Capitulo 1: Introducción a Symfony 2

Symfony es un framework PHP que nos permite muy fácilmente utilizar la arquitectura MVC (Model-View-Controller). Fue escrito desde un origen para ser utilizado sobre la versión 5 de PHP ya que hace ampliamente uso de la orientación a objetos que caracteriza a esta versión y desde la versión 2 de Symfony se necesita **mínimamente PHP 5.3.3**.

Esta guía está actualizada para utilizar la versión 2.2 de Symfony ya que desde esta versión se crearon muchas mejoras.

Symfony Fue creado por una gran comunidad liderada por Fabien Potencier, quién a la fecha, sigue al frente de este proyecto con una visión muy fuertemente orientada hacia las mejores prácticas que hoy en día forman parte del estándar de desarrollo de software.

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

El concepto de Symfony es no reinventar la rueda, por lo que reutiliza conceptos y desarrollos exitosos de terceros y los integra como librerías para ser utilizados por nosotros. Entre ellos encontramos que integra plenamente uno de los frameworks ORM más importantes dentro de los existentes para PHP llamado Doctrine, el cual es el encargado de la comunicación con la base de datos, permitiendo un control casi total de los datos sin importar si estamos hablando de MySQL, PostgreSQL, SQL server, Oracle, entre otros motores ya que la mayoría de las setencias SQL no son generadas por el programador sino por el mismo Doctrine.

Otro ejemplo de esto es la inclusión del framework Twig, un poderoso motor de plantillas que nos permite separar el código PHP del HTML permitiendo una amplia gama de posibilidades y por sobre todo un extraordinario orden para nuestro proyecto.

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

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

Otra de las funcionalidades más interesantes, es que contiene un subframework para trabajar con formularios. Con esto, creamos una clase orientada a objetos que representa al formulario HTML y una vez hecho esto simplemente lo mostramos y ejecutamos.

Es decir que podemos programarlos utilizando herramientas del framework. Esto nos permite tener en un lugar ordenados todos los formularios de nuestra aplicación incluyendo sus validaciones

realizadas en el lado del servidor, ya que symfony implementa objetos validadores muy sencillos y potentes para asegurar la seguridad de los datos introducidos por los usuarios.

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

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

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

Estos son solo algunos de los conceptos que Symfony nos provee sin ni siquiera entrar en otros que son muy importantes como por ejemplo integración con phpunit para realizar facilmente pruebas unitarias y funcionales, soporte para injección de dependencias, fácil utilización de assetics para administrar y optimizar nuestras imagenes, CSS y código JavaScript, etc.

Entendiendo la Arquitectura MVC

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

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

Veamos una imagen para tratar de entenderlo mejor:

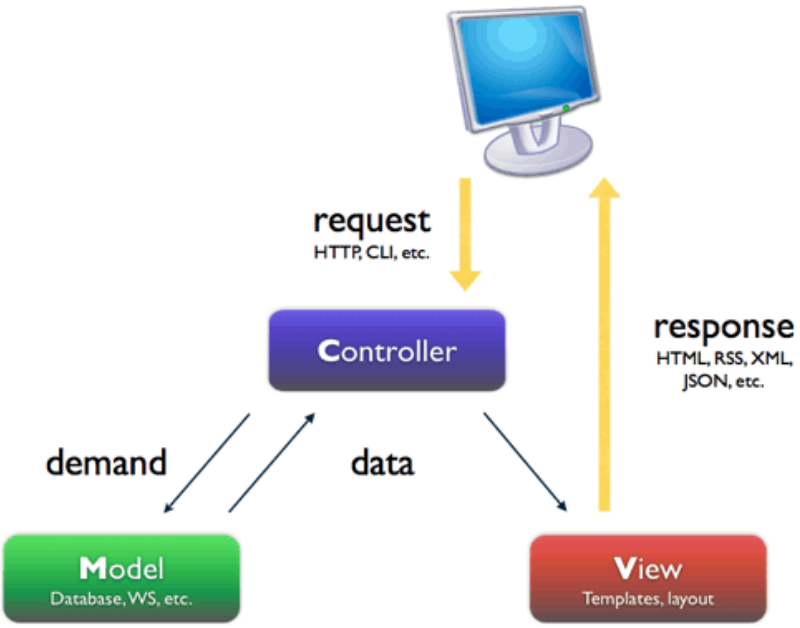


Imagen tomada del manual oficial de Symfony

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

"El controlador *recibe* la petición (Request) y se encarga de *crear y devolver* una respuesta (Response)"; 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.

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)

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

¿Qué es un framework ORM?

La siglas ORM provienen de Object-Relational mapping o Mapeo entre Objetos y Relaciones. Este framework es el encargado de tratar con nuestra base de datos desde la conexión, generación de SQL, manipulación de datos, transacciones y desconexión.

Cuando hablamos de motores de base de datos se dice que cada tabla es una relación, de ahí el nombre de base de datos relacionales, lo que implica que las tablas se encuentran relacionadas entre sí.

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

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

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 cuales son sus campos, tipos de datos, índices, etc. logrando con esto que la **aplicación** conozca el **modelo de los datos** desde un punto de vista orientado a objetos, es decir representado con Clases y Objetos.

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

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

Con esto logramos una abstracción casi del 100% con relación al motor de base de datos, sin importar cual sea, ya que hoy en día la mayoría de ellos se encuentran soportados por Doctrine.

Symfony toma el framework Doctrine y lo incorpora dentro de sí mismo, proporcionándonos todo el soporte necesario para utilizarlo sin preocuparnos por la configuración del mismo.

Utilizando un Motor de Plantillas

Nos hemos acostumbrado a escribir código PHP en el mismo archivo donde se encuentra la estructura HTML de la página. La idea de un motor de plantillas es justamente separar esto en dos capas. La primera sería el programa con la lógica de negocio para resolver el problema especifico de esa página, mientras que la otra sería una página que no contenga el mencionado código sino solo lo necesario para mostrar los datos a los usuarios.

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

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

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

¿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 %}
```

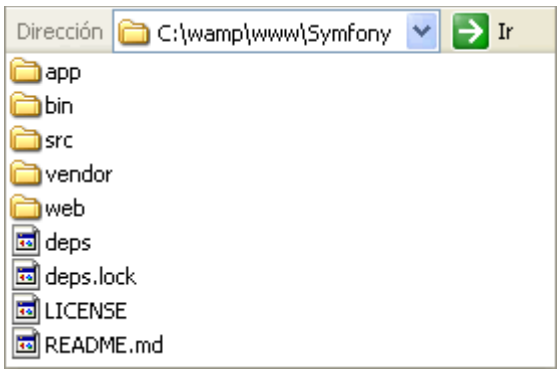
Con Twig mostrar el contenido de una variable es tan simple como usar las dobles llaves **{{ variable }}**, sin necesidad de **echo** ni etiquetas de apertura de PHP (<?php ?>), además de eso **Twig es un lenguaje de plantillas**, lo que nos permite hacer condicionales y estructuras de control muy intuitivas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

Fuente: http://www.maestrosdelweb.com/curso-symfony2-introduccion-instalacion/
http://www.maestrosdelweb.com/curso-symfony2-la-vista-twig/

Capitulo 2

Estructura de un proyecto Symfony (2.x)

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

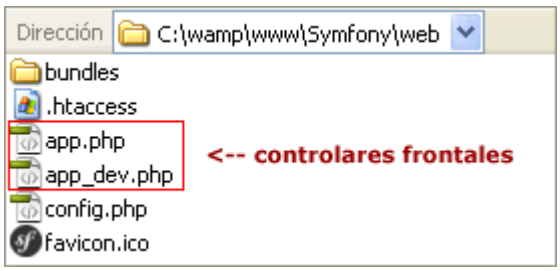


Estructura del proyecto

- **app\:** Aquí se encuentra la configuración correspondiente a todo el proyecto.
- **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. Veremos más de esto en la sección "¿Qué son los Bundles?".
- **vendor\:** En esta carpeta se encuentran los archivos del framework Symfony 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 la utilidades necesarias del framerwork y luego invocan a la página solicitada.

La cache de Symfony

Una de las configuraciones más interesantes de ambos entornos sería con relación a que Symfony maneja una cache donde realiza una especie de pre-compilación de las páginas. Como Symfony maneja tantos archivos y formatos como YAML, XML, Twig y PHP, al momento de ingresar por primera vez al sitio, toma todos los archivos y los convierte a PHP guardándolos dentro de la carpeta app\cache\.

Esto se hace para que no se pierda tiempo generando todo por cada página solicitada. Una vez realizado esto, simplemente las páginas son accedidas por medio de la cache, razón por la cual la primera vez que se ingresa al sitio tardará un poco más que las siguientes y cada vez que se hayan realizado cambios sobre estos archivos debemos borrar la cache para que Symfony la vuelva a generar.

¿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.

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.

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.

Fuente: http://www.maestrosdelweb.com/curso-symfony2-proyecto-bundles/

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.yml 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.

Fuente: http://www.maestrosdelweb.com/curso-symfony2-configurando-bases-de-datos/

Seguridad de acceso

Autenticación vs. Autorización

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

El primer paso es verificar si el usuario está o no autenticado, en tal caso lo deja pasar y el segundo paso es verificar si el usuario tiene el rol necesario para dicha acción. Toda la configuración de Seguridad se encuentra en el archivo:

/app/config/security.yml

Ejemplo del contenido de un archivo de configuracion de seguridad “security.yml”

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

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

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

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

Fuente: http://www.maestrosdelweb.com/curso-symfony2-seguridad-de-acceso/