
# Desarrollo de aplicaciones web con Symfony2

## version 1.0

Juan David Rodríguez García

16 de Julio de 2012

# Contenido

# Curso: Desarrollo de Aplicaciones Web con Symfony2

Contenidos:

# Licencia

# Unidad 1. Inmersión

## *Aplicaciones web*

La World Wide Web es un sistema de documentos de hipertexto o hipermedios enlazados y distribuidos a través de Internet. En sus comienzos, la interacción entre los usuarios de la *WWW* y sus servidores era muy reducida: a través de un software cliente denominado navegador web, el usuario se limitaba a solicitar documentos a los servidores y estos respondían a aquellos con el envío del documento solicitado. A estos documentos que circulaban por el "espacio web" se les denominó páginas web. Cada recurso, conocido como página web, se localiza en este espacio mediante una dirección única que describe tanto al servidor como al recurso: la *URL*. Cada página web puede incorporar las *URL*'*s* de otras páginas como parte de su contenido. De esta manera se enlazan unas con otras.

Han pasado casi 20 años desde la aparición de la Word Wide Web y, aunque en esencia su funcionamiento, basado en el protocolo *HTTP*, sigue siendo el mismo, la capacidad de interacción entre usuarios y servidores se ha enriquecido sustancialmente. De la página web hemos pasado a la aplicación web; un tipo de aplicación informática que adopta, de manera natural, la arquitectura cliente-servidor de la web. De manera que en las peticiones al servidor, el usuario no sólo solicita un recurso, si no que además puede enviar datos. El servidor los procesa y elabora la respuesta que corresponda en función de ellos. Es decir, el servidor construye dinámicamente la página web que le envía al cliente.

Todo el peso de la aplicación reside en el servidor, mientras que el cliente, esto es, el navegador web, se limita a presentar el contenido que recibe mostrándolo al usuario.

Esta evolución comenzó con la aparición de los *CGI*'*s*, que son aplicaciones escritas en cualquier lenguaje de programación y que pueden ser accedidas por el servidor web a petición del cliente, y ha madurado gracias a la aparición de los lenguajes de programación del lado del servidor, como *PHP*, *Java* o *Python*, gracias a los cuales los servidores web (*apache* como ejemplo más conocido y usado) han ampliado su funcionalidades; ya no sólo son capaces de buscar, encontrar y enviar documentos a petición del cliente, si no que también pueden procesar peticiones (acceder a base de datos, realizar peticiones a otros servidores, ejecutar algoritmos, ...) y construir los documentos que finalmente serán enviados al cliente en función de los datos que este les ha proporcionado.

También es relevante en esta evolución de la web la incorporación de procesamiento en los navegadores web mediante lenguajes de *"scripting"* como *javascript*, que permiten la ejecución de ciertos procesos (casi todos relacionados con la manipulación de la interfaz gráfica) en el lado del cliente. De hecho, en la actualidad existen aplicaciones que delegan gran parte de sus procesos al lado del cliente, aunque de todas formas, todo el código es proporcionado desde la parte servidora la primera vez que se solicita el recurso.

Todo esto ha sido bautizado con el omnipresente y manido término de Web 2.0, que en realidad es una manera de referirse a este aumento de la capacidad de interacción con el usuario, y que ha permitido el desarrollo y explosión de las redes sociales y la blogosfera entre otros muchos fenómenos de la reducida pero incesante historia de la *World Wide Web*.

El panorama actual se resume en un interés creciente por las aplicaciones web, hasta el punto de que, en muchos casos, han desplazado a la madura aplicación de escritorio. Son varias las razones que justifican este hecho y, aunque se trata de un tema que por su amplitud no abordaremos en detalle, si que señalaremos algunas:

- Se mejora la mantenibilidad de las aplicaciones gracias a su centralización. Al residir la aplicación en el servidor, desaparece el problema de la distribución de las mismas. Por ejemplo, los cambios en la interfaz de usuario son realizado una sola vez en el servidor y tienen efecto inmediatamente en todos los clientes.

- Se aumenta la capacidad de interacción y comunicación entre los usuarios de la misma, así como de su gestión.
- Al ser *HTTP* un protocolo de comunicación ligero y "sin conexión" (*conectionless*) se evita mantener conexiones abiertas con todos y cada uno de sus clientes, mejorando la eficiencia de los servidores.
- Para utilizar la aplicación, los usuarios tan solo necesitan tener instalado un software denominado navegador web (browser). Esto reduce drásticamente los problemas de portabilidad y distribución. También permite que terminales ligeras, con poca capacidad de proceso, puedan utilizar "grandes" aplicaciones ya que su función se limita a mostrar mediante el navegador los datos que le han sido enviado.
- El desarrollo de dispositivos móviles con conectividad a redes expande el dominio de uso de las aplicaciones web y abre nuevos mercados.
- Se puede acceder a la aplicación desde cualquier punto con acceso a la red donde preste servicio la aplicación. Si se trata de Internet, desde cualquier parte del mundo, si se trata de una intranet desde cualquier parte del mundo con acceso a la misma. Todo ello sin necesidad de instalar nada más que el navegador en la computadora cliente (punto anterior).
- Los lenguajes utilizados para construir las aplicaciones web son relativamente fáciles de aprender. Además algunos de los más utilizados ,como *PHP* y *Python*, se distribuyen con licencias libres y existe una gran cantidad de documentación de calidad disponible en la propia red Internet.
- Recientemente han aparecido en escena varias plataformas y frameworks de desarrollo (por ejemplo *Zend Framework*, *CakePHP*, *symfony*, *Symfony2*) que facilitan la construcción de las aplicaciones web, reduciendo el tiempo de desarrollo y mejorando la calidad.

Obviamente no todo son ventajas; incluso algunas de las ventajas que hemos señalado pueden convertirse en desventajas desde otros puntos de vista. Por ejemplo:

- el hecho de que los cambios realizados en una aplicación web sean efectivos inmediatamente en todos los clientes que la usan, puede dejar sin servicio a un gran número de usuarios si este cambio provoca un fallo (intencionado si se trata de un ataque, o no intencionado si se trata de una modificación que no ha sido debidamente probada). Esto repercute en la necesidad de aumentar las precauciones y la seguridad por parte de los responsables técnicos que mantienen la aplicación.
- La disponibilidad de la aplicación es completamente dependiente de la disponibilidad de la red. Así la aplicación web será útil en entornos donde se garantice la estabilidad de la red. Los programadores necesitan dominar las distintas tecnologías y conceptos que, en estrecha colaboración, conforman la aplicación (*HTTP*, *HTML*, *XML*, *CSS*, *javascript*, lenguajes de scripting del lado del servidor como *PHP*, *Java* o *Python*, ...)
- La triste realidad de las incompatibilidades entre navegadores.

No obstante, la realidad demuestra que el interés por las aplicaciones web es un hecho consumado, lo cual seduce a los programadores de todo el mundo a formarse en las tecnologías y estrategias que permiten desarrollarlas. Este curso tiene como objetivo presentar una de las más exitosas: el desarrollo de aplicaciones web mediante el uso del framework *Symfony2*.

## *Desarrollo rápido y de calidad de aplicaciones web*

La experiencia adquirida tras muchos años de construcción de aplicaciones informáticas de escritorio, dio lugar a la aparición de entornos y frameworks de desarrollo que no solo hacían posible construir rápidamente las aplicaciones, si no que además cuidaban la calidad de las

mismas. Es lo que técnicamente se conoce como *Desarrollo Rápido de Aplicaciones*.

Sin embargo, el desarrollo de aplicaciones web es muy reciente, por lo que estas herramientas de desarrollo rápido y de calidad no han aparecido en el mundo de la web hasta hace bien poco. De hecho la construcción de una aplicación web de calidad no ha estado exenta de dificultades debido a esta carencia. Afortunadamente contamos desde hace unos pocos años con frameworks de desarrollo de aplicaciones web que facilitan el desarrollo de las mismas y están haciendo que el concepto de *Desarrollo Rápido de Aplicaciones* en este campo sea una realidad. *Symfony2* representa una de las herramientas de más éxito para la construcción de aplicaciones web de calidad con *PHP*.

Desarrollar con *Symfony2* hace más sencillo la construcción de aplicaciones web que satisfagan las siguientes características, deseables en cualquier tipo de aplicación informática profesional al margen de sus requisitos específicos.

- Fiabilidad. La aplicación debe responder de forma que sus resultados sean correctos y podamos fiarnos de ellos. También implica que los datos que introducimos como entrada sean debidamente validados para asegurar un comportamiento correcto.
- Seguridad. La aplicación debe garantizar la confidencialidad y el acceso a la misma a usuarios debidamente autentificados y autorizados. En el caso de las aplicaciones web esto es especialmente importante puesto que residen en computadores que, al pertenecer a una red, son accesibles a una gran cantidad de personas. Lo que significa que inevitablemente están expuestas a ser atacadas con fines maliciosos. Por ello deben incorporar mecanismos de protección ante conocidas técnicas de ataque web como puede ser el *Cross Site Scripting (XSS)*.
- Disponibilidad. La aplicación debe prestar servicio cuando se le solicite. Es importante, por tanto, que los cambios requeridos por operaciones relacionadas con el mantenimiento (actualizaciones, migraciones de datos, migraciones de la aplicación a otros servidores, etcétera) sean sencillos de controlar. De esa manera se evitarán largas temporadas de inactividad. La disponibilidad es una de las características más valoradas en las aplicaciones web, ya que el funcionamiento de la misma no depende, por lo general, de sus usuarios si no de los responsables técnicos del sistema donde se encuentre alojada. Hay que pensar en ellos y ponérselo fácil cuando necesiten realizar este tipo de tarea. También es importante que los errores de funcionamiento debidos a errores de programación (*bugs*) sean rápidamente diagnosticados y resueltos para mejorar tanto la disponibilidad como la fiabilidad de la aplicación.
- Mantenibilidad. A medida que se usa una aplicación, aparecen nuevos requisitos y funcionalidades que se desean ofrecer. Un sistema mantenible permite ser extendido sin que ello suponga un coste muy alto, minimizando la probabilidad de introducir errores en los aspectos que ya estaban funcionando antes de emprender la implementación de nuevas característcas.
- Escalabilidad, es decir, que la aplicación pueda ampliarse sin perder calidad en los servicios ofrecidos, lo cual se consigue diseñándola de manera que sea flexible y modular.

## *Presentación del curso*

### *A quién va dirigido*

Este curso va dirigido a personas que ya cuenten con cierta experiencia en la programación de aplicaciones web. A pesar de que *Symfony2* está construido sobre *PHP*, no es tan importante conocer dicho lenguaje como estar familiarizado con las tecnologías de la web y con el paradigma de la programación orientada a objetos.

En la confección del curso hemos supuesto que el estudiante comprende los fundamentos de las tecnologías que componen las aplicaciones web y las relaciones que existen entre ellas:

- El protocolo *HTTP* y los servidores web,
- Los lenguajes de marcado *HTML* y *XML*,
- Las hojas de estilo CSS's,
- Javascript como lenguaje de script del lado del cliente,
- Los lenguajes de script del lado del servidor (*PHP* fundamentalmente),
- Los fundamentos de la programación orientada a objetos (mejor con *PHP*),
- Los fundamentos de las bases de datos relacionales y los sistemas gestores de base de datos.

Obviamente, para seguir el curso, no hay que ser un experto en cada uno de estas tecnologías, pero sí es importante conocerlas hasta el punto de saber cual es el papel que desempeña cada una y como se relacionan entre sí. Cualquier persona que haya desarrollado alguna aplicación web mediante el archiconocido entorno *LAMP* o *WAMP* (*Linux/Windows – Apache – MySQL – PHP*), debería tener los conocimientos necesarios para seguir con provecho este curso.

### *Objetivos del curso*

Cuando finalices el curso habrás adquirido suficiente conocimiento para desarrollar aplicaciones web mediante el empleo del framework de desarrollo en *PHP Symfony2*. Ello significa a grandes rasgos que serás capaz de construir aplicaciones web que:

- Son altamente modulares, extensibles y escalables.
- Separan claramente la lógica de negocio de la presentación, permitiendo que el trabajo de programación y de diseño puedan realizarse independientemente.
- Incorporaran un sistema sencillo, flexible y robusto garantizar la seguridad a los niveles de autentificación y autorización.
- Acceden a las bases de datos a través de una capa de abstracción que permite cambiar de sistema gestor de base de datos sin más que cambiar un parámetro de configuración. No es necesario tocar ni una sola línea de código para ello.
- Cuentan con un flexible sistema de configuración mediante el que se puede cambiar gran parte del comportamiento de la aplicación sin tocar nada de código. Esto permite, entre otras cosas, que se puedan ejecutar en distintos entornos: de producción, de desarrollo y de pruebas, según la fase en la que se encuentre la aplicación.
- Pueden ofrecer el resultado final en varios formatos distintos (*HTML, XML, JSON, RSS, txt, ...*) gracias al avanzado sistema de generación de vistas,
- Cuentan con un potente sistema de gestión de errores y excepciones, especialmente útil en el entorno de desarrollo.
- Implementan un sistema de caché que disminuye los tiempos de ejecución de los scripts.
- Incorpora por defecto mecanismos de seguridad contra ataques XSS y CSRF.
- Pueden ser internacionalizadas con facilidad, aunque la aplicación no se haya desarrollado con la internacionalización como requisito.
- Incorporan un sistema de enrutamiento que proporciona URL's limpias, compuestas exclusivamente por rutas que ocultan detalles sobre la estructura de la aplicación.

- Cuentan con un avanzado sistema de autentificación y autorización.

El curso cubre una porción suficientemente completa sobre las múltiples posibilidades que ofrece *Symfony2* para desarrollar aplicaciones web, incidiendo en sus características y herramientas más fundamentales. El objetivo es que, al final del curso, te sientas cómodo usando *Symfony2* y te resulte sencillo y estimulante continuar profundizando en el framework a medida que tu trabajo lo sugiera.

Cuando emprendas el estudio de este curso, debes tener en cuenta que el aprendizaje de cualquier framework de desarrollo de aplicaciones, y *Symfony2* no es una excepción, es bastante duro en los inicios. Sin embargo merece la pena, pues a medida que se van asimilando los conceptos y procedimientos propios del framework, la productividad aumenta muchísimo.

### *Plan del curso*

Para conseguir los objetivos que nos hemos propuesto hemos optado por un planteamiento completamente práctico en el que se está "picando código" funcional desde el principio del curso.

En la **unidad 2**, sin utilizar *Symfony2* para nada, desarrollamos una sencilla aplicación web en *PHP*. El objetivo de esta unidad es mostrar como se puede organizar el código para que siga los planteamientos del patrón de diseño *Modelo – Vista – Controlador* (*MVC*), gracias al cual separamos completamente la lógica de negocio de la presentación de la información. Es importante comprender los fundamentos de esta organización ya que las aplicaciones desarrolladas con *Symfony2* suelen seguir este patrón. Además en esta unidad se introducen los conceptos de controlador frontal, acción, plantilla y layout, ampliamente usados en el resto del curso.

En la **unidad 3** hacemos una presentación panorámica de *Symfony2*, exponiendo los conceptos fundamentales. En esta unidad volveremos a escribir, esta vez utilizando *Symfony2*, la aplicación de la unidad 2. Dicho ejercicio nos ayudará a realizar la presentación del framework a la vez que servirá como referencia para los conceptos básicos. Avisamos: esta unidad es bastante densa.

La **unidad 4** la hemos dedicado completamente al estudio de un importante patrón de diseño en torno al cual se ha construido *Symfony2*: *La inyección de dependencias*. Es muy importante comprender este concepto para sentirse cómodo con *Symfony2* y para poder extender el framework con soltura.

En la **unidad 5** planteamos el análisis de una aplicación, que aún siendo concebida con criterios pedagógicos, es suficientemente amplia como para ser considerada una aplicación profesional. Se trata de de una aplicación para la gestión de notas (al estilo de los post-it) inspirada en *EverNote*® una herramienta que últimamente está teniendo mucho éxito entre los usuarios de plataformas móviles (smartphones y tabletas). Su desarrollo nos servirá como vehículo para penetrar al interior de *Symfony2* durante el resto del curso.

En las siguientes unidades se construyen progresivamente las distintas funcionalidades de la aplicación analizada en la unidad 5. Cada unidad incide sobre algún aspecto fundamental de *Symfony2*.

En la **unidad 6** profundizamos en el concepto de *routing* y los controladores, conceptos fundamentales sobre los que descansa la lógica de las aplicaciones construidas con *Symfony2*.

En la **unidad 7** se realiza un estudio bastante detallado del ORM *Doctrine2* para el tratamiento de los datos persistentes, es decir, para el acceso a bases de datos. Adelantamos aquí que, a pesar de cubrir los aspectos fundamentales para el desarrollo casi cualquier tipo de aplicación, *Doctrine2* va mucho más allá. Un tratamiento completo de este magnifico *ORM* requiere un curso dedicado al mismo.

La **unidad 8** está dedicada a los servicios de validación de datos y de creación de formularios *HTML*, herramienta fundamental en cualquier aplicación web.

En la **unidad 9** estudiamos el novedoso sistema de seguridad de *Symfony2*, mediante el cual podemos proteger nuestras aplicaciones web en los niveles de autentificación y autorización sin necesidad de picar demasiado código.

La **unidad 10** integra todos los conocimientos acumulados a lo largo del curso para dar forma definitiva a la aplicación de gestión de notas. Aquí incorporamos las plantillas, estilos enriquecidos con *javascript* (*jQuery*) y perfilamos los flecos que se nos han ido quedando en las unidades anteriores.

Por último en la **unidad 11** explicaremos las distintas estrategias que podemos seguir para desarrollar el *backend*, o parte de administración, de las aplicaciones web. Utilizaremos un conocido y potente *bundle* (los *plugins* de *Symfony2*) con el que se pueden construir elegantes y prácticas aplicaciones de *backend* sin necesidad de picar mucho código.

## *Sobre Symfony2*

La primera versión estable de *Symfony2* ha sido desarrollada en un tiempo record de 448 días. El primer *commit* tiene fecha de 3/6/2010, mientras que el último, correspondiente a la versión 2.0.1, es del 25/08/2011. Al extraordinario talento del equipo liderado por *Fabien Potencier*, lider del proyecto, se une los más de 6 años de experiencia acumulada durante el desarrollo de la primera versión de este producto. El resultado ha sido doble; por una parte han construido un conjunto de componentes que actúan como piezas de un *Lego* para la solución de problemas relacionados con la web y, por otro, han elaborado un framework para el desarrollo de aplicaciones web utilizando como pilares tales componentes. Por eso cuando se habla de *Symfony2* debemos interpretar, en función del contexto como con toda palabra polisémica, si se hace referencia a los componentes o al framework. Este curso va de lo segundo.

*Symfony 1*, el antecesor de *Symfony2*, ha sido y sigue siendo un framework líder en el mundo *PHP*. Prueba de ello es la amplia y activa comunidad que lo desarrolla y que lo utiliza. El número de aplicaciones registradas en el trac5 del proyecto también da una idea de la confianza que muchos desarrolladores han depositado en symfony 1. Pues bien, un análisis de la situación actual parece indicar que *Symfony2* va camino de destronar a su antecesor (si no lo ha hecho ya). La actividad del proyecto, que puede seguirse públicamente en su repositorio de *github* (https://github.com/symfony/symfony) y la proliferación de sitios sobre *Symfony2* soportan esta observación. Otro indicativo del éxito que está teniendo esta nueva versión, es el hecho de que los desarrolladores de *Drupal*, han decidido utilizar varios de los componentes de *Symfony2* para el desarrollo de la versión 8 de su producto.

## *La documentación de Symfony2*

Posiblemente una de las características más apreciadas de symfony 1 fue la cantidad y la calidad de documentación oficial que existe. Ello proporciona la tranquilidad de saber que prácticamente cualquier problema que se le presente al programador estará resuelto, o al menos estará resuelto algo parecido, en alguno de los muchos documentos que sobre *symfony 1* se han escrito.

También en *Symfony2*, no podía ser de otra forma, se ha prestado mucha atención a la calidad de la documentación, aunque por el momento únicamente se encuentra en inglés e italiano. El documento imprescindible es el libro oficial (http://symfony.com/doc/current/book/index.html), que está complementado por el libro de recetas (http://symfony.com/doc/current/cookbook/index.html), una guía de referencia con las principales opciones de configuración (http://symfony.com/doc/current/reference/index.html), algunos videos demostrativos, y la documentación de la *API* (http://api.symfony.com/2.0/).

Todo ello lo encontrarás en el sitio oficial de *Symfony2* (http://symfony.com/). Si finalmente terminas atrapado en las redes de *Symfony2* no te quepa duda de que se convertirá en una de tus herramientas imprescindibles. Esperamos que este curso también se encuentre entre ellas.

Como ya hicimos con el curso "*Desarrollo de aplicaciones web con symfony*", hemos desarrollado el texto desde una perspectiva más pedagógica que técnica, ya que es en aquel aspecto donde la documentación oficial de *Symfony2* es más endeble. Este curso supone un apoyo pedagógico para aprender a desarrollar aplicaciones web con *Symfony2*, y no pretende ni sustituir ni desdeñar la documentación oficial. Muy al contrario creemos, como ya hemos señalado, que dicha documentación es muy valiosa y debe formar parte del equipo de recursos necesarios para desarrollar con *Symfony2*.

## *Comentarios*

Aquí puedes enviar comentarios, dudas y sugerencias. Utiliza la barra de *scroll* interna para recorrer todos los mensajes. El formulario de envío se encuentra al final.

*Autor del código: Juan David Rodríguez García <juandavid.rodriguez@ite.educacion.es>*

# Unidad 2: Desarrollo de una aplicación web siguiendo el patrón MVC

## *El patrón MVC en el desarrollo de aplicaciones web*

Muchos de los problemas que aparecen en la ingeniería del software son similares en su estructura. Y, por tanto, se resuelven de manera parecida. A lo largo de la historia de esta disciplina se han elaborado un buen número de esquemas resolutivos que son conocidos con el nombre des *patrones de diseño* [1] y cuyo conocimiento y aplicación son de una inestimáble ayuda a la hora de diseñar y construir una aplicación informática.

Posiblemente uno de los más conocidos y utilizados sea el patrón "*Modelo, Vista, Controlador*" *(MVC)*, que propone organizar una aplicación en tres partes bien diferenciadas y débilmente acopladas entre sí, de manera que los cambios que se produzcan en una no afecten demasiado a las otras (idealmente nada). El nombre del patrón enumera cada una de las partes:

- **El Controlador**. En este artefacto se incluye todo lo referente a la lógica de control de la aplicación, que no tiene nada que ver con las características propias del negocio para el que se está construyendo la aplicación. En el caso de una aplicación web, un ejemplo sería la manipulación de la *request HTTP*.
- **El Modelo**. Donde se implementa todo lo relativo a la lógica de negocio, es decir, los aspectos particulares del problema que la aplicación resuelve. Si, por ejemplo estamos desarrollando un *blog*, un ejemplo sería una librería de funciones para la gestión de los comentarios.
- **La Vista**. Aquí se ubica el código encargado de "pintar" el resultado de los procesos de la aplicación. En una aplicación web la vista se encarga de producir documentos *HTML*, *XML*, *JSON*, etcétera, con los datos que se hayan calculado previamente en la aplicación.

Para que el conjunto funcione, las partes deben interaccionar entre sí. Y en este punto encontramos en la literatura distintas soluciones. La que proponemos en este curso es la mostrada en la siguiente figura:

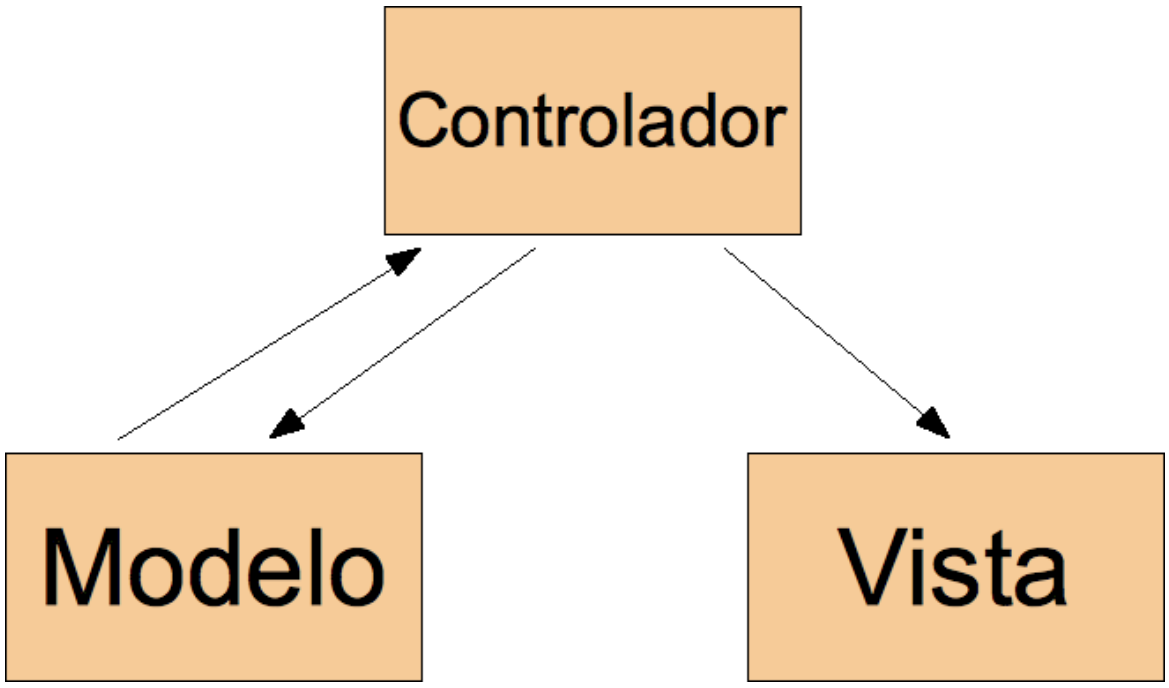


*Diagrama del modelo MVC*

El controlador recibe la orden de entrada y se encarga de procesarla utilizando, si es preciso, los servicios del modelo para ello. Una vez que ha realizado el cálculo entrega los datos "crudos" a la vista y esta se encarga de decorarlos adecuadamente. La característica más importante de esta solución es que la vista nunca interacciona con el modelo.

Las aplicaciones web más típicas pueden plantearse según este patrón: el controlador recibe una petición *HTTP* y la procesa, haciendo uso del modelo calcula los datos de salida y los entrega a la vista, la cual se encarga de construir una respuesta *HTTP* con las cabeceras adecuadas y un **payload** o cuerpo de la respuesta que suele ser un contenido *HTML*, *XML* o *JSON*.

Aunque no todas las aplicaciones web se pueden ajustar a este modelo, si es cierto que la idea de separar responsabilidades o las distintas areas de un problema en sistemas débilmente acoplados, es una estrategia común en las metodología que utilizan el paradigma de programación orientado a objetos. Es lo que se conoce en la terminología anglosajona como *Separation Of Concerns* [2].

En esta unidad vamos a plantear y desarrollar una sencilla aplicación web utilizando como guía de diseño este patrón. De esta manera ilustraremos sus ventajas y nos servirá como material introductorio a la arquitectura de *Symfony2*. Llegados a este punto hemos de indicar que *Fabien Potencier*, lider del proyecto, declara que *Symfony2*, o mejor dicho, la edición standard del framework construido con los componentes de *Symfony2*, no es un framework MVC, ya que no trata para nada del modelo y deja al programador absoluta libertad para incoporar las librerías que más le convenga. En nuestra humilde opinión, creemos que esta apreciación es el resultado de "hilar muy fino", y que, en términos prácticos, *Symfony2* se ajusta al patrón MVC. De hecho, en su edición standard, incorpora *Doctrine* como ORM. Y aunque es cierto que *Doctrine* es un proyecto autónomo y externo a *Symfony2*, no deja de ser un servicio prestado por el framework para la construcción del Modelo. Realmente este tipo de discusiones son interesante y nos ayudan a ejercitar nuestra mente, pero no creemos que sean vitales para aprender a utilizar el framework.

> ***Nota***
>
> En http://fabien.potencier.org/article/49/what-is-symfony2, puedes leer un interesante artículo de Fabien Potencier en el que describe lo que es *Symfony2* y trata su punto de vista acerca de estos detalles.

## *Descripción de la aplicación*

Vamos a construir una aplicación web para elaborar y consultar un repositorio de alimentos con datos acerca de sus propiedades dietéticas. Utilizaremos una base de datos para almacenar dichos datos que consistirá en una sola tabla con la siguiente información sobre alimentos:

- El nombre del alimento,
- la energía en kilocalorías ,
- la cantidad de proteínas,
- la cantidad hidratos de carbono en gramos
- la cantidad de fibra en gramos y
- la cantidad de grasa en gramos,

todo ello por cada 100 gramos de alimento.

Aunque se trata de una aplicación muy sencilla, cuenta con los elementos suficientes para trabajar el aspecto que realmente pretendemos estudiar en esta unidad: la organización del código siguiendo las directrices del patrón *MVC*. Comprobaremos como esta estrategia nos ayuda a mejorar las posibilidades de crecimiento (escalabilidad) y el mantenimiento de las aplicaciones que desarrollamos.

## *Diseño de la aplicación (I). Organización de los archivos*

La "anatomía" de una aplicación web típica consiste en:

1. El código que será procesado en el servidor (*PHP*, *Java*, *Python*, etcétera) para construir dinámicamente la respuesta.
2. Los *Assets*, que podemos traducir como "activos" de la aplicación, y que lo constituyen todos aquellos archivos que se sirven directamente sin ningún tipo de proceso. Suelen ser imágenes, *CSS's* y código *Javascript*.

El servidor web únicamente puede acceder a una parte del sistema de ficheros que se denomina *Document Root*. Es ahí donde se buscan los recursos cuando se realiza una petición a la raíz del servidor a través de la URL `http://el.servidor.que.sea/`. Sin embargo, el código ejecutado para construir dinámicamente la respuesta puede "vivir" en cualquier otra parte, fuera del *Document root* [3].

Tode esto sugiere una manera de organizar el código de la aplicación para que no se pueda acceder desde el navegador más que al código estrictamente imprescindible para que esta funcione. Se trata, simplemente, de colocar en el **Document root** sólo los activos y los scripts *PHP* de entrada a la aplicación. El resto de archivos, fundamentalmente librerías *PHP's* y ficheros de configuración (XML, YAML, JSON, etcétera), se ubicarán fuera del **Document Root** y serán incluidos por los scripts de inicio según lo requieran.

Siguiendo estas conclusiones, nuestra aplicación presentará la siguiente estructura de directorio:

```
.
├── app
└── web
    ├── css
    ├── images
    └── js
```

Configuraremos nuestro servidor web para que el directorio `web` sea su *Document root*, y en `app` colocaremos el código *PHP* y la configuración de la aplicación.

## *Diseño de la aplicación (II). El controlador frontal*

La manera más directa y *naïf* de construir una aplicación en *PHP* consiste en escribir un script *PHP* para cada página de la aplicación. Sin embargo esta práctica presenta algunos problemas, especialmente cuando la aplicación que desarrollamos adquiere cierto tamaño y pretendemos que siga creciendo. Veamos algunos de los problemas más significativos de este planteamiento.

Por lo general, todos los scripts de una aplicación realizan una serie de tareas que son comunes. Por ejemplo: interpretar y manipular la *request*, comprobar las credenciales de seguridad y cargar la configuración. Esto significa que una buena parte del código puede ser compartido entre los scripts. Para ello podemos utilizar el mecanismo de inclusión de ficheros de *PHP* y fin de la historia. Pero, ¿qué ocurre si en un momento dado, cuando ya tengamos escrito mucho código, queremos añadir a todas las páginas de la aplicación una nueva característica que requiere, por ejemplo, el uso de una nueva librería?. Tenemos,

entonces, que añadir dicha modificación a todos los scripts *PHP* de la aplicación. Lo cual supone una degradación en el mantenimiento y un motivo que aumenta la probabilidad de fallos una vez que el cambio se haya realizado.

Otro problema que ocurre con esta estrategia es que si se solicita una página que no tiene ningún script *PHP* asociado, el servidor arrojará un error (404 Not Found) cuyo aspecto no podemos controlar dentro de la propia aplicación (es decir, sin tocar la configuración del servidor web).

Como se suele decir, ¡a grandes males grandes remedios!; si el problema lo genera el hecho de tener muchos scripts, que además comparten bastante código, utilicemos uno solo que se encargue de procesar todas las peticiones. A este único script de entrada se le conoce como **controlador frontal**.

Entonces, ¿cómo puedo crear muchas páginas distintas con un solo script?. La clave está en utilizar la *query string* de la URL como parte de la ruta que define la página que se solicita. El controlador frontal, en función de los parámetro que lleguen en la *query string* determinará que acciones debe realizar para construir la página solicidada.

> ***Nota***
>
> La **query string** es la parte de la URL que contiene los datos que se pasarán a la aplicación web. Por ejemlo, en: `http://tu.servidor/index.php?accion=hola`, la **query string** es: `?accion=hola`.

## *Construcción de la aplicación. Vamos al lío.*

Pues eso, vamos al lío aplicando todo lo que llevamos dicho hasta el momento:

- El patrón de diseño *MVC*,
- La estructura de directorios que expone únicamente los ficheros indispensables para el servidor web y,
- La idea de que todas la peticiones pasen por un solo script, el controlador frontal

## *Creación de la estructura de directorios*

Comenzamos creando la estructura de directorios propuesta anteriormente. Por lo pronto, en nuestro entorno de desarrollo y por cuestiones de comodidad, crearemos la estructura en alguna ubicación dentro del **Document root**.

> ***Nota***
>
> Si estás utilizando como sistema operativo *Ubuntu*, el **Document root** se encuentra en `/var/www`, es ahí donde debes crear un directorio denominado `alimentos` que alojará la estructura propuesta. Si estás utilizando *XAMP* en *Windows*, se encuentra en `C:/xampp/htdocs`.
>
> Es importante resaltar que esto no debería hacerse en un entorno de producción, ya que dejamos al servidor web acceder directamente al directorio app, y es algo que deseamos evitar. Sin embargo, de esta manera podemos añadir todos los proyectos que queramos sin tener que tocar la configuración del servidor web. Lo cual es algo muy agradecido cuando se está desarrollando. En un entorno de producción debemos

> asegurarnos de que el directorio `web` es el **Document root** del servidor (o del VirtualHost de nuestra aplicación, si es que estamos alojando varias webs en un mismo servidor).

Nuestra implementación del patrón *MVC* será muy sencilla; crearemos una clase para la parte del controlador que denominaremos `Controller`, otra para el modelo que denominaremos `Model`, y para los parámetros de configuración de la aplicación utilizaremos una clase que llamaremos `Config`. Los archivos donde se definen estas clases los ubicaremos en el directorio `app`. Por otro lado las Vistas serán implementadas como plantillas *PHP* en el directorio `app/templates`.

Los archivos *CSS*, *Javascript* , las imágenes y el controlador frontal los colocaremos en el directorio web.

Cuando terminemos de codificar, la estructura de ficheros de la aplicación presentará el siguiente aspecto:

```
/var/www/alimentos
          ├── app
          │   ├── templates
          │   ├── Controller.php
          │   ├── Model.php
          │   └── Config.php
          │
          └── web
              ├── css
              ├── images
              ├── js
              └── index.php
```

## *El controlador frontal y el mapeo de rutas*

En cualquier aplicación web se deben definir las *URL's* asociadas a cada una de sus páginas. Para la nuestra definiremos las siguientes:

| URL | Acción |
|---|---|
| http://tu.servidor/alimentos/index.php?ctl=inicio | mostrar pantalla inicio |
| http://tu.servidor/alimentos/index.php?ctl=listar | listar alimentos |
| http://tu.servidor/alimentos/index.php?ctl=insertar | insertar un alimento |
| http://tu.servidor/alimentos/index.php?ctl=buscar | buscar alimentos |
| http://tu.servidor/alimentos/index.php?ctl=ver&id=x | ver el alimento *x* |

A cada una de estas *URL's* les vamos a asociar un método público de la clase `Controller`. Estos métodos se suelen denominar **acciones**. Cada **acción** se encarga de calcular dinámicamente los datos requeridos para construir su página. Podrá utilizar, si le hace falta, lo servicios de la clase `Model`. Una vez calculados los datos, se los pasará a una plantilla donde se realizará, finalmente, la construcción del documento *HTML* que será devuelto al cliente.

Todos estos elementos serán "orquestados" por el controlador frontal, el cual lo implementaremos en un script llamado `index.php` ubicado en el directorio `web`. En concreto,

la responsabilidad del controlador frontal será:

- cargar la configuración del proyecto y las librerías donde implementaremos la parte del Modelo, del Controlador y de la Vista.
- Analizar los parámetros de la petición *HTTP* (**request**) comprobando si la página solicitada en ella tiene asignada alguna acción del Controlador. Si es así la ejecutará, si no dará un error 404 (**page not found**).

Llegados a este punto es importante aclara que, el **controlador** frontal y la clase `Controller`, son distintas cosas y tienen distintas responsabilidades. El hecho de que ambos se llamen *controladores* puede dar lugar a confusiones.

El controlador frontal tiene el siguiente aspecto. Crea el archivo `web/index.php` y copia el siguiente código.

```php
<?php
// web/index.php

// carga del modelo y los controladores
require_once __DIR__ . '/../app/Config.php';
require_once __DIR__ . '/../app/Model.php';
require_once __DIR__ . '/../app/Controller.php';

// enrutamiento
$map = array(
    'inicio' => array('controller' =>'Controller', 'action' =>'inicio'),
    'listar' => array('controller' =>'Controller', 'action' =>'listar'),
    'insertar' => array('controller' =>'Controller', 'action' =>'insertar'),
    'buscar' => array('controller' =>'Controller', 'action' =>'buscarPorNombre'),
    'ver' => array('controller' =>'Controller', 'action' =>'ver')
);

// Parseo de la ruta
if (isset($_GET['ctl'])) {
    if (isset($map[$_GET['ctl']])) {
        $ruta = $_GET['ctl'];
    } else {
        header('Status: 404 Not Found');
        echo '<html><body><h1>Error 404: No existe la ruta <i>' .
                $_GET['ctl'] .
                '</p></body></html>';
        exit;
    }
} else {
    $ruta = 'inicio';
}

$controlador = $map[$ruta];
// Ejecución del controlador asociado a la ruta

if (method_exists($controlador['controller'],$controlador['action'])) {
    call_user_func(array(new $controlador['controller'], $controlador['action']));
} else {

    header('Status: 404 Not Found');
```

```
     echo '<html><body><h1>Error 404: El controlador <i>' .
             $controlador['controller'] .
             '->' .
```

```
44              $controlador['action'] .
45              '</i> no existe</h1></body></html>';
46  }
```

- En las líneas 5-7 se realiza la carga de la configuración del modelo y de los controladores.
- En las líneas 10-16 se declara un array asociativo cuya función es definir una tabla para mapear (asociar), rutas en acciones de un controlador. Esta tabla será utilizada a continuación para saber qué acción se debe disparar.
- En las líneas 19-31 se lleva a cabo el parseo de la *URL* y la carga de la acción, si la ruta está definida en la tabla de rutas. En caso contrario se devuelve una página de error. Observa que hemos utilizado la función `header()` de *PHP* para indicar en la cabecera *HTTP* el código de error correcto. Además enviamos un pequeño documento *HTML* que informa del error. También definimos a `inicio` como una ruta por defecto, ya que si la **query string** llega vacía, se opta por cargar esta acción.

### *Nota*

En honor a la verdad tenemos que decir que lo que estamos llamando parseo de la *URL*, no es tal. Simplemente estamos extrayendo el valor de la variable `ctl` que se ha pasado a través de la petición *HTTP*. Sin embargo, hemos utilizado este termino porque lo ideal sería que, en lugar de utilizar parámetros de la petición *HTTP* para resolver la ruta, pudiésemos utilizar rutas *limpias* (es decir, sin caracteres `?` ni `&` ) del tipo:

```
http://tu.servidor/index.php/inicio
http://tu.servidor/index.php/buscar
http://tu.servidor/index.php/ver/5
```

En este caso sí es necesario proceder a un parseo de la *URL* para buscar en la tabla de rutas la acción que le corresponde. Esto, obviamente, es más complejo. Pero es lo que hace (y muchas cosas más) el componente *Routing* de *Symfony2*. En la siguiente unidad realizaremos un primer acercamiento a dicho componente que te dará una idea de la potencia del mismo.

## *Las acciones del Controlador. La clase `Controller`.*

Ahora vamos a implementar las acciones asociadas a las *URL's* en la clase `Controller`. Crea el archivo `app/Controller.php` y copia el siguiente código:

```
1  <?php
2
3  class Controller
4  {
5
6      public function inicio()
7      {
8          $params = array(
```

Las acciones del Controlador. La clase Controller.

```
 9              'mensaje' => 'Bienvenido al curso de Symfony2',
10              'fecha' => date('d-m-yyy'),
11          );
12          require __DIR__ . '/templates/inicio.php';
13      }
14
15      public function listar()
16      {
17          $m = new Model(Config::$mvc_bd_nombre, Config::$mvc_bd_usuario,
18                      Config::$mvc_bd_clave, Config::$mvc_bd_hostname);
19
20          $params = array(
21              'alimentos' => $m->dameAlimentos(),
22          );
23
24          require __DIR__ . '/templates/mostrarAlimentos.php';
25      }
26
27      public function insertar()
28      {
29          $params = array(
30              'nombre' => '',
31              'energia' => '',
32              'proteina' => '',
33              'hc' => '',
34              'fibra' => '',
35              'grasa' => '',
36          );
37
38          $m = new Model(Config::$mvc_bd_nombre, Config::$mvc_bd_usuario,
39                      Config::$mvc_bd_clave, Config::$mvc_bd_hostname);
40
41          if ($_SERVER['REQUEST_METHOD'] == 'POST') {
42
43              // comprobar campos formulario
44              if ($m->validarDatos($_POST['nombre'], $_POST['energia'],
45                       $_POST['proteina'], $_POST['hc'], $_POST['fibra'],
46                       $_POST['grasa'])) {
47                  $m->insertarAlimento($_POST['nombre'], $_POST['energia'],
48                           $_POST['proteina'], $_POST['hc'], $_POST['fibra'],
49                           $_POST['grasa']);
50                  header('Location: index.php?ctl=listar');
51
52              } else {
53                  $params = array(
54                      'nombre' => $_POST['nombre'],
55                      'energia' => $_POST['energia'],
56                      'proteina' => $_POST['proteina'],
57                      'hc' => $_POST['hc'],
58                      'fibra' => $_POST['fibra'],
59                      'grasa' => $_POST['grasa'],
60                  );
61                  $params['mensaje'] = 'No se ha podido insertar el alimento. Revisa el formulario';
62              }
```

```
63          }
64
65          require __DIR__ . '/templates/formInsertar.php';
66      }
67
68      public function buscarPorNombre()
69      {
70          $params = array(
71              'nombre' => '',
72              'resultado' => array(),
73          );
74
```

```
 75          $m = new Model(Config::$mvc_bd_nombre, Config::$mvc_bd_usuario,
 76                      Config::$mvc_bd_clave, Config::$mvc_bd_hostname);
 77
 78          if ($_SERVER['REQUEST_METHOD'] == 'POST') {
 79              $params['nombre'] = $_POST['nombre'];
 80              $params['resultado'] = $m->buscarAlimentosPorNombre($_POST['nombre']);
 81          }
 82
 83          require __DIR__ . '/templates/buscarPorNombre.php';
 84      }
 85
 86      public function ver()
 87      {
 88          if (!isset($_GET['id'])) {
 89              throw new Exception('Página no encontrada');
 90          }
 91
 92          $id = $_GET['id'];
 93
 94          $m = new Model(Config::$mvc_bd_nombre, Config::$mvc_bd_usuario,
 95                      Config::$mvc_bd_clave, Config::$mvc_bd_hostname);
 96
 97          $alimento = $m->dameAlimento($id);
 98
 99          $params = $alimento;
100
101          require __DIR__ . '/templates/verAlimento.php';
102      }
103
104  }
```

Esta clase implementa una serie de métodos públicos, que hemos denominado acciones para indicar que son métodos asociados a *URL's*. Fíjate como en cada una de las acciones se declara un array asociativo ( `params` ) con los datos que serán pintados en la plantilla. Pero en ningún caso hay información acerca de como se pintarán dichos datos. Por otro lado, casi todas las acciones utilizan un objeto de la clase `Models` para realizar operaciones relativas a la lógica de negocio, en nuestro caso a todo lo relativo con la gestión de los alimentos.

Para comprender el funcionamiento de las acciones, comencemos por Analizar la función `listar()` . Comienza declarando un objeto del modelo (línea 17) para pedirle posteriormente el conjunto de alimentos almacenados en la base de datos. Los datos recopilados son almacenados en el array asociativo `params` (líneas 20-22). Por último incluye el archivo `/templates/mostrarAlimentos.php` (línea 24). Tal archivo, que denominamos **plantilla**, será el encargado de construir el documento *HTML* con los datos del array `params`. Observa que todas las acciones tienen la misma estructura: realizan operaciones, recojen datos y llaman a una plantilla para construir el documento *HTML* que será devuelto al cliente.

Observa también que en las acciones del controlador no hay ninguna operación que tenga que ver con la lógica de negocio, todo lo que se hace es lógica de control.

Analicemos ahora la acción `insertar()`, cuya lógica de control es algo más compleja debido a que tiene una doble funcionalidad:

1. Enviar al cliente un formulario HTML,
2. Validar los datos sobre un alimento que se reciben desde el cliente para insertarlos en la base de datos.

La función comienza por declarar un array asociativo con campos vacíos que coinciden con los de la tabla alimento (líneas 29-36). A continuación comprueba si la petición se ha

realizado mediante la operación *POST* (línea 41), si es así significa que se han pasado datos a través de un formulario, si no es así quiere decir que simplemente se ha solicitado la página para ver el formulario de inserción. En este último caso, la acción pasa directamente a incluir la plantilla que pinta el formulario (línea 65). Como el array de parámetros está vacío, se enviará al cliente un formulario con los campos vacíos (cuando veas el código de la plantilla lo verás en directo, por lo pronto basta con saber que es así).

Por otro lado, si la petición a la acción `insertar()` se ha hecho mediante la operación *POST*, significa que se han enviado datos de un formulario desde el cliente (precisamente del formulario vacío que hemos descrito un poco más arriba). Entonces se extraen los datos de la petición, se comprueba si son válidos (línea 44) y en su caso se realiza la inserción (línea 47) y una redirección al listado de alimentos (línea 50). Si los datos no son válidos, entonces se rellena el array de parámetros con los datos de la petición (líneas 53-60) y se vuelve a pintar el formulario, esta vez con los campos rellenos con los valores que se enviaron en la petición anterior y con un mensaje de error.

Todo el proceso que acabamos de contar no tiene nada que ver con la lógica de negocio; esto es, no decide cómo deben validarse los datos, ni cómo deben insertarse en la base de datos, esas tareas recaen en el modelo (el cual, obviamente debemos utilizar). Lo importante aquí es que debe haber una operación de validación para tomar una decisión: insertar los datos o reenviar el formulario relleno con los datos que envió el usuario y con un mensaje de error. Es decir, únicamente hay código que implementa la lógica de control.

> ***Nota***
>
> El esquema de control que se acaba de presentar resulta muy práctico y ordenado para implementar acciones que consisten en recopilar datos del usuario y realizar algún proceso con ellos (almacenarlos en una base de datos, por ejemplo). A lo largo del curso aparecerá, con más o menos variaciones, en varias ocasiones.

## *La implementación de la Vista.*

### *Las plantillas PHP*

Ahora vamos a pasar a estudiar la parte de la Vista, representada en nuestra solución por las plantillas. Aunque en el análisis que estamos haciendo ya hemos utilizado la palabra "plantilla" en varias ocasiones, aún no la hemos definido con precisión. Así que comenzamos por ahí.

Una plantilla es un fichero de texto con la información necesaria para generar documentos en cualquier formato de texto (*HTML*, *XML*, *CSV*, *LaTeX*, *JSON*, etcétera). Cualquier tipo de plantilla consiste en un documento con el formato que se quiere generar, y con variables expresadas en el lenguaje propio de la plantilla y que representas a lo valores que son calculados dinámicamente por la aplicación.

Cuando desarrollamos aplicaciones web con *PHP*, la forma más sencilla de implementar plantillas es usando el propio *PHP* como lenguaje de plantillas. ¿Qué significa esto? Acudimos al refranero popular y decimos aquello de que *una imagen vale más que mil palabras*. Con todos vosotros un ejemplo de plantilla *HTML* que usa *PHP* como lenguaje de plantillas (dedícale un ratito a observarla y analizarla, ¿qué es lo que te llama la atención en el aspecto del código *PHP* que aparece?)

```
1       <table>
2           <tr>
3               <th>alimento (por 100g)</th>
4               <th>energía (Kcal)</th>
5               <th>grasa (g)</th>
6           </tr>
7           <?php foreach ($params['alimentos'] as $alimento) :?>
8           <tr>
9               <td><a href="index.php?ctl=ver&id=<?php echo $alimento['id']?>">
10                      <?php echo $alimento['nombre'] ?>
11                  </a>
12          </td>
13              <td><?php echo $alimento['energia']?></td>
14              <td><?php echo $alimento['grasatotal']?></td>
15          </tr>
16          <?php endforeach; ?>
17
18      </table>
```

Esencialmente no es más que un trozo de documento *HTML* donde la información dinámica se obtiene procesando código *PHP*. La característica principal de este código *PHP* es que debe ser escueto y corto. De manera que no "contamine" la estructura del *HTML*. Por ello cada instrucción *PHP* comienza y termina en la misma línea. La mayor parte de estas instrucciones son `echo's` de variables escalares. Pero también son muy usuales la utilización de bucles `foreach` - `endforeach` para recorrer arrays de datos, así como los bloques condicionales `if` - `endif` para pintar bloques según determinadas condiciones.

En el ejemplo de más arriba se genera el código *HTML* de una tabla que puede tener un número variable de filas. Se recoje en la plantilla el parámetro `alimentos` , que es un array con datos de alimentos, y se genera una fila por cada elemento del array con información de la *URL* de una página sobre el alimento (línea 9), y su nombre, energía y grasa total (líneas 10-14).

Observa también la forma de construir el bucle `foreach`, se abre en la línea 7 y se cierra en la 16. Lo particular de la sintaxis de este tipo de bucle para plantillas es que la instrucción `foreach` que lo abre terminan con el caracter `:`. Y la necesidad de cerrarlo con un `<?php endforeach; ?>`.

### *El layout y el proceso de decoración de plantillas*

En una aplicación web, muchas de las páginas tienen elementos comunes. Por ejemplo, un caso típico es la cabecera donde se coloca el mensaje de bienvenida, el menú y el pie de página. Este hecho, y la aplicación del conocido principio de buenas prácticas de programación *DRY* (*Don't Repeat Yourself*, No Te Repitas), lleva a que cualquier sistema de plantillas que se utilice para implementar la vista utilice otro conocido patrón de diseño: El *Decorator*, o Decorador [4]. Aplicado a la generación de vistas la solución que ofrece dicho patrón es la de añadir funcionalidad adicional a las plantillas. Por ejemplo, añadir el menú y el pie de página a las plantillas que lo requieran, de manera que dichos elementos puedan reutilizarse en distintas plantillas. Literalmente se trata de *decorar* las plantillas con elementos adicionales reutilizables.

Nuestra implementación del patrón *Decorator* es muy simple y, por tanto limitada, pero suficiente para asimilar las bases del concepto y ayudarnos a comprender más adelante la filosofía del sistema de plantillas de *Symfony2*, denomindado *twig* [5].

Nuestras plantillas serán ficheros *PHP* del tipo que acabamos de explicar, y las ubicaremos en el directorio `app/templates`. Como ya has visto en el código del controlador, las acciones finalizan incluyendo alguno de estos archivos. Comencemos por estudiar la plantilla

app/templates/mostrarAlimentos.php, que es la que utiliza la acción listar() para pintar los alimentos que obtiene del modelo. Crea el archivo app/templates/mostrarAlimentos.php con el siguiente código:

app/templates/mostrarAlimentos.php

```
 1  <?php ob_start() ?>
 2
 3  <table>
 4      <tr>
 5          <th>alimento (por 100g)</th>
 6          <th>energía (Kcal)</th>
 7          <th>grasa (g)</th>
 8      </tr>
 9      <?php foreach ($params['alimentos'] as $alimento) :?>
10      <tr>
11          <td><a href="index.php?ctl=ver&id=<?php echo $alimento['id']?>">
12                  <?php echo $alimento['nombre'] ?></a></td>
13          <td><?php echo $alimento['energia']?></td>
14          <td><?php echo $alimento['grasatotal']?></td>
15      </tr>
16      <?php endforeach; ?>
17
18  </table>
19
20
21  <?php $contenido = ob_get_clean() ?>
22
23  <?php include 'layout.php' ?>
```

Como ves, las líneas 3-18 son las que se han puesto como ejemplo de plantilla *PHP* hace un momento. La novedad son las líneas 1 y 21-23. En ellas está la clave del nuestro proceso de decoración. Para comprenderlo del todo es importante echarle un vistazo al fichero app/templates/layout.php, incluido al final de la plantilla. Créalo y copia el siguiente código:

app/templates/layout.php

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
 2  <html>
 3      <head>
 4          <title>Información Alimentos</title>
 5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 6          <link rel="stylesheet" type="text/css" href="<?php echo 'css/'.Config::$mvc_vis_css ?>" />
 7
 8      </head>
 9      <body>
10          <div id="cabecera">
11              <h1>Información de alimentos</h1>
12          </div>
13
14          <div id="menu">
15              <hr/>
16              <a href="index.php?ctl=inicio">inicio</a> |
17              <a href="index.php?ctl=listar">ver alimentos</a> |
```

```
18              <a href="index.php?ctl=insertar">insertar alimento</a> |
19              <a href="index.php?ctl=buscar">buscar por nombre</a> |
```

```
20              <a href="index.php?ctl=buscarAlimentosPorEnergia">buscar por energia</a> |
21              <a href="index.php?ctl=buscarAlimentosCombinada">búsqueda combinada</a>
22              <hr/>
23          </div>
24
25          <div id="contenido">
26              <?php echo $contenido ?>
27          </div>
28
29          <div id="pie">
30              <hr/>
31              <div align="center">- pie de página -</div>
32          </div>
33      </body>
34  </html>
```

El nombre del fichero es bastante ilustrativo, es un *layout HTML*, es decir, un diseño de un documento *HTML* que incluye como elemento dinámico a la variable `$contenido` (línea 26), la cual esta definida al final de la plantilla `mostrarAlimentos.php`, y cuyo contenido es precisamente el resultado de interpretar las líneas comprendidas entre el `ob_start()` y `$contenido = ob_get_clean()` . En la documentación de estas funciones (http://php.net/manual/es/function.ob-start.php) puedes ver que el efecto de `ob_start()` es enviar todos los resultados del script desde la invocación de la función a un buffer interno. Dichos resultados se recojen a través de la función `ob_get_clean()` . De esa manera conseguimos decorar la plantilla con el layout. Esta técnica es utilizada en todas las plantillas, de manera que todos los elementos comunes a todas las páginas son escritos una sóla vez en `layout.php` y reutilizados con todas las plantillas generadas con los datos de cada acción.

Observa que el *layout* que hemos propuesto incluye:

- los estilos *CSS* (línea 6),
- el menú de la aplicación (líneas 14-23)
- el pie de página (líneas 29-32)

A continuación mostramos el código del resto de las plantillas:

`app/templates/inicio.php`

```
1  <?php ob_start() ?>
2  <h1>Inicio</h1>
3  <h3> Fecha: <?php echo $params['fecha'] ?> </h3>
4  <?php echo $params['mensaje'] ?>
5
6  <?php $contenido = ob_get_clean() ?>
7
8  <?php include 'layout.php' ?>
```

`app/templates/formInsertar.php`

```
1  <?php ob_start() ?>
2
3  <?php if(isset($params['mensaje'])) :?>
4  <b><span style="color: red;"><?php echo $params['mensaje'] ?></span></b>
5  <?php endif; ?>
6  <br/>
```

```
 7  <form name="formInsertar" action="index.php?ctl=insertar" method="POST">
 8      <table>
 9          <tr>
10              <th>Nombre</th>
11              <th>Energía (Kcal)</th>
12              <th>Proteina (g)</th>
13              <th>H. de carbono (g)</th>
14              <th>Fibra (g)</th>
15              <th>Grasa total (g)</th>
16          </tr>
17          <tr>
18              <td><input type="text" name="nombre" value="<?php echo $params['nombre'] ?>" /></td>
19              <td><input type="text" name="energia" value="<?php echo $params['energia'] ?>" /></td>
20              <td><input type="text" name="proteina" value="<?php echo $params['proteina'] ?>" /></td>
21              <td><input type="text" name="hc" value="<?php echo $params['hc'] ?>" /></td>
22              <td><input type="text" name="fibra" value="<?php echo $params['fibra'] ?>" /></td>
23              <td><input type="text" name="grasa" value="<?php echo $params['grasa'] ?>" /></td>
24          </tr>
25
26      </table>
27      <input type="submit" value="insertar" name="insertar" />
28  </form>
29  * Los valores deben referirse a 100 g del alimento
30
31  <?php $contenido = ob_get_clean() ?>
32
33  <?php include 'layout.php' ?>
```

app/templates/buscarPorNombre.php

```
 1      <?php ob_start() ?>
 2
 3       <form name="formBusqueda" action="index.php?ctl=buscar" method="POST">
 4
 5          <table>
 6              <tr>
 7                  <td>nombre alimento:</td>
 8                  <td><input type="text" name="nombre" value="<?php echo $params['nombre']?>">(puedes utilizar '%' como comodín)</td>
 9
10                  <td><input type="submit" value="buscar"></td>
11              </tr>
12          </table>
13
14          </table>
15
16       </form>
17
18       <?php if (count($params['resultado'])>0): ?>
19       <table>
20          <tr>
21              <th>alimento (por 100g)</th>
22              <th>energía (Kcal)</th>
23              <th>grasa (g)</th>
24          </tr>
25          <?php foreach ($params['resultado'] as $alimento) : ?>
26              <tr>
27                  <td><a href="index.php?ctl=ver&id=<?php echo $alimento['id'] ?>">
28                          <?php echo $alimento['nombre'] ?></a></td>
29                  <td><?php echo $alimento['energia'] ?></td>
30                  <td><?php echo $alimento['grasatotal'] ?></td>
31              </tr>
32          <?php endforeach; ?>
33
34      </table>
35       <?php endif; ?>
36
37       <?php $contenido = ob_get_clean() ?>
38
39       <?php include 'layout.php' ?>
```

app/templates/verAlimento.php

```
1  <?php ob_start() ?>
2
3  <h1><?php echo $params['nombre'] ?></h1>
4  <table border="1">
5
6      <tr>
```

```
 7          <td>Energía</td>
 8          <td><?php echo $alimento['energia'] ?></td>
 9
10      </tr>
11      <tr>
12          <td>Proteina</td>
13          <td><?php echo $alimento['proteina']?></td>
14
15      </tr>
16      <tr>
17          <td>Hidratos de Carbono</td>
18          <td><?php echo $alimento['hidratocarbono']?></td>
19
20      </tr>
21      <tr>
22          <td>Fibra</td>
23          <td><?php echo $alimento['fibra']?></td>
24
25      </tr>
26      <tr>
27          <td>Grasa total</td>
28          <td><?php echo $alimento['grasatotal']?></td>
29
30      </tr>
31
32  </table>
33
34
35  <?php $contenido = ob_get_clean() ?>
36
37  <?php include 'layout.php' ?>
```

Todas las plantillas recurren al uso de las funciones `ob_start()` y `ob_get_clean()` y a la inclusión del *layout* para realizar el proceso de decoración.

## *El Modelo. Accediendo a la base de datos*

Ya sólo nos queda presentar al Modelo. En nuestra aplicación se ha implementado en la clase `Model` y esta compuesto por una serie de funciones para persistir datos en la base de datos, recuperarlos y realizar su validación.

Dependiendo de la complejidad del negocio con el que tratemos, el modelo puede ser más o menos complejo y, además de tratar con la persistencia de los datos puede incluir funciones para ofrecer otros servicios relacionados con el negocio en cuestión. Crea el archivo `app/Model.php` y copia el siguiente código:

`app/Model.php`

```
 1  <?php
 2
 3  class Model
 4  {
 5      protected $conexion;
 6
 7      public function __construct($dbname,$dbuser,$dbpass,$dbhost)
```