML
  + Guardar datos en variables que vamos a mostrar al usuario, y el motor de plantillas se encarga de obtener la plantilla con el HTML mostrando contenido de variables en su respectivo lugar.
    - Diseño = Diseñador
    - Logica = Programador

**LINKS DE DESCARGA**

WEB DEL TUTORIAL DE USO DE SYMFONY EN ESPAÑOL

<http://www.maestrosdelweb.com/editorial/curso-symfony2-introduccion-instalacion/>

WEB OFICIAL

<http://symfony.com/>

Quick Tour: Guia de introduccion

<http://symfony.com/doc/2.0/quick_tour/the_big_picture.html>

THE BOOK: Libro con todas las funciones

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

COOKBOOK: Recopilacion de artículos sobre puntos insteresantes.

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

GLOSARIO

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

DOCTRINE

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

TWIG

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

COMPAÑÍA A LA QUE PERTENESE SYMFONY, DOCTRINE Y TWIG

<http://sensiolabs.com/>

BLOG EN ESPAÑOL

<http://www.symfony.es/>

DESCARGAR VERSIONES DE SYMFONY

<http://symfony.com/download>

PUBLICACION DE PRUEBAS DE TEMPLATES

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

SERVIDOR: WAMPSERVER

<http://www.wampserver.com/en/>

YAML

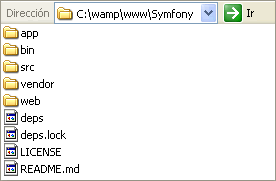
<http://es.wikipedia.org/wiki/YAML>

**NOTA**

Para este manual se usará la última versión disponible de wampserver utilizando Windows como Sistema Operativo, habiéndolo instalado en C:\wamp\. La última versión de Symfony se encuentra en 2.0.9.

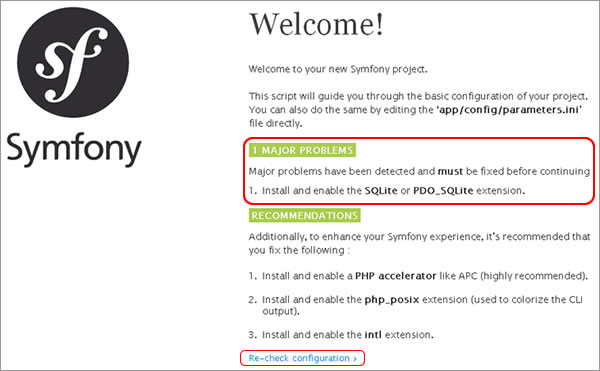
**INSTALANDO EL FRAMEWORK**

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.



Estructura del proyecto

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

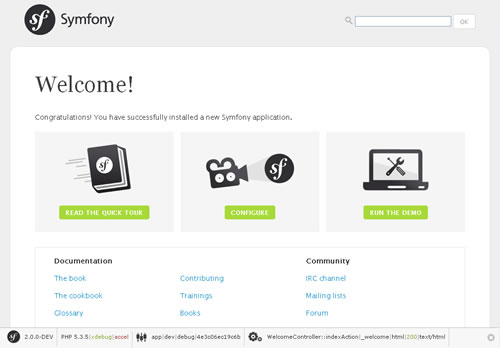


Esta pantalla nos mostrará los requerimientos mínimos y las recomendaciones para usar Symfony en nuestro server.

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 tenemos que 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 podríamos 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.

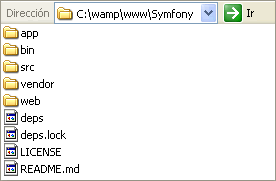


Página de bienvenida

# [EL PROYECTO Y LOS BUNDLES](http://www.maestrosdelweb.com/editorial/curso-symfony2-proyecto-bundles/)

## ESTRUCTURA DE UN PROYECTO SYMFONY

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

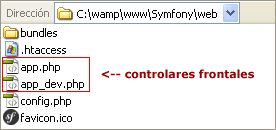


Estructura del proyecto

* **app\:** Aquí se encuentra la configuración correspondiente a todo el proyecto. Debe existir **una** sola aplicación por proyecto. El concepto de tener varias aplicaciones en esta versión es conocido por tener varios Bundles.
* **bin\:** Dentro de esta carpeta tenemos el script vendors.sh que se utiliza para actualizar el framework vía consola.
* **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
* **vendor\:** En esta carpeta se encuentran los archivos del frameworkSymfony y de las demás librerías de terceros como por ejemplo Doctrine, Twig, etc.
* **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 poder 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 podemos acceder a las mismas páginas pero con diferentes configuraciones. Existen dos entornos configurados por defecto en un proyecto Symfony: desarrollo y producción.

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 las utilidades necesarias del framerwork y luego invocan a la página solicitada.

Este es el motivo por el cual en el [capítulo anterior](http://www.maestrosdelweb.com/editorial/curso-symfony2-introduccion-instalacion/), 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\.

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\.

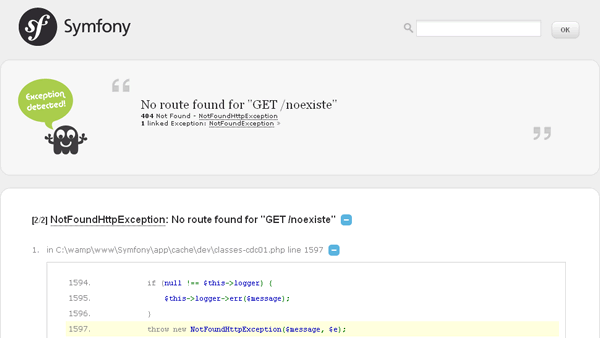
#### SYMFONY PROFILER

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 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.

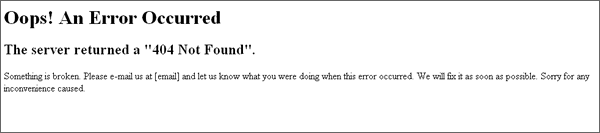
[https://lh3.googleusercontent.com/-sWDtm-eJIiI/TkAbOj8ckFI/AAAAAAAAAFg/ELpRqjM4NjM/s800/sf2_webprofiler.jpg](https://lh3.googleusercontent.com/-sWDtm-eJIiI/TkAbOj8ckFI/AAAAAAAAAFg/ELpRqjM4NjM/s800/sf2_webprofiler.jpg)

#### 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 dira: Error 404 – Página no encontrada.

[](https://lh5.googleusercontent.com/-pCJf-putv1I/TkAj8efvGNI/AAAAAAAAAFk/1KAtJu9DUMs/s800/sf2_404_dev.png)

Mensajes de error en entorno de Desarrollo

[](https://lh5.googleusercontent.com/-6egoHNRcoAk/TkAll2pZUUI/AAAAAAAAAFw/_-XhXsRQTdE/s800/sf2_404_prod.png)

Mensajes de error en entorno de Producción

## ¿QUÉ SON LOS BUNDLES?

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 poder llevar 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 Maestro 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> dándonos la bienvenida. Nosotros crearemos nuestro propio Bundle pero para esto haremos 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> que corresponde a la bienvenida. Ahora para ingresar a esta página debemos escribir <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

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

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

C:\>cd wamp\www\Symfony

Utilidad para pedirle a Symfony que nos diga que versión del framework se está usando.

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:

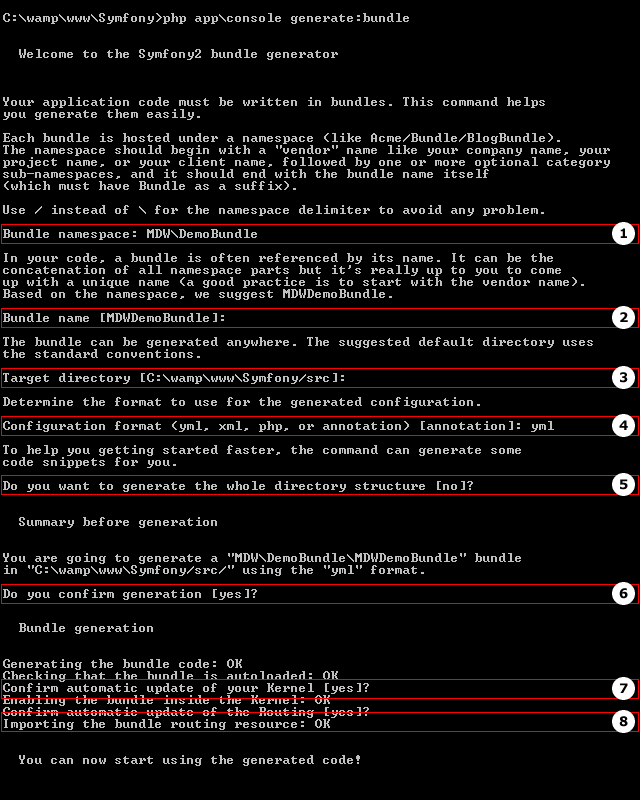
C:\wamp\www\Symfony>php app\console

### 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

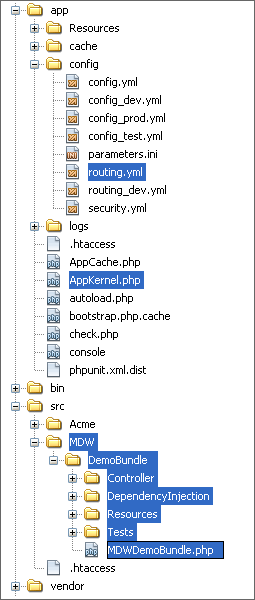
Con este comando se ejecutará un generador que nos hará varias preguntas para crear nuestro Bundle como se muestra a continuación:

[](https://lh3.googleusercontent.com/-4GfbdVRJgQk/Tt5r1IBsv3I/AAAAAAAAAT0/ed01HjC1_nM/s800/generador_bundles_symfony2.png)

Generador de Bundles de Symfony2

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.

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\MDWDemoBundle(),

);

### NOTA

En Symfony 1.x, esto sería similar a habilitar plugins en el archivo ProjectConfiguration.class.php.

También se agregaron las siguientes líneas al archivo app\config\routing.yml

MDWDemoBundle:

    resource: "@MDWDemoBundle/Resources/config/routing.yml"

    prefix:   /

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> 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}.

# [CREANDO PÁGINAS CON SYMFONY 2](http://www.maestrosdelweb.com/editorial/curso-symfony2-creando-paginas/)

Nuestro primer Bundle **MDW\DemoBundle**. Aquí crearemos todos los ejemplos para este manual.

**PASOS PARA CREAR NUESTRAS PÁGINAS**

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

1. **Asignación de una ruta:** 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.

**1. ASIGNACIÓN DE UNA RUTA**

La ruta es 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. Tendremos que importar el archivo routing.yml de nuestro bundle dentro del app\config\routing.yml que es el archivo de rutas genérico para todo el proyecto.

Al crear nuestro Bundle con el script “console”, 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 crearán nuestras rutas.

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.
* Finalmente **/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: IdentificadorDelBundle:Controller:Action

1. **IdentificadorDelBundle:** En este caso nuestro identificador es MDWDemolBundle.
2. **Controller:** El nombre de la clase que contendrá los actions (acciones). Estas clases se encuentran en la carpeta “Controller” de nuestros Bundles.
3. **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**.

**2. CREACIÓN DE UN CONTROLADOR**

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.

**3. 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 }}!

Twig permite separar el código PHP del HTML por lo que por ejemplo en lugar de escribir:

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

podemos simplemente poner:

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 obtendremos de un array.

**EJEMPLO 1**

Lo primero que tenemos que pensar es 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>

**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>. Donde el parámetro “1″ sería el id del artículo que queremos ver.

# [SISTEMA DE ROUTING](http://www.maestrosdelweb.com/editorial/curso-symfony2-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 yweb/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**:

**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.

**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.

**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 podemos ver 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 eliminas las 3 rutas que pertenecen al AcmeDemoBundle, luego, si lo deseas eliminas 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

# [DEFINICIÓN DE RUTAS CON COMODINES](http://www.maestrosdelweb.com/editorial/curso-symfony2-definicion-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 asemejan 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 con 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 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**:

* **\_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.
* **\_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 coincidir con **en** o **es**, además {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 link’s 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>

En el caso de querer una url absoluta solo debemos 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-&amp;amp;gt;get('router')-&amp;amp;gt;generate('blog\_mostrar', array('slug' =&amp;amp;gt; 'mi-articulo'));

// Atajo si extiendes la clase Controller:

$url = $this-&amp;amp;gt;generateUrl('blog\_mostrar', array('slug' =&amp;amp;gt; 'mi-articulo'));

}

}

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

# [TRABAJANDO CON EL CONTROLADOR EN SYMFONY2](http://www.maestrosdelweb.com/editorial/el-controlador/)

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.

#### PERO, ¿QUE 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).

### DEFINIENDO EL CONTROLADOR

Los controladores en Symfony2 se declaran como archivos con el sufijo **Controller**“MicontroladorController.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 referencia 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.

### 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, por ahora 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](http://www.maestrosdelweb.com/editorial/curso-symfony2-definicion-de-rutas-con-comodines/):

// 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 ve en el patrón, Symfony busca la correspondiente plantilla en el directorio MyBundle/Resoruces/views, según su propia estructura interna detallada capítulo de la Vista. Como indicamos en el capítulo 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 puedes apreciar, los atajos de la clase base Controller son de mucha ayuda, y si en dado caso necesitas 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 obtenemos 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 procesado un POST, o dependiendo de la lógica que apliquemos en el Controlador, si en dado caso tengamos 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;// suponiendo 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.

# [MANEJANDO LAS VISTAS CON TWIG EN SYMFONY2](http://www.maestrosdelweb.com/editorial/curso-symfony2-la-vista-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](http://www.maestrosdelweb.com/editorial/phpintro/).

#### ¿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.

### LO BÁSICO DE TWIG

{# comentario #}

{{ mostrar\_algo }}

{% hacer algo %}

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:

1. En nuestros **Bundles**: src/Vendor/MyBundle/Resources/views
2. 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 laextensión 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 =

1. src/MDW/DemoBundle/Resources/**views/Blog**/show.html.twig
2. app/Resources/MDWDemoBundle/**views/Blog**/show.html.twig

MDWDemoBundle**::**show.html.twig =

1. src/MDW/DemoBundle/Resources/**views**/show.html.twig
2. 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 indicamos 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 puedes notar 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:

1. **{% 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.
2. **{% 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 nivelespropuesto 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 artíclo solicitado es: {{articulo}}

{% endblock %}

De está manera al renderizar la plantilla **show.html.twig** ésta a su vez 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 se nos pueden presentar es la de fragmentar el código común y repetitivo en nuestras plantillas (conocido como partials), 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 clausula **with**.

### REUTILIZANDO CONTROLADORES

En algunos casos en nuestras plantillas incluidas necesitamos acceder a datos del modelo que de otra forma sólo seria posible si desde el controlador añadimos la consulta hacia el modelo (por ejemplo un barner 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**(conocido en Symfony1 como los components), 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, además de que dicho controlador **No requiere** que le definas una ruta.

### 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:

<img src="{{ asset('images/logo.gif') }}" alt="mi logo!" />

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 (FORMATEANDO 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 ocasiones 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 puede ser 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. -->

<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 puedes notar 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 FINAL

Se puedes usar PHP como motor de plantillas, pero si elijes quedarte con twig no olvides revisar su documentación (<http://twig.sensiolabs.org/documentation>) para extender tus conocimientos y mejorar tu desempeño.

# [CONFIGURANDO NUESTRA BASE DE DATOS](http://www.maestrosdelweb.com/editorial/curso-symfony2-configurando-bases-de-datos/)

Entraremos al mundo de Doctrine usándolo dentro de nuestro proyecto para así tener ya los datos dinámicamente almacenados en nuestra base de datos y poder manipularlos.

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 entra 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 cuáles 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 podemos usar 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.

## CREANDO TABLAS Y CONOCIENDO 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 poder 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, el 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 ya nos empezará a preguntar 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 parametro name: @ORM\Table(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 indice autonumérico mientras que si es PostgreSQL usará un dato de tipo SERIAL.

Ahora creemos 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 probar 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 podremos ver 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 Articulo 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\ManyToOne(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:

* **@ORM\ManyToOne:** Esta anotación le dice que desde esta tabla de comentarios existe una relación de muchos a uno con la entidad Articles
* **@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 poder 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 podemos ver 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

### NOTA

En cualquiera de los tres casos **doctrine:schema:create**,**doctrine:schema:drop** y **doctrine:eschema:update**podemos usar el parámetro especial “–dump-squl” 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

# [MANIPULANDO DATOS CON DOCTRINE](http://www.maestrosdelweb.com/editorial/curso-symfony2-manipulando-datos-con-doctrine/)

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 y 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.
* **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 comencemos 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 lineas 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('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');

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\GeneratedValue(strategy=”AUTO”). Tambień 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 asterísco 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 obtemos 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 se 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](http://www.doctrine-project.org/docs/orm/2.1/en/reference/dql-doctrine-query-language.html) 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 se 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();

$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 necesitar pasar filtros para el WHERE, podemos hacerlo 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 articulos 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 Controladores, podríamos ( y deberíamos para mantener el código más ordenado y mantenible)  tener todos las consultas relacionadas con los articulos 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')->findArticlesByAuthorAndTitle('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.

# [VALIDACIÓN DE DATOS Y CREACIÓN DE FORMULARIOS](http://www.maestrosdelweb.com/editorial/curso-symfony2-validacion-de-datos-y-creacion-de-formularios/)

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](http://symfony.com/doc/current/book/validation.html#constraints).

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 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 ingles, 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 quisiéramos 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;

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](http://symfony.com/doc/current/book/validation.html#constraints) 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.

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 poder 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, quién ya sabe como validarse por si 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 titulo” y si tenemos más errores los mensajes por cada error.

### CREACIÓN DE FORMULARIOS

La posibilidad de crear formularios es uno de los temas que más me gusta de Symfony ya que los mismos 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](http://www.maestrosdelweb.com/guias/#guia-html5).

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) 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 querer 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. Tenemos que tener 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 poder 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 debe contener 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 formuario. 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">

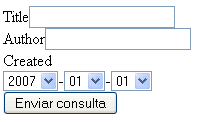
{{ form\_widget(form) }}

<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>



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>

<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>

</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](http://es.wikipedia.org/wiki/Cross_Site_Request_Forgery). Con eso ya vemos como Symfony nos propone ya un estándar de seguridad. A esta seguridad también se suma que por medio de Doctrine también tenemos validado los problemas de SQL Injection.

Si miramos el código podemos notar 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 tremendamente 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)

{

    $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 que para no extender mucho este capítulo lo veremos quizá en otra entrega:

* **form\_errors(form):** Renderiza lo 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 lo 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 el 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 FINAL

En este 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.

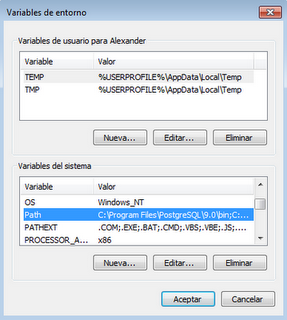
El sub-framework de formularios es uno de los que más me hicieron sentir la diferencia entre usar un framework y no hacerlo y todavía hay muchas otras herramientas que nos permite usar como los formularios embebidos.

En el primer capítulo de esta guía 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 podemos ver 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.

En el siguiente capítulo hablaremos sobre la integración de Ajax en nuestras aplicaciones hechas con Symfony2.

# Tener en cuenta

## [Agregar al path en windows](http://sudokill.blogspot.com/2011/05/agregar-una-variable-al-path-en-windows.html)

[](http://3.bp.blogspot.com/-6lgebKfXQf8/Tca4de_fm4I/AAAAAAAAAzc/nK9u3VemLbA/s1600/Variables+de+entorno.png)

A veces, al seguir algún tutorial .. nos dicen que debemos agregar alguna variable al path, por ejemplo python, java, etc ..

El procedimiento a seguir no es muy complicado y es como sigue:

1. Click derecho sobre Mi PC o Equipo y seleccionamos la opción "Propiedades"

2. Click en "Opciones Avanzadas del Sistema"

3. En la pestaña "Opciones Avanzadas" seleccionamos "Variables de Entorno"

4. En la parte inferior seleccionamos la variable path y clickeamos en Editar

5. Ahora solo nos debemos de ubicar al inicio del valor de la variable, pegar nuestra ruta a agregar y colocarle un punto y coma, para separarla de los otros campos.

¿Qué nos permite esto?

Nos permitirá llamar a los ejecutables de nuestra ruta agregada(por ejemplo: C:\Python27) desde el cmd.

INCLUDE     resource: "@MDWDemoBundle/Resources/config/routing.yml"

URL pattern: /hello/{name} //Llaves indica que es un parametro

UNIDADES 5 – 6, NO ENTENDIDA.