



Desarrollo de aplicaciones web con symfony 1.4

version 1.0

Juan David Rodríguez García

23 de Julio de 2012

Contenido

Curso: Desarrollo de Aplicaciones Web con symfony 1.4	1
Unidad 1. Conceptos básicos para el seguimiento del curso.	2
Aplicaciones web	2
Desarrollo rápido y de calidad de aplicaciones web	3
Presentación del curso	4
A quién va dirigido	4
Objetivos del curso	5
Plan del curso	6
Sobre symfony	7
La documentación de symfony	7
Instalación de symfony	8
Instalación en Windows de XAMPP + symfony	8
Instalación de symfony en Ubuntu	9
Unidad 2: Desarrollo de una aplicación web siguiendo el patrón MVC	10
El patrón MVC en el desarrollo de aplicaciones web	10
Descripción de la aplicación	11
Diseño de la aplicación (I). Organización de los archivos	11
Diseño de la aplicación (II). El controlador frontal	12
Construcción de la aplicación. Vamos al lío.	13
Creación de la estructura de directorios	13
El controlador frontal y el mapeo de rutas	14
Las acciones del Controlador. La clase Controller.	16
La implementación de la Vista.	19
Las plantillas PHP	19
El layout y el proceso de <i>decoración de plantillas</i>	20
El Modelo. Accediendo a la base de datos	24
La configuración de la aplicación	26
Incorporar las CSS's	26
La base de datos	27
Unidad 3. Desarrollo de una aplicación con Symfony	29
Creamos el proyecto.	29
Generación del esquema, el modelo y los formularios	33
Las aplicaciones y los módulos	36
¡Y ahora a programar!	41
Acción de inicio (index)	42
Búsqueda de alimentos por nombre	45

Búsqueda de alimentos por energía	48
Búsqueda combinada.	49
Inserción de registros	53
Ejercicios de ampliación.	55
Conclusión	56
Unidad 4: Análisis de la aplicación “Gestor Documental”	58
Descripción de la aplicación	58
Catálogo de requisitos	59
Gestión de usuarios	59
Gestión de documentos	59
Comentarios	60
Puntuación	60
Modelo de datos.	61
Descripción del proceso de subida de archivos	63
Escenarios	64
Búsqueda y listado de documentos	64
Subida de documentos	65
Subida de nuevas versiones de los documentos	66
Modificación de los metadatos de un documento	67
Añadir un comentario a una versión	68
Valorar una versión.	69
Gestión de Usuarios	69
Gestión de Categorías y Gestión de Tipos de Archivos	71
Diseño arquitectónico.	71
Frontend: el gestor documental	71
Backend: administración del gestor documental	71
Inicio de sesión. Los plugins de symfony	72
Resumen del diseño arquitectónico	73
Recursos para la construcción de la aplicación	73
Conclusión	73
Unidad 5: Profundizando en el modelo.	75
Construcción del proyecto	75
La capa de abstracción de base de datos en symfony	78
Generación del ORM de Propel (Modelo)	79
Jerarquía de clases del modelo con Propel.	80
Métodos de las clases registro	81
Métodos de las clases peer y el objeto Criteria.	82
Implementación del listado documentos	86

Listado de todos los documentos	86
Implementación del filtro de documentos	90
Procesamiento de la petición. El objeto SfWebRequest.	91
Implementación del páginado.	93
Implementación de operaciones sobre documentos concretos.	95
Descarga de las versiones de un documento.	95
Mostrar los metadatos de un documento.	97
Conclusión	99
Unidad 6: La sesión de usuario en el servidor. El inicio de sesión.	100
La sesión de usuario en el servidor	100
De vuelta con el filtro perdido.	103
Aplicaciones seguras. Autentificación y autorización.	106
Autentificación y Autorización	106
Seguridad en la acción	106
El modelo de seguridad	109
Registro de usuario o inicio de sesión	113
Conclusiones.	118
Unidad 7: Profundizando en la arquitectura MVC de Symfony	119
El patrón MVC en symfony	119
La parte del controlador.	121
El controlador frontal	121
Los filtros y las acciones	122
Asociación de la plantilla a la acción	125
Implementación de un filtro para la selección de CSS en función del perfil del usuario.	126
La parte de la vista	128
La vistas HTML	128
El proceso de decoración. Los layouts.	128
Uso de javascripts y CSS's. Los ficheros de configuración view.yml	129
Asociación de la vista a la acción.	131
Los helpers	131
Partials y componentes.	133
Las vistas no HTML	136
Canales de noticias RSS.	137
Especificando el formato de la petición.	137
Conclusión	139
Unidad 8: El framework de formularios de Symfony	140

Formularios HTML	140
Componentes de la interfaz gráfica	140
Procesamiento de los datos introducidos en el formulario.	142
Estructura de los formularios de Symfony	143
Los widgets	144
Los validadores	146
Los formularios	148
Creación y modificación de documentos.	152
El formulario DocumentosForm	153
Implementación de la creación de nuevos documentos	156
Implementación de la modificación de documentos existentes	159
Subida de versiones	164
El formulario VersionesForm	164
Implementación de la subida de nuevas versiones	164
Conclusión	167
Unidad 9: Desarrollo de la parte de administración.	168
Transformación del módulo de inicio de sesión en un plugin	168
Los plugins de symfony	168
Nuestro plugin de inicio de sesión	170
Desarrollo de la aplicación de administración (backend).	174
Creación de la aplicación backend	174
El generador automático de módulos de administración.	176
Generación de los módulos de administración	177
Cambios propuestos para adaptar los módulos de administración a nuestras necesidades.	178
Estructura de un módulo de administración generado automáticamente.	179
El fichero de configuración generator.yml	181
Implementación de las modificaciones propuestas	183
Ajustes globales	183
Ajustes del módulo usuario	184
Ajustes del módulo de gestión de categorías.	187
Ajustes del módulo de gestión de tipos de ficheros permitidos	188
Conclusión	188
Unidad 10: Internacionalización y enrutamiento	190
Internacionalización y Localización	190
La cultura del usuario.	190
Traducción de la interfaz de las aplicaciones internacionalizadas*	193

Traducción de campos de tablas de la base de datos	194
Enrutamiento	195
Conclusión	201
Indices y tablas	202

Curso: Desarrollo de Aplicaciones Web con symfony 1.4

Curso: Desarrollo de Aplicaciones Web con symfony 1.4

por: Juan David Rodríguez García <juandavid.rodriguez@ite.educacion.es>

Contenidos:

Unidad 1. Conceptos básicos para el seguimiento del curso.

Unidad 1. Conceptos básicos para el seguimiento del curso.

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**1, 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.

Desarrollo rápido y de calidad de aplicaciones web

- 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” (*connectionless*) 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 *symfony*) 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 symfony*.

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

Presentación del curso

possible 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. *Symfony* representa una de las herramientas de más éxito para la construcción de aplicaciones web de calidad con PHP.

Desarrollar con *symfony* 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 autenticados 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 características.
- **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 *symfony* 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

Objetivos del curso

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 symfony*. 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 potente 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.

El curso cubre una porción suficientemente completa sobre las múltiples posibilidades que ofrece *symfony* para desarrollar aplicaciones *web*, incidiendo en sus características y

Plan del curso

herramientas más fundamentales. El objetivo es que, al final del curso, te sientas cómodo usando *symfony* 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 *symfony* 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 crece de una manera abismal.

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 *symfony* 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 *symfony*, como se verá más adelante, la utiliza en su implementación. 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 *symfony*, exponiendo los conceptos fundamentales. En esta unidad volveremos a escribir, esta vez utilizando *symfony*, la aplicación de la unidad 2. Dicho ejercicio nos ayudará a realizar la presentación del *framework* a la vez que servirá como referencia de los conceptos de base. Avisamos: esta unidad es bastante densa.

En la unidad 4 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 un gestor documental y su desarrollo nos servirá como vehículo para penetrar al interior de *symfony* durante el resto del curso.

En las siguientes unidades se construyen progresivamente las distintas funcionalidades de la aplicación analizada en la unidad 4. Cada unidad incide sobre algún aspecto fundamental de *symfony*.

En la unidad 5 profundizamos en el concepto de *modelo* y de capa de abstracción de acceso a datos (*ORM*). Aquí se tratan con bastante profundidad los aspectos relacionados con el acceso a bases datos.

En la unidad 6 se describe el mecanismo para el manejo de la sesión de usuario en el servidor que proporciona *symfony*, mediante el cual se trata la seguridad a los niveles de autentificación y autorización. En esta unidad se construye un procedimiento para que los usuarios inicien sesión en la aplicación que puede ser reutilizado en cualquier otra aplicación *web* construida con *symfony*.

En la unidad 7 se profundiza en la arquitectura *MVC* de *symfony*. Esta unidad ofrece un buen número de detalles, tanto del controlador como de la vista, mediante los cuales se enriquecen las aplicaciones *web* desarrolladas con *symfony*.

En la unidad 8 se presenta el fabuloso *framework* de formularios de *symfony*. Este sistema, por sí solo, constituye una poderosa herramienta para la definición de formularios y la validación de los datos que son enviados en las peticiones *HTTP* al servidor *web*. También se estudian los formularios que son generados automáticamente por *symfony* para la gestión de cada una de las tablas de la base de datos.

Sobre symfony

En la unidad 9 se construye completamente y de manera automática la parte de administración de la aplicación analizada en el tema 4. Además se construye un *plugin* con el fin de reutilizar código.

Por último, la unidad 10 presenta los conceptos de internacionalización y enrutamiento y la manera en que *symfony* los trata.

Sobre symfony

Symfony es uno de los *frameworks* para el desarrollo de aplicaciones *web* en *PHP* más conocidos y utilizados. Prueba de ello es la amplia y activa comunidad que lo desarrolla y que lo utiliza. El número de aplicaciones registradas en el *trac**5 del proyecto también da una idea de la confianza que muchos desarrolladores depositan en **symfony*. Al margen de estos parámetros basados en la amplitud de una comunidad, la fiabilidad del *framework* está ampliamente asegurada gracias a los más de 8500 test unitarios y funcionales que componen la batería de test del producto y que constituyen una garantía de calidad.

Desde la primera versión de *symfony* hace 5 años se han lanzado 5 versiones estables, siendo la última de ellas la versión 1.4, sobre la cual hemos desarrollado este curso. No obstante gran parte de los conceptos que se tratan en el curso son independientes de la versión. Incluso muchos de ellos son comunes a otros *frameworks* de desarrollo. Esto es así por que los creadores de *symfony**6 han partido del famoso principio consistente en "no reinventar la rueda" y han utilizado muchas ideas y patrones que han tenido éxito en el mundo del desarrollo de *software en general, más allá del *PHP*. De hecho muchas ideas provienen del exitoso *framework* de desarrollo de aplicaciones *web* *Ruby On Rails*.

La documentación de symfony

Possiblemente una de las características más apreciadas de *symfony* es 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* se han escrito. A medida que avanzas en el curso comprobarás numerosas referencias a documentos donde se trata con más detalle el concepto que en ese momento se esté trabajando.

Teniendo en cuenta este hecho, hemos desarrollado el texto de este curso desde una perspectiva más pedagógica que técnica, ya que es en aquel aspecto donde la documentación oficial de *symfony* es más endeble. Este curso supone un apoyo pedagógico para aprender a desarrollar aplicaciones *web* con *symfony*, y no pretende ni sustituir ni desdellar la documentación oficial. Muy al contrario creemos que dicha documentación es muy valiosa y debe formar parte del equipo de recursos necesarios para desarrollar con *symfony*. Por ello realizaremos a continuación una relación de dicha documentación:

Documento	Descripción
<i>A Gentle Introduction to symfony</i>	En este manual se tratan todos los aspectos fundamentales y no tan fundamentales sobre <i>symfony</i> . A nuestro modo de ver es más que una introducción.
<i>Practical symfony</i>	Son 24 tutoriales de, supuestamente, 1 hora cada uno. En ellos se desarrolla desde el principio hasta el final una aplicación completa consistente en un sitio para el registro y difusión de ofertas de trabajo. Trata prácticamente todos los aspectos de <i>symfony</i> .

Instalación de symfony

<i>The symfony Reference Book</i>	Es el manual de referencia de <i>symfony</i> . Muy apropiado cuando uno ya se siente cómodo con el <i>framework</i> .
<i>More with symfony</i>	Es un libro solo apto para para “ <i>symfony masters</i> ”. En él se explican recovecos de <i>symfony</i> muy interesantes.
<i>La API</i>	Como es de esperar, representa la documentación más árida pero más útil en ocasiones en las que hay que explorar qué posibilidades ofrece un objeto, o como deben ser los argumentos de una función.

Todos estos documentos se encuentran en el sitio web oficial de *symfony*:

http://www.symfony-project.org/doc/1_4/

Además existen traducciones de muchos de ellos en el mismo sitio de *symfony*. También son interesantes las traducciones sobre *symfony* realizadas en *librosweb*:

<http://www.librosweb.es/symfony/index.html>

Instalación de symfony

La instalación de *symfony* está perfectamente explicada en la siguiente dirección *web*:

http://www.symfony-project.org/getting-started/1_4/en/03-Symfony-Installation

No obstante hemos creado la siguiente guía de instalación por si tuvieses problemas con el inglés.

Instalación en Windows de XAMPP + symfony

- Descargar xampp para Windows (versión autoextraible exe):

<http://www.apachefriends.org/download.php?xampp-win32-1.7.3.zip>

- Ejecutar el archivo y se instala solito. A las preguntas se les responde con la opción que ofrecen por defecto.
- Abrir el control panel e iniciar los servicios apache y mysql. Asegúrate que las casillas Svc de estos servicios están desactivadas.
- Prueba el web server. Abre un navegador y solicita la URL <http://localhost>. Debe responderte el servicio recién instalado. Elige el idioma y verás la página de inicio de XAMPP. Ahora selecciona la herramienta *phpMyAdmin* para comprobar que el servidor de base de datos funciona correctamente.
- Descargar *symfony* (archivo <http://www.symfony-project.org/get/symfony-1.4.6.zip>)
- Extraer el contenido en la carpeta C:\xampp\php
- Añadir a la variable de entorno Path la ruta siguiente: C:\xampp\php\symfony-1.4.6\data\bin. Para ello haz click con el botón derecho del ratón en Mi PC y selecciona Propiedades → Opciones Avanzadas → Variables de entorno, selecciona la variable Path, la editas y añades la ruta anterior.
- Ya puedes usar el comando *symfony* en una terminal desde cualquier directorio.

Instalación de symfony en Ubuntu

Instalación de symfony en Ubuntu

- Instala mediante *synaptic* el servidor *web apache* con *php* y el servidor *mysql* (paquetes *apache2*, *mysql-server* y *phpmyadmin*)
- Prueba los servicios instalados. Para ello abre un navegador y accede a la URL `http://localhost` y `http://localhost/phpmyadmin`
- Descargar `symfony` (archivo zip: <http://www.symfony-project.org/get/symfony-1.4.6.zip>)
- Mover el archivo anterior al directorio `/usr/share/php/`: `sudo mv symfony-1.4.6.zip /usr/share/php`
- Descomprimirlo:
- `sudo unzip symfony-1.4.6.zip`
- Hacer un enlace simbólico en el directorio `/usr/bin` a la herramienta de línea de comandos de *symfony*: `cd /usr/bin sudo ln -s /usr/share/php/symfony-1.4.6/data/bin/symfony`
- Ya puedes usar el comando *symfony* en una terminal desde cualquier directorio.

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 de *patrones de diseño*¹ y cuyo conocimiento y aplicación son de una inestimable ayuda a la hora de diseñar y construir una aplicación informática.

Possiblemente 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:

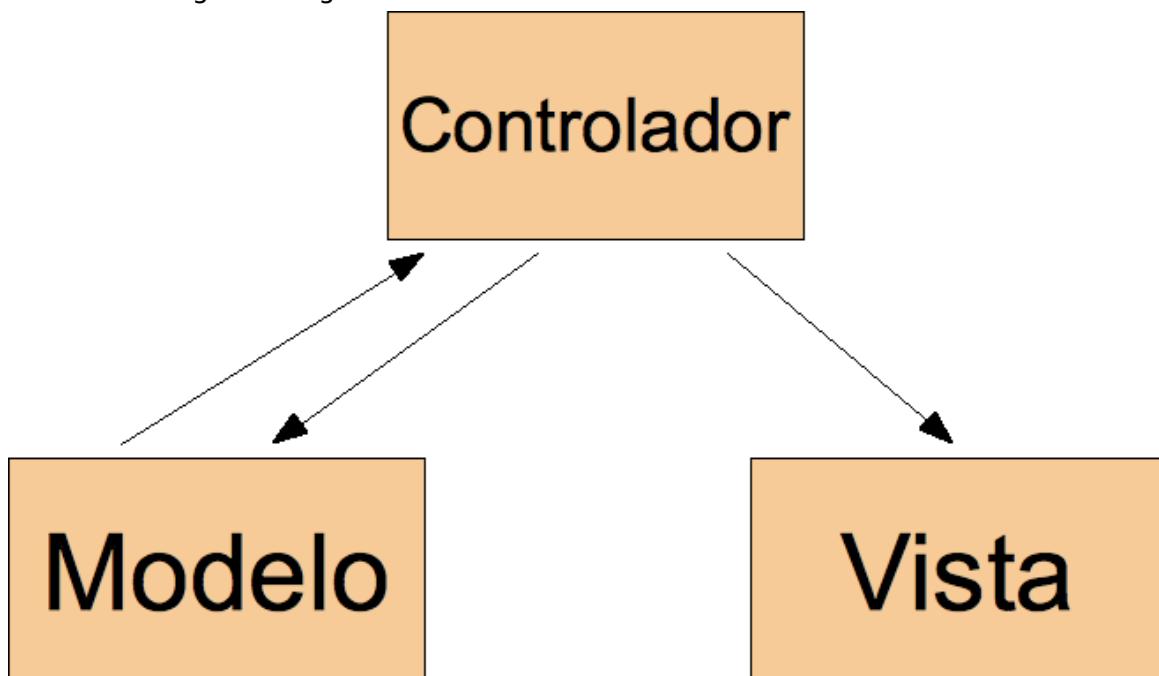


Diagrama del modelo MVC

Descripción de la aplicación

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 áreas de un problema en sistemas débilmente acoplados, es una estrategia común en las metodologías 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 *symfony 1.4*.

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*.

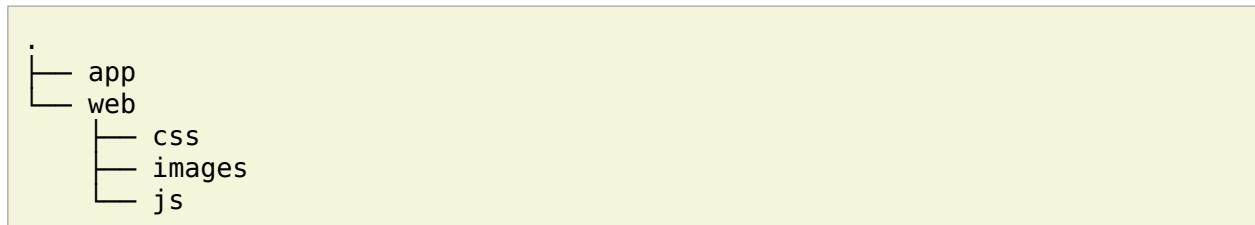
El servidor web únicamente puede acceder a una parte del sistema de ficheros que se denomina *Document Root*. Es ahí donde se buscan los recursos cuando se realiza una petición a la raíz del servidor a través de la URL <http://el.servidor.que.sea/>. Sin

Diseño de la aplicación (II). El controlador frontal

embargo, el código ejecutado para construir dinámicamente la respuesta puede "vivir" en cualquier otra parte, fuera del *Document root*³.

Todo 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:



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, ia 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 solicitada.

Construcción de la aplicación. Vamos al lío.

Nota

La **query string** es la parte de la URL que contiene los datos que se pasarán a la aplicación web. Por ejemplo, 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 las 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`.

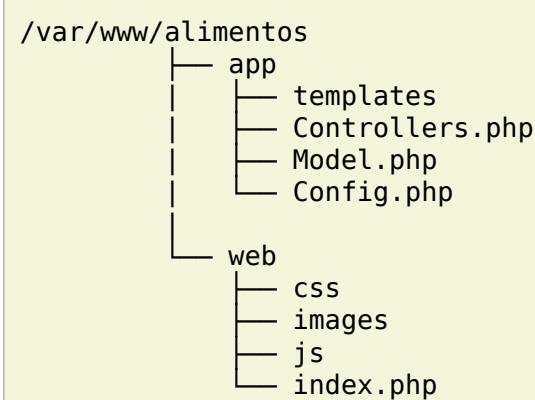
Es importante resaltar que esto no debería hacerse en un entorno de producción, ya que dejamos al servidor web acceder directamente al directorio `app`, y es algo que deseamos evitar. Sin embargo, de esta manera podemos añadir todos los proyectos que queramos sin tener que tocar la configuración del 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:

El controlador frontal y el mapeo de rutas



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
<code>http://tu.servidor/alimentos/index.php?ctl=inicio</code>	mostrar pantalla inicio
<code>http://tu.servidor/alimentos/index.php?ctl=listar</code>	listar alimentos
<code>http://tu.servidor/alimentos/index.php?ctl=insertar</code>	insertar un alimento
<code>http://tu.servidor/alimentos/index.php?ctl=buscar</code>	buscar alimentos
<code>http://tu.servidor/alimentos/index.php?ctl=ver&id=x</code>	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, los 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 aclarar 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.

```
1 <?php
2 // web/index.php
3
4 // carga del modelo y los controladores
```

El controlador frontal y el mapeo de rutas

```
5  require_once __DIR__ . '/../app/Config.php';
6  require_once __DIR__ . '/../app/Model.php';
7  require_once __DIR__ . '/../app/Controller.php';
8
9 // enrutamiento
10 $map = array(
11     'inicio' => array('controller' =>'Controller', 'action' =>'inicio'),
12     'listar' => array('controller' =>'Controller', 'action' =>'listar'),
13     'insertar' => array('controller' =>'Controller', 'action' =>'insertar'),
14     'buscar' => array('controller' =>'Controller', 'action' =>'buscarPorNombre'),
15     'ver' => array('controller' =>'Controller', 'action' =>'ver')
16 );
17
18 // Parseo de la ruta
19 if (isset($_GET['ctl'])) {
20     if (isset($map[$_GET['ctl']])) {
21         $ruta = $_GET['ctl'];
22     } else {
23         header('Status: 404 Not Found');
24         echo '<html><body><h1>Error 404: No existe la ruta <i>' .
25             $_GET['ctl'] .
26             '</i></h1></body></html>';
27         exit;
28     }
29 } else {
30     $ruta = 'inicio';
31 }
32
33 $controlador = $map[$ruta];
34 // Ejecución del controlador asociado a la ruta
35
36 if (method_exists($controlador['controller'],$controlador['action'])) {
37     call_user_func(array(new $controlador['controller'], $controlador['action']));
38 } else {
39
40     header('Status: 404 Not Found');
41     echo '<html><body><h1>Error 404: El controlador <i>' .
42             $controlador['controller'] .
43             '->' .
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.

Las acciones del Controlador. La clase Controller.

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 término 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 *symfony 1.4*.

Las acciones del Controlador. La clase Controller.

Ahora vamos a implementar las acciones asociadas a las *URL*'s en la clase Controllers. 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(  
9              'mensaje' => 'Bienvenido al curso de symfony 1.4',  
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     {
```

Las acciones del Controlador. La clase Controller.

```
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,
```

Las acciones del Controlador. La clase Controller.

```
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 llenos con los valores que se enviaron en la petición anterior y con un mensaje de error.

La implementación de la Vista.

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 lleno 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 representan a los 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 (dedicale 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; ?>
```

El layout y el proceso de decoración de plantillas

```
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 carácter *:*. 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 Decorador⁴. 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 *symfony 1.4*.

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>
```

El layout y el proceso de decoración de plantillas

```
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. Créalos 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 energía</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 está 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

El layout y el proceso de decoración de plantillas

\$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 recogen 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 sola 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>Proteína (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' ?>
```

El layout y el proceso de decoración de plantillas

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>Proteína</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>
```

El Modelo. Accediendo a la base de datos

```
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)
8      {
9          $mvc_bd_conexion = mysql_connect($dbhost, $dbuser, $dbpass);
10
11         if (!$mvc_bd_conexion) {
12             die('No ha sido posible realizar la conexión con la base de datos: ' . mysql_error());
13         }
14         mysql_select_db($dbname, $mvc_bd_conexion);
15
16         mysql_set_charset('utf8');
17
18         $this->conexion = $mvc_bd_conexion;
19     }
20
21
22
23     public function bdConexion()
24     {
25
26     }
27
28     public function dameAlimentos()
29
30     {
31         $sql = "select * from alimentos order by energia desc";
32
33         $result = mysql_query($sql, $this->conexion);
```

El Modelo. Accediendo a la base de datos

```
34     $alimentos = array();
35     while ($row = mysql_fetch_assoc($result))
36     {
37         $alimentos[] = $row;
38     }
39
40     return $alimentos;
41 }
42
43 public function buscarAlimentosPorNombre($nombre)
44 {
45     $nombre = htmlspecialchars($nombre);
46
47     $sql = "select * from alimentos where nombre like '" . $nombre . "' order by energia desc";
48
49     $result = mysql_query($sql, $this->conexion);
50
51     $alimentos = array();
52     while ($row = mysql_fetch_assoc($result))
53     {
54         $alimentos[] = $row;
55     }
56
57     return $alimentos;
58 }
59
60 public function dameAlimento($id)
61 {
62     $id = htmlspecialchars($id);
63
64     $sql = "select * from alimentos where id=". $id;
65
66     $result = mysql_query($sql, $this->conexion);
67
68     $alimentos = array();
69     $row = mysql_fetch_assoc($result);
70
71     return $row;
72 }
73
74
75 public function insertarAlimento($n, $e, $p, $hc, $f, $g)
76 {
77     $n = htmlspecialchars($n);
78     $e = htmlspecialchars($e);
79     $p = htmlspecialchars($p);
80     $hc = htmlspecialchars($hc);
81     $f = htmlspecialchars($f);
82     $g = htmlspecialchars($g);
83
84     $sql = "insert into alimentos (nombre, energia, proteina, hidratocarbono, fibra, grasatotal) values ('" .
85             $n . "','" . $e . "','" . $p . "','" . $hc . "','" . $f . "','" . $g . "')";
86
87     $result = mysql_query($sql, $this->conexion);
88
89     return $result;
90 }
91
92 public function validarDatos($n, $e, $p, $hc, $f, $g)
93 {
94     return (is_string($n) &
95             is_numeric($e) &
96             is_numeric($p) &
97             is_numeric($hc) &
98             is_numeric($f) &
99             is_numeric($g));
100 }
101
102 }
```

Cuando el controlador requiere el uso del modelo, creamos un objeto de tipo `$m = new Model()`. El constructor de esta clase realiza una conexión con la base de datos y la pone disponible a todos sus métodos al añadir la conexión creada como atributo del objeto. Cada función utiliza esta conexión para realizar su cometido contra la base de datos.

La última función de la clase, `validarDatos()`, es algo distinta, ya que no utiliza para nada la conexión con la base de datos. Simplemente valida datos. Si la aplicación fuera más compleja sería interesante crear una clase dedicada a la validación. De manera que atendamos al principio de la *Separation of Concerns*.

La configuración de la aplicación

A lo largo y ancho de todo el código expuesto, aparece cada tanto una referencia a unos atributos estáticos de la clase `Config`. Por ejemplo, en la línea 10 del archivo `app/Model.php`, aparece `Config::$mvc_bd_hostname`, `Config::$mvc_bd_usuario`, etcétera. Se trata de parámetros de configuración que hemos definido en una clase denominada `Config`:

`app/Config.php`

```
1 <?php
2
3 class Config
4 {
5     static public $mvc_bd_hostname = "localhost";
6     static public $mvc_bd_nombre = "alimentos";
7     static public $mvc_bd_usuario = "root";
8     static public $mvc_bd_clave = "root";
9     static public $mvc_vis_css = "estilo.css";
10 }
```

Esta clase está disponible durante todo el script de manera que se pueden utilizar sus valores a lo largo del código, y cambiarlos sin más que modificar este fichero.

Con esto ya tenemos todo el código de la parte de servidor. Ya sólo nos falta darle un toque de estilo a los documentos *HTML* que enviamos al cliente y crear la base de datos que almacenará los datos persistentes sobre los alimentos.

Incorporar las CSS's

Crea el directorio `web/css` y en él coloca un archivo llamado `estilo.css` con el siguiente contenido:

`web/css/estilo.css`

```
body {
    padding-left: 11em;
    font-family: Georgia, "Times New Roman",
    Times, serif;
    color: purple;
    background-color: #d8da3d }

ul.navbar {
    list-style-type: none;
    padding: 0;
    margin: 0;
    position: absolute;
    top: 2em;
    left: 1em;
    width: 9em }

h1 {
    font-family: Helvetica, Geneva, Arial,
    SunSans-Regular, sans-serif }

ul.navbar li {
    background: white;
    margin: 0.5em 0;
    padding: 0.3em;
```

La base de datos

```
border-right: 1em solid black }
ul.navbar a {
  text-decoration: none }
a:link {
  color: blue }
a:visited {
  color: purple }
address {
  margin-top: 1em;
  padding-top: 1em;
  border-top: thin dotted }
#contenido {
  display: block;
  margin: auto;
  width: auto;
  min-height:400px;
}
```

Fíjate que en el archivo app/templates/layout.php se incluye (línea 6) este archivo CSS que acabamos de crear. Como dicho *layout* decora a todas las plantillas, estos estilos afectarán a todas las páginas.

La base de datos

En el Sistema Gestor de Base de Datos MySQL que vayas a utilizar, utilizando algún cliente MySQL crea una base de datos para almacenar los alimentos. Introduce algunos registros para probar la aplicación.

```
CREATE TABLE `alimentos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nombre` varchar(255) NOT NULL,
  `energia` decimal(10,0) NOT NULL,
  `proteina` decimal(10,0) NOT NULL,
  `hidratocarbono` decimal(10,0) NOT NULL,
  `fibra` decimal(10,0) NOT NULL,
  `grasatotal` decimal(10,0) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Lo importante para que la conexión funcione, es que los parámetros de conexión que se establecen en el fichero app/Config.php coincidan con los de tu base de datos.

Sugerencia

Lo más cómodo para desarrollar es tener todo en el mismo computador: tanto el servidor web (*apache*) como el servidor de base de datos (*MySQL*). Además es muy práctico, aunque nada seguro, utilizar en el servidor MySQL el usuario root (superadministrador) como usuario de nuestras aplicaciones. Así no hay que preocuparse por temas de permisos. Repetimos: es lo más cómodo pero, a la vez, lo más inseguro. Esta práctica está justificada ÚNICAMENTE en un entorno de desarrollo en local, donde la seguridad, en principio no es primordial.

La base de datos

Ya puedes juntar todas las piezas y probar la aplicación introduciendo en tu navegador la *URL* correspondiente:

`http://tu.servidor/ruta/a/alimentos/web/index.php`

¡Suerte!

Unidad 3. Desarrollo de una aplicación con Symfony

El objetivo fundamental de este curso es que aprendas a desarrollar aplicaciones *web* de calidad con *symfony*, un potente *framework* de desarrollo en *PHP*. Para conocerlo en profundidad utilizaremos una estrategia de acercamiento en espiral a nuestro objeto de estudio, es decir, lo rodearemos trazando círculos de radio cada vez más pequeños, de forma que cada vuelta que demos nos revelará un conocimiento más profundo de esta magnífica herramienta de desarrollo.

Esta estrategia nos proporcionará resultados visibles y prácticos desde el principio, desde la primera vuelta, aunque ello será a costa de tratar superficialmente algunos conceptos que, progresivamente, irán definiéndose con más precisión a medida que avanza el curso.

En esta unidad volveremos a desarrollar la aplicación que construimos en la unidad 2. Pero ahora utilizaremos *symfony*. Será nuestra primera vuelta, la menos precisa pero la más panorámica, que nos proporcionará una primera imagen del *framework* aún difusa pero suficiente para mostrar sus características fundamentales.

Nota

Recursos de apoyo y ampliación
http://www.librosweb.es/symfony_1_2/capitulo3/crear_una_aplicacion_web.html
http://www.librosweb.es/symfony_1_2/capitulo4.html

Creamos el proyecto.

El desarrollo de cualquier aplicación *web* desarrollada con *symfony* comienza por la generación del **proyecto** que es la columna vertebral que articula y da cohesión a todos los elementos que conforman la aplicación: las librerías del modelo, el código *PHP* de los módulos, los archivos de configuración, las *CSS*'s, las librerías *Javascript*, las imágenes y otros recursos que pueda requerir la aplicación, incluido la documentación.

Al igual que hicimos en el ejercicio de la unidad 2, crearemos un directorio en un lugar accesible por el servidor *web*, donde desplegaremos los ficheros del proyecto:

```
$ mkdir /opt/lamp/htdocs/unidad3
```

Symfony proporciona una herramienta de desarrollo que nos asiste a lo largo del desarrollo de nuestra aplicación mediante las denominadas **tareas**. Dicha herramienta está desarrollada en *PHP* y se utiliza a través de la linea de comandos (*PHP-CLI*). Su nombre, como es de esperar, es **symfony**. Podemos ver un listado de todas las tareas que ofrece esta herramienta, así como su modo de uso invocándola sin ningún argumento:

```
$ symfony
```

El primer uso que haremos de la misma será la generación del proyecto, para lo que nos ubicamos dentro del directorio que acabamos de crear y ejecutamos la siguiente instrucción:

```
$ symfony generate:project unidad3 --orm=Propel2
```

la cual genera un conjunto de ficheros organizados funcionalmente en un árbol de directorios que denominaremos: **el proyecto**. A medida que desarrollemos la aplicación, ubicaremos

Unidad 3. Desarrollo de una aplicación con Symfony

los archivos en el directorio que le corresponda según su función, de la misma manera que ya hicimos en el mini-framework de la unidad 2.

Aunque el número de directorios ha crecido sustancialmente y pueda resultar abrumador en un principio, verás como al poco de trabajar con *symfony* te manejarás con bastante soltura por la estructura recién creada, y comprobarás que las posibilidades de *symfony* superan extraordinariamente las de mini-framework de la unidad anterior.

Desde este momento hasta el final del curso, los conceptos *controlador frontal*, *acción*, *plantilla* y *layout* introducidos en la unidad 2 y derivados del patrón *Modelo - Vista - Controlador*, son utilizados intensivamente. Por ello es importante que los tengas bien asimilados antes de continuar.

Vamos a describir la función de cada directorio del proyecto:

Directorio	Función
<i>apps</i>	Bajo este directorio se alojarán los ficheros con las acciones y las plantillas agrupados en módulos que a su vez se agrupan en aplicaciones . La mayor parte de los ficheros desarrollados manualmente por el programador, es decir el código específico de la aplicación, se alojan aquí. Por lo pronto está vacío pues aún no hemos comenzado a programar nada.
<i>lib</i>	Aquí se guardarán todas las clases, generadas automáticamente por <i>symfony</i> , mediante las cuales podremos gestionar las tablas de la base de datos con independencia del sistema gestor de base de datos que utilicemos. Estas clases constituyen un <i>ORM (Object Relational Mapping)</i> , es decir una capa de abstracción entre la base de datos y la aplicación que permite al programador utilizar siempre los mismos métodos (funciones de objetos) para manipular los datos sin que éste se tenga que preocupar por la sintaxis propia de la base de datos que utilice. <i>Symfony</i> incorpora dos <i>ORM's</i> bien conocidos: <i>Propel</i> y <i>Doctrine</i> , en este curso utilizaremos el primero de ellos. En la terminología de <i>symfony</i> al conjunto formado por todas estas clases se les denomina el modelo . Además <i>symfony</i> proporciona un <i>subframework</i> independiente para la gestión de formularios <i>HTML</i> que puede ser utilizado en otros proyectos que no sean <i>symfony</i> . Si decidimos utilizarlo en nuestro proyecto, lo cual recomendamos encarecidamente, también se alojarán en este directorio un conjunto de clases que modelan formularios HTML para la manipulación de las tablas de la base de datos. Estas clases son generadas automáticamente a partir de la estructura de la base de datos. Por último también se alojarán los filtros , que son un tipo especial de formulario que se utiliza para realizar búsquedas acotadas sobre los datos de las tablas de la base de datos.

Unidad 3. Desarrollo de una aplicación con Symfony

<i>config</i>	Aquí se almacenan los ficheros de configuración del proyecto: <i>ProjectConfiguration.class.php</i> , en él se realiza la inclusión del núcleo de <i>symfony</i> mediante la clase <i>sfCoreAutoload</i> que se encarga de realizar la inclusión automática de las clases requeridas en nuestras acciones. <i>databases.yml</i> , en este archivo se establecen los parámetros de conexión para cada base de datos que utilice el proyecto: <i>host</i> , nombre de usuario, clave y nombre de la base de datos. <i>schema.yml</i> , es un archivo en el que se define la estructura de la base de datos que vamos a utilizar. Por cada base de datos que requiera el proyecto tendremos un esquema que se denominará: <i>basedatos1.schema.yml</i> , <i>basedatos2.schema.yml</i> , etcétera. Lo más normal es utilizar sólo una base de datos, se utilizará entonces tan solo un archivo denominado <i>schema.yml</i> . Este (estos) archivo(s) es (son) el punto de partida para la generación tanto de las clases del modelo como de los formularios y los filtros. Son archivos que sólo se necesitan durante la fase de construcción del software. <i>propel.ini</i> , es el archivo de configuración de <i>Propel</i> , y al igual que el <i>schema.yml</i> , únicamente se necesita durante la fase de construcción de la aplicación.
---------------	---

Unidad 3. Desarrollo de una aplicación con Symfony

web	<p>En este directorio residen los recursos que el servidor web puede suministrar como respuestas a las peticiones <i>HTTP</i>. En un entorno de producción debería coincidir con el directorio raíz del servidor web. Es importante, por cuestiones de seguridad, que una vez desplegada la aplicación en el servidor de producción, nos aseguremos que el servidor web únicamente tiene acceso a este directorio, es decir, que es imposible obtener un archivo del resto de los directorios a través de una URL. Para conseguir esto hay que configurar adecuadamente el servidor web. En esta URL del proyecto <i>symfony</i> se explica con detalle como hacerlo:</p> <p>http://www.symfony-project.org/getting-started/1_4/en/05-Web-Server-Configuration</p> <p>No obstante, en los entornos de desarrollo este hecho no es tan importante, por ello en el desarrollo de nuestro ejercicio no lo tendremos en cuenta y desplegaremos todos los ejercicios bajo el directorio raíz del servidor web que tengamos instalado. Dentro del directorio web encontramos: Por cada aplicación (en el sentido de agrupación de módulos que le da <i>symfony</i>), tenemos dos archivos denominados controladores frontales, cuya finalidad es la misma que la del controlador frontal del mini- framework de la unidad 2, aunque su lógica es mucho más compleja. Si, por ejemplo tenemos dos aplicaciones denominadas <i>aplicacion1</i> y <i>aplicacion2</i> respectivamente, a la primera le corresponderán los controladores frontales:</p> <p style="padding-left: 40px;"><i>index.php</i> y <i>aplicacion1_dev.php</i></p> <p>Mientras que a la segunda:</p> <p style="padding-left: 40px;"><i>aplicacion2.php</i> y <i>aplicacion2_dev.php</i></p> <p>Estos archivos son el único punto de entrada a cada aplicación y ejecutarán la acción del módulo que se le indique mediante parámetros <i>GET</i> en la URL. El archivo que no lleva el sufijo _dev, es el controlador de producción mientras que el que lo lleva es el de desarrollo. Obviamente, sólo podemos utilizar uno de los controladores frontales. La diferencia más importante entre ambos es que el controlador de desarrollo incorpora un sistema de depuración (<i>debug</i>) que ofrece al programador muchísima información sobre la ejecución de las acciones (variables de sesión, <i>request</i>, respuesta, tiempo de ejecución, <i>queries SQL</i> lanzadas, y muchas más cosas). Es por ello muy importante que en el servidor de producción no se pueda acceder a la aplicación a través del controlador frontal de desarrollo. También tenemos un directorio para el almacenamiento de los css's denominado <i>css</i>. Otro denominado <i>js</i> para el almacenamiento de las librerías de funciones <i>javascript</i>. Otro denominado <i>images</i> para el almacenamiento de las imágenes. Otro denominado <i>uploads</i> para el almacenamiento de los ficheros subidos al servidor desde los clientes.</p>
-----	--

Generación del esquema, el modelo y los formularios

<i>plugins</i>	Las aplicaciones construidas con <i>symfony</i> pueden ser reorganizadas en plugins que tienen la particularidad de distribuirse mediante canales de <i>PEAR</i> y que pueden ser incorporados fácilmente a nuestras aplicaciones <i>symfony</i> ampliando sus funcionalidades. Es en este directorio donde se alojarán los <i>plugins</i> , si decidimos utilizar alguno. Existe un sitio web del proyecto <i>symfony</i> dedicado a los <i>plugins</i> . Allí podemos encontrar varios cientos de ellos. Merece la pena destacar el <i>plugin</i> de gestión de usuarios <i>sfGuardPlugin</i> , mediante el que podemos incorporar en poco tiempo un sistema de gestión de usuarios y grupos con autenticación y autorización.
<i>doc</i>	Para almacenar la documentación
<i>data</i>	Para almacenar datos externos a la propia aplicación como pueden ser <i>scripts</i> para poblar la base de datos
<i>test</i>	Aquí se generarán de forma semiautomática tests para realizar pruebas unitarias.
<i>cache</i>	Es el directorio utilizado por el sistema de caché de <i>symfony</i> para optimizar el tiempo de ejecución.
<i>log</i>	Aquí se alojan los archivos de registro para el diagnóstico y control de las aplicaciones.

Como podemos observar, es “algo” más complejo que el mini-framework que desarrollamos en la unidad 2. Pero el esfuerzo requerido para asimilarlo merecerá la pena. Tranquilos, aún no hemos dado la primera vuelta de la espiral.

Generación del esquema, el modelo y los formularios

En este segundo paso generaremos el resto de la estructura básica que dará soporte a la aplicación que vamos a construir: las clases con las que accederemos a la base de datos y los formularios que el programador podrá utilizar para gestionar las tablas de dicha base de datos. Por ello, para continuar necesitamos confeccionar previamente una base de datos. El ejercicio que se desarrolla en esta unidad, como ya hemos advertido al principio, es el mismo que el de la unidad anterior, por tanto utilizaremos la misma base de datos: *alimentos*.

En primer lugar definimos los parámetros de conexión a la base de datos para que la aplicación tenga acceso a la misma. Esto se hace a través del fichero *config/databases.yml*:

```

dev:
  propel:
    param:
      classname: DebugPDO
      debug:
        realmemoryusage: true
        details:
          time: { enabled: true }
          slow: { enabled: true, threshold: 0.1 }
          mem: { enabled: true }
          mempeak: { enabled: true }
          memdelta: { enabled: true }

test:

```

```
propel:  
    param:  
        classname: DebugPDO  
  
all:  
    propel:  
        class: sfPropelDatabase  
        param:  
            classname: PropelPDO  
            dsn: mysql:dbname=alimentos;host=localhost  
            username: root  
            password: root  
            encoding: utf8  
            persistent: true  
            pooling: true
```

Nota

Antes de continuar abrimos un pequeño paréntesis para adelantar que la mayor parte de los ficheros de configuración de *symfony* utilizan el formato *YAML* para definir los parámetros de configuración. Según dicen en el sitio oficial del proyecto, “*YAML is a human friendly data serialization standard for all programming languages*”, es decir un standard para la serialización de datos fácil de leer por humanos y máquinas. Algo así como el *XML* pero más fácil de interpretar por las personas. Los archivos *YAML* utilizan la identación de los elementos como base para definir estructuras jerárquicas (árboles). Esto es algo que puedes comprobar directamente echando un vistazo al fichero anterior.

Los ficheros de configuración de *symfony* suelen clasificar los parámetros de configuración en varias secciones, cada una de ellas corresponde a lo que se denomina un entorno de ejecución. Son 3 los entornos de ejecución que el *framework* ofrece por defecto, aunque podemos añadir otros nuevos. Se conocen como desarrollo (cuya clave es *dev*), producción (cuya clave es *prod*) y pruebas (cuya clave es *test*). Esta estrategia permite definir distintos valores para el mismo parámetro en cada entorno de ejecución. Si deseamos definir un parámetro de configuración común para todos los entornos utilizamos la clave *all*. El archivo anterior muestra este hecho con claridad, aunque no especifica parámetros propios para el entorno de producción, con lo que este entorno asumiría únicamente los parámetros definidos en la sección de clave *all*.

Y ahora te preguntarás, ¿y que cosa es esto de los entornos de ejecución? Podemos decir brevemente que cada entorno de ejecución responde a distintas configuraciones. Por ejemplo, mientras en el entorno de desarrollo tenemos acceso a una gran cantidad de datos para la depuración de la aplicación, en el de producción no. Así mismo podríamos definir una base de datos para el entorno de producción y otra para el de desarrollo. Y la otra pregunta que surge ¿cómo ejecutamos un entorno u otro? La respuesta ya la hemos adelantado cuando explicábamos la existencia de dos controladores frontales por aplicación, uno para desarrollo y otro para producción. En ellos se especifica qué entorno deseamos utilizar en la ejecución del *script*.

En el siguiente apartado volveremos a la carga con el tema de los entornos de ejecución. Por lo pronto vamos a cerrar el paréntesis con la intención de no perder demasiado el hilo.

Generación del esquema, el modelo y los formularios

En cada entorno de ejecución se ha definido una conexión denominada *propel*. La conexión consiste en una serie de parámetros que el *framework* utilizará para conectarse a la base de datos. Los más importantes son:

- *dsn*: que define el nombre de la base de datos y el host, lo que se conoce generalmente como *Data Source Name*.
- *username*: el nombre de usuario de la base de datos
- *password*: el *password* de dicho usuario

Si el proyecto va a requerir el acceso a varias bases de datos, es aquí donde debemos añadir las conexiones con los parámetros para acceder a las mismas. El nombre de las conexiones lo decide el programador, aunque por defecto la primera conexión creada al generarse el proyecto se denomina *propel*, como acabamos de ver.

A continuación modificamos el fichero de configuración *config/propel.ini*, el cual es utilizado por *symfony* para la generación del esquema a partir de una base de datos existente. Lo que se denomina una introspección a la base de datos. En él indicamos los parámetros de conexión a la base de datos en cuestión: (líneas 6 a 9 del fichero *config/propel.ini*)

```
...
propel.database.url      = mysql:dbname=sf_alimentos;host=localhost
propel.database.creole.url = ${propel.database.url}
propel.database.user     = root
propel.database.password = root
...
```

El archivo que acabamos de modificar contiene otras muchas directivas que indican a *Propel* como debe comportarse. Por lo general podemos dejarlo tal y como está, únicamente se deben cambiar los parámetros de conexión a la base de datos. Este fichero es necesario únicamente durante la fase de desarrollo para la generación automática del **esquema**. No es necesario para la ejecución de la aplicación *web* una vez desarrollada, y se puede obviar una vez que se encuentre en producción.

Comenzamos por generar el esquema:

```
$ symfony propel:build-schema
```

Si abres y examinas el archivo generado (*config/schema.yml*) comprobarás que su contenido es una definición de la estructura de la base de datos al estilo de un *DDL* (*Data Definition Language*). Es decir, en él se definen las tablas, sus columnas y sus relaciones. De cara al modelo de datos, este archivo es el más fundamental del *framework*; la generación del modelo, los formularios y los filtros se basan en él.

Nota

Hemos generado automáticamente el archivo *schema.yml* a partir de una base de datos existente, es decir, realizando ingeniería inversa o una introspección a la base de datos. Pero también se puede proceder al revés: en primer lugar confeccionar manualmente fichero *schema.yml*, y a partir de él generar las sentencias *SQL* para construir la base de datos mediante la instrucción:

```
$ symfony propel:build-sql
```

En este caso se puede prescindir del archivo de configuración *propel.ini*. Sin embargo, pensamos que es más cómodo comenzar por una base de datos ya creada, puesto que existen muchas herramientas visuales que asisten y facilitan la creación de la misma y dicha tarea resulta más cómoda que escribir manualmente el esquema.

Ahora generamos el modelo, los formularios y los filtros:

```
$ symfony propel:build-model  
$ symfony propel:build-forms  
$ symfony propel:build-filters
```

Bajo el directorio *lib* del proyecto encontramos los nuevos directorios *model*, *form* y *filter*. En ellos se encuentran ficheros cuyos nombres hacen referencia a las tablas de la base de datos, y que contienen clases que utilizaremos en el desarrollo de la aplicación para manipular los datos de dichas tablas.

Las aplicaciones y los módulos

El *framework* puede ser imaginado como un inmenso puzzle al que le faltan sus piezas centrales para completarlo y poder contemplarlo en su totalidad. Dependiendo del contenido de las piezas que faltan, el resultado final tendrá un aspecto u otro. Podemos pensar que el puzzle nos ofrece un paisaje natural con el cielo azul, unas montañas al fondo, un fértil valle con su río y sus árboles frutales. El hueco central se puede llenar con cualquier escena que imaginemos: animales pastando, amigos pasando una tarde de campo, o cualquier otra cosa. Lo que acabamos de describir sería un *framework* para la construcción de paisajes bucólicos, en analogía con *symfony* que es un *framework* para la construcción de aplicaciones *web*. La tarea del programador es, precisamente, colocar esas piezas que faltan, las cuales representan las características propias de la aplicación que se desea construir. El resto de elementos comunes que tienen todas las aplicaciones (la gestión de las sesiones, la gestión de las peticiones *HTTP*, la construcción de la respuesta *HTTP*, la validación de los formularios, la gestión de la seguridad, y otras muchas más) lo ofrece el *framework*, igual que ese puzzle incompleto proporciona el marco que alojará la obra completa. El *framework* es como un escenario común donde pueden tener lugar distintas escenas (las aplicaciones).

La parte que falta y cuya elaboración corresponde al programador, comprende las acciones y las plantillas que “dibujan” los datos procesados por aquellas. Recordemos que en el mini-*framework* de la unidad 2, las acciones se implementaban en ficheros (una acción por fichero) y se alojaban en un único directorio denominado *acciones*. Es obvio que a medida que crezca la aplicación el directorio de acciones se saturará de archivos dificultando su mantenibilidad.

Para evitar este hecho, *symfony* utiliza una estrategia de agrupación de las acciones en conjuntos denominados **módulos**, los cuales, a su vez se agrupan en conjuntos denominados **aplicaciones**. Resumiendo:

- un proyecto de *symfony* se “acopla” a una o varias bases de datos, y está compuesto por un número determinado de aplicaciones que, a su vez están compuestas por módulos con acciones.

Las aplicaciones y los módulos

Cómo realizar dicha agrupación de acciones en módulos y aplicaciones es una tarea de diseño que corresponde al programador. Esta es la única decisión de diseño arquitectónico que debemos realizar, ya que el resto las impone el *framework*.

Veamos qué forma física adoptan estos módulos y aplicaciones mediante la práctica. La aplicación que desarrollamos en la unidad anterior y que volvemos a desarrollar ahora con *symfony*, comprendía 8 acciones que manipulaban los datos de una tabla con información dietética sobre alimentos. Se trata de un número de acciones bastante manejable para que sean tratadas dentro de un mismo conjunto. Por ello las agruparemos dentro de un único módulo. A su vez este módulo debemos crearlo dentro de una aplicación, ya que *symfony* exige que, al menos, exista una aplicación para alojar módulos.

Creamos la aplicación:

```
$ symfony generate:app dietetica
```

Con lo cual hemos creado una aplicación denominada *dietetica*, que en términos de ficheros se ha traducido en la creación de un directorio llamado *dietetica* que cuelga del directorio *apps* del proyecto. A continuación se describen los directorios y archivos generados con la aplicación *dietetica*.

Las aplicaciones y los módulos

<i>config</i>	<p>Es un directorio donde se alojan los archivos de configuración particulares de esta aplicación. Observa que prácticamente todos son ficheros YAML.</p> <p><i>app.yml</i>: aquí el programador puede definir los parámetros de configuración que haya decidido utilizar en el desarrollo de la aplicación.</p> <p><i>cache.yml</i>: donde se definen los parámetros que controlan el comportamiento del sistema de caché de las vistas. Por defecto está deshabilitado.</p> <p><i>factories.yml</i>: aquí podemos reemplazar los objetos claves que utiliza el <i>framework</i> para llevar a cabo su trabajo por otros que implementen la misma interfaz y que, obviamente, realicen adecuadamente su cometido, aunque de manera distinta a los originales. Como puedes comprender, tocar este archivo exige un amplio conocimiento del <i>framework</i>, a parte de una importante justificación, ya que estamos hablando de modificaciones a nivel del núcleo.</p> <p><i>filters.yml</i>: Más adelante veremos que <i>symfony</i> implementa para su ejecución un patrón de diseño denominado cadena de responsabilidades, en el cual una serie de objetos van operando secuencialmente sobre el resultado del objeto que le precede en la cadena. Este fichero permite que el programador incorpore nuevos objetos (filtros) a dicha cadena.</p> <p><i>routing.yml</i>: Que es la base del sistema de enrutamiento de <i>symfony</i> gracias al cual las <i>URL</i> de los proyectos <i>symfony</i> presentan un aspecto más estilizado evitando el uso de los caracteres "&", "=" y "?" y haciéndolas más cortas gracias a la supresión del nombre de algunos parámetros. La última unidad de este curso trata este tema con más detalle.</p> <p><i>security.yml</i>: Donde se definen las políticas de seguridad para el acceso a las acciones. La unidad 6 desarrolla este tema con detalle.</p> <p><i>settings.yml</i>: Aquí se definen los principales parámetros de configuración de la aplicación, tales como la cultura por defecto, el máximo tiempo de respuesta para una petición, los módulos habilitados, etcétera. A medida que avances en el curso se te revelarán el significado de los parámetros más usados de este fichero.</p> <p><i>frontendConfiguration.class.php</i>: El método <i>configure</i> de esta clase se ejecuta cuando la aplicación se carga. Se puede utilizar para realizar operaciones de inicialización, aunque en la mayoría de aplicaciones se quedará tal y como está.</p>
<i>i18n</i>	Aquí se guardarán los archivos con las traducciones de los textos a otros idiomas si deseamos internacionalizar la aplicación.
<i>lib</i>	Contiene las clases y librerías externas utilizadas exclusivamente en los módulos de la aplicación. Las clases que aquí se encuentren serán cargadas automáticamente por el <i>framework</i> cuando la acción las necesite, liberando al programador de tener que incluirla manualmente mediante las instrucciones <i>include</i> o <i>require</i> de PHP. Es lo que se denomina autocarga.
<i>modules</i>	En este directorio se alojarán los módulos de la aplicación, que no son más que directorios cuyo nombre es el del módulo.

Las aplicaciones y los módulos

templates	En este directorio se encuentran los <i>layouts</i> con que serán decoradas las plantillas que “dibujan” los resultados de las acciones de la aplicación. Recuerda los conceptos de <i>layout</i> y plantilla que se estudiaron en la unidad anterior, especialmente recuerda el gráfico que ilustraba como la combinación de <i>layout</i> + plantilla da lugar al documento final.
-----------	--

Pero además, bajo el directorio web del proyecto se han generado dos archivos PHP: *index.php* y *dietetica_dev.php*, que son los **controladores frontales de la aplicación**, es decir los puntos de entrada que deben ser invocados desde el navegador web a través de la URL para ejecutar la aplicación.

nos de nuevo al tema de los entornos. Veamos el contenido de cada uno de los controladores frontales de la aplicación:

index.php:

```
require_once(dirname(__FILE__).'../../config/ProjectConfiguration.class.php');
$configuration = ProjectConfiguration::getApplicationConfiguration('dietetica', 'prod', false);
$context->createInstance($configuration)->dispatch();
```

dietetica_dev.php

```
if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1')))

>You are not allowed to access this file. Check '.basename(__FILE__).' for more information.';

require_once(dirname(__FILE__).'../../config/ProjectConfiguration.class.php');
$configuration = ProjectConfiguration::getApplicationConfiguration('dietetica', 'dev', true);
$context->createInstance($configuration)->dispatch();
```

Las primeras líneas del controlador de desarrollo implementan un control de seguridad que garantiza que dicho archivo solo se ejecutará únicamente si la petición HTTP se hace desde el propio computador que aloja la aplicación, que es la situación típica de desarrollo. De esta manera se evita que, por accidente, en un entorno de producción se haya “colado” este controlador y que los usuarios lo ejecuten obteniendo una cantidad de datos sobre la aplicación, lo cual comprometería seriamente su seguridad. A parte de este detalle el resto del código es el mismo, siendo la única diferencia el valor de los parámetros que se pasan a la función *ProjectConfiguration::getApplicationConfiguration*.

Este primer parámetro define la función *ProjectConfiguration::getApplicationConfiguration* define la aplicación que se ejecutarán el controlador frontal, el segundo define el entorno; producción (*prod*), desarrollo (*dev*) o pruebas (*test*), que se utilizará para la configuración de la aplicación en sí misma y con el tercero se decide si se va a utilizar el modo de depuración o no.

no. Es decir, es en el controlador donde se define qué configuración (entorno de ejecución) se utilizará en la ejecución del *script*. De forma que cuando desarrollemos utilizaremos para comprobar el resultado de nuestras líneas de código el controlador de desarrollo, mientras que cuando la aplicación este finalizada y lista para ser desplegada en el servidor de producción, se utilizará el controlador de producción.

Esperamos que con esto quede más claro el sistema de configuración de *symfony* basado en los entornos de ejecución y el uso de ficheros *YAML*. No obstante, si quieres estudiar más a fondo este tema puedes consultar la siguiente URL:

```
http://www.symfony-project.org/gentle-introduction/1_4/en/05-Configuring-Symfony
```

o en castellano:

```
http://www.librosweb.es/symfony_1_2/capitulo5.html
```

Como acabamos de decir, ambos ficheros controlan el acceso y la ejecución de la aplicación, de manera que para ejecutar una acción determinada hay que solicitarlo en la petición *HTTP* indicando mediante parámetros *GET* dicha acción y el módulo al que pertenece. Obviamente, sólo podemos utilizar uno de ellos, de manera que:

```
http://nonbredelservidor/index.php/nombre_modulo/nombre_accion
```

devolvería el resultado de la acción *nombre_accion* del módulo *nombre_modulo*, y:

```
http://nonbredelservidor/dietetica_dev.php/nombre_modulo/nombre_accion
```

devolvería lo mismo con la diferencia de que en la parte superior derecha de la página aparecería una barra de depuración mediante la que podemos acceder información detallada sobre la ejecución de la aplicación: datos de la petición, datos de la respuesta, configuración de la vista, consultas *SQL* ejecutadas, tiempo de ejecución, configuración del servidor, etcétera. Como puedes suponer, datos tan valiosos para el desarrollo como peligrosos para ser enviados por error en un entorno de producción.

Ahora vamos a construir el módulo que albergará las acciones de nuestra aplicación.

```
$ symfony generate:module dietetica alimentos
```

La instrucción anterior genera un módulo de la aplicación *dietetica* denominado *alimentos*. Lo que físicamente se traduce en la creación del directorio *apps/dietetica/modules/alimentos*. Vamos a describir su contenido:

iY ahora a programar!

actions	<p>Como el propio nombre del directorio sugiere, aquí se definirán las acciones del módulo. Abrimos el directorio y nos encontramos con un fichero denominado <i>actions.class.php</i>.. En él se implementa una clase que se denomina <i>alimentosActions</i> y que extiende a la clase <i>sfActions</i>. <i>Symfony</i> sabe como y cuando debe tratar las clases derivadas de <i>sfActions</i> cuyo nombre sigue el patrón <i>{nombreModulo}Actions</i>, y que se encuentran en los directorios <i>actions</i> de cada módulo.</p> <p>El programador implementará las acciones del módulo como métodos de dicha clase cuyos nombres deben seguir el patrón <i>execute{NombreAccion}</i>. Además deben ser declarados de alcance público. Por ejemplo, una acción que se llame '<i>inicio</i>' sería implementada en un método de la clase como sigue:</p> <div style="background-color: #f0f0d0; padding: 10px;"><pre>public function executeInicio(sfWebRequest \$request) { ... // Aquí el código de la acción ... }</pre></div> <p>El argumento del método (<i>\$request</i>), es un objeto que lleva información acerca de los parámetros de la petición <i>HTTP</i> (<i>GET</i>, <i>POST</i>, <i>FILES</i>, y algunas otras cosas más), para facilitar al programador su acceso y manipulación de una misma manera en todos los proyectos desarrollados con <i>symfony</i>.</p>
templates	<p>Una vez que la acción se ha ejecutado, al igual que en el mini-framework de la unidad anterior, los datos que ha procesado deben ser visualizados de alguna manera. En este directorio se implementarán las plantillas encargadas de realizar dicha tarea. Cada plantilla se corresponde con un archivo <i>PHP</i> confeccionado de manera que muestre explícitamente la estructura <i>HTML</i>, exactamente de la misma forma que explicamos en la unidad anterior. Por lo general, el nombre de las plantillas seguirá el siguiente patrón: <i>{nombreAccion}Success.php</i>, y, como podrás imaginar, será una plantilla creada para visualizar los datos de la acción <i>nombreAccion</i>. Sin embargo también existen otras posibilidades que serán explicadas más adelante.</p>

Aunque la tarea *generate:module* sólo crea los directorios anteriores, en ocasiones también tendremos que agregar a mano los directorios *lib*, *config* y *doc* a nivel de módulo.

En el primero podremos colocar clases y funciones externas utilizadas exclusivamente por el módulo. Al igual que en los otros directorios *lib*, a nivel de aplicación y de proyecto, las librería que residan en este directorio serán cargadas automáticamente por el *framework* cuando sean necesarias.

En el segundo podremos incluir archivos de configuración propios del módulos, y en el tercero la documentación del mismo.

iY ahora a programar!

Llegó la hora de trabajar un poco. Hasta el momento ha sido *symfony* quien ha trabajado por nosotros generando la sólida estructura del edificio que vamos a construir. Ahora nos corresponde a nosotros construir sus tabiques y decorar los ambientes.

Acción de inicio (index)

Antes de comenzar, y para que todo funcione correctamente, es preciso que modifiques el fichero de configuración `apps/dietetica/config/settings.yml` y pongas el parámetro `no_script_name` de la sección `prod` (es decir del entorno de ejecución de producción) a `false`. En el caso de estar activado, este parámetro indica a `symfony` que debe eliminar el nombre del controlador frontal (`index.php`) cuando se construyen las `URL's` de la aplicación. Lo cual estiliza la `URL` pero requiere que el servidor web esté debidamente configurado para un entorno de producción, y este no es nuestro caso.

Acción de inicio (index)

Comenzaremos por la implementación de la acción '`inicio`', la más sencilla de la aplicación. No obstante, para aprovechar las características de `symfony`, en lugar de '`inicio`' le vamos a llamar '`index`', ya que es así como se denomina la acción por defecto en `symfony`.

El siguiente código muestra el código de la clase `alimentosActions` una vez implementada la acción '`index`':

Contenido del archivo: `apps/dietetica/modules/alimentos/actions/actions.class.php`

```
<?php

/**
 * alimentos actions.
 *
 * @package    alimentos
 * @subpackage alimentos
 * @author     Your name here
 * @version    SVN: $Id: actions.class.php 12479 2008-10-31 10:54:40Z fabien $
 */
class alimentosActions extends sfActions
{
    /**
     * Executes index action
     *
     * @param sfWebRequest $request A request object
     */

    public function executeIndex(sfWebRequest $request)
    {
        $this -> param_mensaje = 'Bienvenido a la primera aplicación del curso "desarrollo de aplicaciones web con symfony"';
        $this -> param_fecha = date('d - M - Y');
    }
}
```

El código es esencialmente el mismo que su homólogo de la aplicación desarrollada en la unidad 2. La diferencia estriba en que:

1. la acción se implementa en un método de la clase `alimentosActions` en lugar de hacerse en un fichero.
2. los parámetros que deseamos mostrar en la plantilla deben definirse como atributos de la clase `alimentosActions` mediante el identificador `$this`, que es una referencia al objeto que ha sido instanciado.

Cuando finaliza la ejecución de la acción se ejecuta la plantilla `indexSuccess.php`, la cual reconoce como parámetros dinámicos variables cuyo nombre coincide con los atributos definidos mediante `$this` en la acción. Veamos todo esto más claramente en el código de la plantilla `indexSuccess.php`:

Contenido del archivo:
`apps/dietetica/modules/alimentos/templates/indexSuccess.php`

```
<h3> Fecha: <?php echo $param_fecha ?> </h3>
<?php echo $param_mensaje ?>
```

Expresado de una manera poco formal: los parámetros que deseamos mostrar en la plantilla y que provienen de la acción, se invocan con el mismo nombre que se le dio en la acción pero sin utilizar `$this`.

Acción de inicio (index)

Y ahora ya podemos ver el resultado de nuestra primera acción. Abrimos el navegador web y solicitamos el recurso correspondiente a la *URL*:

```
http://localhost/unidad3/web/index.php/alimentos/index
```

Y veremos el resultado de ejecutar la acción *index* del módulo *alimentos*. Cómo *index* es la acción por defecto, podemos obtener el mismo resultado eliminando el parámetro *index* de la petición:

```
http://localhost/unidad3/web/index.php/alimentos
```

Nota

Hay que aclarar aquí que *symfony* utiliza un sistema de enrutamiento que permite el uso de *URL*'s limpias, lo cual significa que el paso de parámetros *GET* en la petición *HTTP* no se realiza mediante los caracteres '?' y '&' tras el nombre del fichero *index.php*. En su lugar se utiliza únicamente el carácter '/'. De manera que el primer parámetro tras *index.php*, *alimentos* en nuestro caso, corresponderá al módulo, mientras que el segundo, *index* en nuestro caso, a la acción que se desea ejecutar. Si la acción requiriese parámetros adicionales se indicarían los pares *parámetro/valor-parámetro* separados igualmente por el carácter '/'.

El sistema de enrutamiento es una herramienta muy potente que interpreta ("parsea") la *URL* de acuerdo a un conjunto de reglas, que pueden ser definidas por el programador, para saber qué acción de qué módulo debe ser ejecutada y con qué parámetros. En un principio utilizaremos las reglas preestablecidas de enrutamiento, que consisten en lo que acabamos de explicar. Por lo pronto, para desarrollar aplicaciones *web* con *symfony* no es necesario saber más acerca del sistema de enrutamiento. Pero señalaremos que la configuración a medida de dicho sistema mejora tanto el aspecto como la seguridad de la aplicación, ya que es posible eliminar información estructural acerca de la propia aplicación, como puede ser el nombre de ciertos parámetros.

Si queremos usar la herramienta de depuración debemos utilizar el controlador de desarrollo. Utiliza ahora esta *URL* para ejecutar la acción:

```
http://localhost/unidad3/web/dietetica_dev.php/alimentos/index
```

Observa la presencia de una barra de herramientas en la esquina superior derecha de la página recibida. Te aconsejamos que eches un rato jugando con ella. Explora e interpreta la información que ofrece. A medida que desarrolles con *symfony* encontrarás más valiosa esta herramienta pues te ayudará a resolver muchos problemas.

Nota

si quieres ver la barra de depuración con iconos debes copiar el siguiente directorio de la distribución de *symfony* al directorio *web* del proyecto:

Acción de inicio (index)

```
cp -R /ruta/a/symfony/data/web/sf web/
```

Ya tenemos el resultado de la acción inicial de presentación, pero echamos de menos el menú y el pie de página que debería enmarcar a todas las acciones de la aplicación. Lo incorporaremos a continuación definiendo el *layout* de la aplicación, lo que implica modificar el archivo *apps/dietetica/templates/layout.php*.

Contenido del archivo: *apps/dietetica/templates/layout.php*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <?php include_http_metas() ?>
        <?php include_metas() ?>
        <?php include_title() ?>
        <?php include_stylesheets() ?>
        <?php include_javascripts() ?>
        <link rel="shortcut icon" href="/favicon.ico" />
    </head>
    <body>
        <div id="cabecera">
            <h1>Información de alimentos</h1>
        </div>

        <div id="menu">
            <hr/>
            <a href=<?php echo url_for('alimentos/index') ?>>>inicio</a> | 
            <a href=<?php echo url_for('alimentos/insertarAlimento') ?>>>insertar alimento</a> | 
            <a href=<?php echo url_for('alimentos/buscarAlimentosPorNombre') ?>>>buscar por nombre</a> | 
            <a href=<?php echo url_for('alimentos/buscarAlimentosPorEnergia') ?>>>buscar por energía</a> | 
            <a href=<?php echo url_for('alimentos/buscarAlimentosCombinada') ?>>>búsqueda combinada</a>
            <hr/>
        </div>

        <div id="contenido">
            <?php echo $sf_content ?>
        </div>

        <div id="pie">
            <hr/>
            <div align="center">- pie de página -</div>
        </div>
    </body>
</html>
```

Lo resaltado en negrita es la parte que hemos añadido al fichero *apps/dietetica/templates/layout.php* que fue generado automáticamente por *symfony*. Observa por otro lado la línea sombreada. Ella es la responsable de mostrar el resultado de la acción en curso. La variable *\$sf_content* es propia de *symfony* y su valor es el contenido de la plantilla una vez realizadas las sustituciones de los parámetros dinámicos definidos en la acción. Dicha línea es la equivalente a la siguiente línea del *layout* del mini-framework de la unidad anterior:

```
<?php include('plantillas/'.{$mvc_vis_plantilla}.'Plantilla.php') ?>
```

Otra diferencia del *layout* de *symfony* respecto al de nuestro mini-framework es el contenido de la sección *head*. En el caso de *symfony* las etiquetas *meta* y el título de la página (*title*) son establecidos por configuración mediante el fichero de configuración de la aplicación *apps/dietetica/config/view.yml*. En este archivo también se especifican las *CSS's* que deseamos utilizar y los archivos *javascript*.

Por último observa los enlaces que forman el menú: el atributo *href* se construye a través de una función denominada *url_for()*, la cual construye la *URL* correcta para la ejecución del módulo y la acción que se le pasa en el argumento. De esta manera cuando cambiemos la

Búsqueda de alimentos por nombre

aplicación de servidor y/o ubicación, no tendremos que cambiar todos los enlaces de las acciones, ya que *symfony* construirá la *URL* correcta mediante dicha función. Ni que decir tiene que para lograr esta independencia de las *URL's* respecto de la ubicación de la aplicación es necesario utilizar la función *url_for()*. *Symfony* proporciona un conjunto de funciones de este tipo y son denominadas *helpers*.

Ejecutamos de nuevo la acción *inicio* de la aplicación *dietetica* en el navegador y comprobamos como ahora aparece el menú y el pie de página, aunque aún sin estilos. Así que vamos a incluir las mismas *CSS's* de la unidad 2 para obtener el mismo resultado. Dos párrafos antes hemos dado la clave para saber como incluir las *CSS's* y los *javascript*. Abre el fichero *apps/dietetica/config/view.yml* y dile que deseas utilizar la hoja de estilos *estilo.css*.

```
default:
    http_metas:
        content-type: text/html

metas:
    title:      dietética
    description: symfony project
    keywords:   symfony, project, alimentos
    language:   es
    robots:     index, follow

stylesheets:  [estilo.css]
javascripts: []

has_layout:  on
layout:       layout
```

Hemos aprovechado la edición de este fichero para cambiar algunas etiquetas *metas*.

Por último hemos de colocar el archivo *estilo.css* en su sitio que es, como ya se ha explicado anteriormente, el directorio *web/css* del proyecto. Si quisieramos incluir más *CSS's* tan solo tenemos que añadir los ficheros correspondientes en este directorio e indicarlo en el archivo de configuración de la aplicación *view.yml*.

Ahora si, ya tenemos la primera acción de nuestra aplicación funcionando al 100%.

Búsqueda de alimentos por nombre

Ahora se trata de adaptar el código de la unidad anterior al esquema de funcionamiento de *symfony*, tal y como hicimos con la acción *inicio* (*index*). Para ello tenemos en cuenta que las acciones se construyen como métodos públicos de la clase *alimentosActions* que comienzan con el prefijo *execute*, y que las plantillas se llaman como la acción cuyos resultados se desea mostrar seguidas del sufijo *Success*.

En primer lugar presentamos el formulario de búsqueda de alimentos por nombre. Para ello, igual que se hizo en la unidad 2, creamos una acción que denominaremos *buscarAlimentosPorNombre*. Esta acción tan sólo tiene que especificar la plantilla que dibuja al formulario. Pero como *symfony* automáticamente utiliza una plantilla denominada de la misma forma que la acción terminada en *Success* podemos dejar vacía la acción. Es decir cuando sea ejecutada esta acción se ejecutará una función vacía, lo cual dará un resultado satisfactorio y, automáticamente, se interpretará el código de la plantilla *buscarAlimentosPorNombreSuccess.php*.

Así pues al fichero *actions.class.php* debemos añadir la siguiente función:

Búsqueda de alimentos por nombre

```
public function executeBuscarAlimentosPorNombre(sfWebRequest $request)
{
}
```

Y en el directorio *templates* del módulo *alimentos*, añadiremos el fichero *buscarAlimentosPorNombreSuccess.php* (la plantilla), cuyo contenido es el siguiente:

```
<form name="formBusqueda" action="php echo url_for('alimentos/procesarFormBusquedaPorNombre') ?" method="POST">
    <table>
        <tr>
            <td>alimento:</td>
            <td><input type="text" name="nombre"></td>
            <td><input type="submit" value="buscar"></td>
        </tr>
    </table>
</table>
</form>
```

Es decir, exactamente el mismo código de la plantilla homóloga de la unidad 2, que se denominamos entonces: *formBusquedaPorNombrePlantilla.php*. La única diferencia es el contenido del atributo *action* del formulario; en este caso la *URL* se construye con el *helper url_for()*.

Podemos ejecutar ahora la acción recién horneada y veremos el formulario decorado con el *layout* de la aplicación, es decir con el menú y el pie de página que, como ya debes saber, es común para toda la aplicación.

Accedemos a través de la *URL*:

<http://localhost/unidad3/web/index.php/alimentos/buscarAlimentosPorNombre1>

O también a través del menú de la aplicación.

Falta la segunda parte del proceso: procesar los datos que el usuario ha introducido en el formulario, realizar la búsqueda en la base de datos y mostrar los resultados. Para ello necesitaremos una nueva acción con su plantilla correspondiente. El nombre de esta acción ya lo hemos determinado en el parámetro *action* del formulario: *procesarFormBusquedaPorNombre*, así pues añadimos una nueva función al archivo *actions.class.php*:

```
public function executeProcesarFormBusquedaPorNombre(sfWebRequest $request)
{
    $this -> param_alimentos = array();

    $nombreAlimento = $request -> getParameter('nombre');

    $c = new Criteria();
    $c -> add(AlimentosPeer::NOMBRE, $nombreAlimento, Criteria::LIKE);

    $this -> param_alimentos = AlimentosPeer::doSelect($c);

    $this -> setTemplate('mostrarAlimentos');
}
```

Este código da lugar al mismo resultado que su homólogo de la unidad anterior: construir un *array* con información acerca de los alimentos que satisfacen un criterio de búsqueda, aunque la forma de hacerlo es distinta.

Búsqueda de alimentos por nombre

En primer lugar, los parámetros del formulario que forman parte de la petición *POST*, no se recogen utilizando la variable global `$_POST` de *PHP*, en su lugar se utiliza el objeto `$request` que es el argumento de todas las acciones (métodos `execute`) de *symfony*, y que proporciona, como veremos a lo largo del curso, útiles métodos para la gestión de la petición *HTTP* (`request`) además de una manera homogénea de tratar los parámetros en toda la aplicación. La función `getParameter()` admite como argumento un *string* y devuelve el valor del parámetro (*GET* o *POST*) cuyo nombre coincide con el argumento. Así pues `$nombreAlimento` almacenará el valor que el usuario haya introducido en el formulario de búsqueda.

En segundo lugar, no se ha utilizado ninguna sentencia *SQL* explícitamente para realizar la búsqueda en la base de datos. Los responsables de realizar esta función son el objeto *Criteria* y la función `doSelect()` de la clase estática *AlimentosPeer*. Esta clase pertenece al modelo, y puedes encontrarla en un archivo denominado *AlimentosPeer.php* dentro del directorio *lib* del proyecto. El funcionamiento de las clases del modelo de *Propel* para la manipulación de las bases de datos lo estudiaremos en una unidad dedicada a ello. Pero para calmar la curiosidad explicaremos de forma breve, sin entrar en detalles profundos, como se utiliza el modelo para realizar consultas.

Comienza el proceso por definir un objeto de la clase *Criteria* (`$c` en nuestro código), al que se le añaden mediante el método `add` todos los criterios de búsqueda que precisemos. Esto equivale a establecer la parte *WHERE* de una sentencia *SQL*. Acto seguido se pasa el objeto *Criteria* como argumento a la función `doSelect()` de una clase estática que dispone de funciones para gestionar los registros de las tablas de la base de datos. Cada tabla tiene asociada una clase de este tipo cuyo nombre es *NombreTablaPeer*. En el caso que nos ocupa: *AlimentosPeer*. La función `doSelect()` se encarga de construir la sentencia *SQL* que corresponda al Sistema Gestor de Base de Datos que se utilice en la aplicación, y nos devuelve un *array* con el conjunto de registros que satisface la consulta. Cada registro devuelto en dicho *array* es representado por un objeto que proporciona métodos para acceder y manipular el valor de sus campos (*getters* y *setters*). La clase que representa dichos registros se llama igual que la tabla que manipulamos; en nuestro caso *Alimentos*. Según lo dicho, la variable `$this` → `param_alimentos` será un array de objetos *Alimentos*. Además, al ser definido como miembro de la clase *alimentosActions*, estará disponible en la plantilla subsiguiente.

Según lo visto hasta el momento, lo que deberíamos hacer a continuación es crear una plantilla denominada *ProcesarFormBusquedaPorNombreSuccess.php*, para pintar los datos obtenidos por la acción. Sin embargo, dado que vamos a desarrollar más acciones que también muestran una lista de alimentos, hemos decidido implementar una única plantilla que denominaremos *mostrarAlimentosSuccess.php* (fíjate que el nombre no corresponde a ninguna acción, lo acabamos de inventar), y obligaremos a la acción a que utilice esta plantilla mediante la función `setTemplate()` de la clase *sfActions* (de la cual deriva *alimentosActions*). Esto es lo que hace precisamente la última línea de la acción `executeProcesarFormBusquedaPorNombre()`.

A continuación mostramos el código de la nueva plantilla *mostrarAlimentosSuccess.php*:

```
<table>
    <tr>
        <th>alimento (por 100g)</th>
        <th>energía (Kcal)</th>
        <th>grasa (g)</th>
    </tr>
    <?php foreach ($param_alimentos as $alimento) :?>
    <tr>
        <td><?php echo $alimento -> getNombre() ?></td>
```

Búsqueda de alimentos por energía

```
<td><?php echo $alimento -> getEnergia() ?></td>
<td><?php echo $alimento -> getGrasatotal() ?></td>
</tr>
<?php endforeach; ?>

</table>
```

Volvemos a utilizar el bucle *foreach* al estilo de plantillas (llamémosle de esta manera cuando no se utilizan llaves para delimitar su alcance) tal y como hicimos en la unidad anterior, para iterar sobre el *array* de objetos *Alimentos* que representan registros de la tabla *alimentos* y que proviene de la acción *procesarFormBusquedaPorNombre*. Observa como el valor de los campos de la tabla *alimentos* es accedido mediante los métodos *getters* del objeto. La plantilla construye una tabla con los alimentos encontrados en la acción de búsqueda. Es importante percatarse de que esta plantilla podrá ser utilizada por cualquier acción que le facilite un *array* de objetos *Alimentos* denominado *\$param_alimentos*.

Ya sólo nos queda probar el funcionamiento de la búsqueda por nombre desde el principio, es decir, llenar el formulario y pulsar en el botón 'buscar'.

Búsqueda de alimentos por energía

En esencia, esta funcionalidad es idéntica a la búsqueda de alimentos por nombre. Por ello transcribimos el código de las acciones y plantillas que la implementan y explicamos las singularidades del caso.

Añadimos los siguiente métodos a la clase *alimentosActions*:

```
public function executeBuscarAlimentosPorEnergia(sfWebRequest $request)
{
}

public function executeProcesarFormBusquedaPorEnergia(sfWebRequest $request)
{
    if(is_numeric($_POST['energia_min']) && is_numeric($_POST['energia_max']))
    {
        $this -> param_alimentos = array();

        $energiaMin = $request -> getParameter('energia_min');
        $energiaMax = $request -> getParameter('energia_max');

        $c = new Criteria();
        $c -> add(AlimentosPeer::ENERGIA, $energiaMin, Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::ENERGIA, $energiaMax, Criteria::LESS_THAN);
        $c -> addDescendingOrderByColumn(AlimentosPeer::ENERGIA);

        $this -> param_alimentos = AlimentosPeer::doSelect($c);

        $this -> setTemplate('mostrarAlimentos');
    }
    else
    {
        $this -> setTemplate('mostrarErrorFormulario');
    }
}
```

Y creamos las siguientes plantillas:

El formulario de búsqueda *buscarAlimentosPorEnergiaSuccess.php*:

Búsqueda combinada.

```
<form name="formBusqueda" action="php echo url_for('alimentos/procesarFormBusquedaPorEnergia') ?" method="POST">
    <table>
        <tr>
            <th>propiedad</th>
            <th>mínimo</th>
            <th>máximo</th>
        </tr>
        <tr>
            <td>energía (Kcal):</td>
            <td><input type="text" name="energia_min"></td>
            <td><input type="text" name="energia_max"></td>
            <td> <input type="submit" value="buscar"></td>
        </tr>
    </table>
</form>
```

Y la plantilla *mostrarErrorFormularioSuccess.php*:

```
<b>Algún parámetro del formulario no es válido</b>
<br/>
<a href="javascript:history.back(1)">Volver</a>
```

Examinemos las singularidades del caso:

En primer lugar, la acción recoge dos parámetros que provienen del formulario: *energia_min* y *energia_max*, y comprueba que ambos sean valores numéricos para garantizar una búsqueda sin errores. Si no lo son, se muestra un mensaje de error a través de la plantilla *mostrarErrorFormularioSuccess.php*, y en caso contrario se procesa la petición mediante la creación de un criterio. Ahora el criterio tiene dos condiciones *AND*, lo que se traduce en utilizar dos veces el método *add()*. Además como deseamos mostrar los alimento en orden decreciente de energía, utilizamos el método *addDescendingOrderByColumn()* de este mismo objeto. La función *doSelect()* nos devolverá un *array* de objetos *Alimentos* que cumplen dicho criterio. Por último mostramos el resultado con la misma plantilla (*mostrarAlimentosSuccess.php*) que confeccionamos en el apartado anterior.

Y la funcionalidad de búsqueda queda así construida. Ahora puedes probarla accediendo a ella desde el menú de la aplicación o desde la URL:

<http://localhost/unidad3/web/index.php/alimentos/buscarAlimentosPorEnergia>

Búsqueda combinada.

Esta funcionalidad permitirá la búsqueda de alimentos cuyas cantidades de energía (en Kcal), proteínas (en gramos), hidratos de carbono (en gramos), fibra (en gramos) y grasa en (gramos) se encuentren simultáneamente entre unas cantidades mínimas y máximas definidas por el usuario para cada magnitud.

Esencialmente es igual que el caso anterior, basta con definir 10 casillas en lugar de 2 para la introducción de los valores mínimos y máximo de cada magnitud, y modificar el criterio de búsqueda (objeto *Criteria*) para incluir todas las condiciones.

Sin embargo, aunque podamos proceder de esta manera para realizar la implementación de la búsqueda combinada, desarrollaremos el apartado utilizando un nuevo elemento muy útil y poderoso de *symfony*: el *framework* de formularios. Como corresponde a este capítulo, realizaremos un acercamiento panorámico con la finalidad de mostrar su eficacia. Más adelante dedicamos un capítulo completo a los formularios de *symfony* donde los estudiaremos con detalle.

Comencemos por mostrar el código de las acciones que debemos añadir a la clase *alimentosActions*:

Búsqueda combinada.

```
public function executeProcesarFormBusquedaCombinada(sfWebRequest $request)
{
    $this -> formulario = new BusquedaCombinadaForm();

    $datos = $request -> getParameter('parametros');

    $this->formulario->bind($datos);
    if ($this->formulario->isValid())
    {
        $c = new Criteria();
        $c -> add(AlimentosPeer::ENERGIA, $datos['energia_min'], Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::ENERGIA, $datos['energia_max'], Criteria::LESS_THAN);
        $c -> add(AlimentosPeer::PROTEINA, $datos['proteina_min'], Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::PROTEINA, $datos['proteina_max'], Criteria::LESS_THAN);
        $c -> add(AlimentosPeer::HIDRATOCARBONO, $datos['hc_min'], Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::HIDRATOCARBONO, $datos['hc_max'], Criteria::LESS_THAN);
        $c -> add(AlimentosPeer::FIBRA, $datos['fibra_min'], Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::FIBRA, $datos['fibra_max'], Criteria::LESS_THAN);
        $c -> add(AlimentosPeer::GRASATOTAL, $datos['grasa_min'], Criteria::GREATER_THAN);
        $c -> add(AlimentosPeer::GRASATOTAL, $datos['grasa_max'], Criteria::LESS_THAN);

        $this -> param_alimentos = AlimentosPeer::doSelect($c);
        $this -> setTemplate('mostrarAlimentos');
    }
    else
    {
        $this -> setTemplate('buscarAlimentosCombinada');
    }
}
```

Al igual que en los casos anteriores añadimos dos acciones: una para mostrar el formulario y la otra para procesarlo. Analicemos las diferencias.

Comprobamos que en la acción *buscarAlimentosCombinada*, se define un atributo de la clase *alimentosActions* denominado *\$this->formulario* como instancia de la clase *BusquedaCombinadaForm*. Dicho atributo, como ya hemos estudiado anteriormente, estará disponible en la plantilla correspondiente. La clase *BusquedaCombinadaForm* define el formulario que deseamos utilizar en la búsqueda combinada.

En efecto, los formularios de *symfony* consisten en clases definidas por el usuario que derivan de la clase base *sfForm* ofrecida por el propio *framework*. Los formularios son considerados clases de librería, y por tanto su lugar de residencia es alguno de los directorios *lib* del proyecto. Como es un formulario que utilizaremos exclusivamente en el módulo *alimentos*, lo colocamos bajo el directorio *lib* de dicho módulo. Este directorio debemos crearlo a mano pues la tarea *generate:module* no lo hace por defecto. Así pues creamos el directorio *lib*:

```
$ mkdir apps/dietetica/modules/alimentos/lib
```

Y creamos el fichero *BusquedaCombinadaForm.php* donde definiremos el formulario:

```
<?php

class BusquedaCombinadaForm extends sfForm
{
    public function configure()
    {
        $this->setWidgets(array(
            'energia_min' => new sfWidgetFormInput(),
            'energia_max' => new sfWidgetFormInput(),
            'proteina_min' => new sfWidgetFormInput(),
```

Búsqueda combinada.

```
'proteina_max' => new sfWidgetFormInput(),
'hc_min'        => new sfWidgetFormInput(),
'hc_max'        => new sfWidgetFormInput(),
'fibra_min'     => new sfWidgetFormInput(),
'fibra_max'     => new sfWidgetFormInput(),
'grasa_min'     => new sfWidgetFormInput(),
'grasa_max'     => new sfWidgetFormInput(),
));
$this->widgetSchema->setNameFormat('parametros[%s]');

$this->setValidators(array(
'energia_min'  => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'energia_max'  => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'proteina_min'  => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'proteina_max'  => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'hc_min'        => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'hc_max'        => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'fibra_min'     => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'fibra_max'     => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'grasa_min'     => new sfValidatorNumber(array('required' => true, 'min' => 0)),
'grasa_max'     => new sfValidatorNumber(array('required' => true, 'min' => 0)),
));
}
?>
```

Los formularios de *symfony* están compuestos por **widgets** que representan los elementos de formulario *HTML* (*input*, *checkbox*, *textarea*, *radio*, etcétera) y por **validadores** asociados a cada uno de los *widgets*. Analiza el código anterior y podrás identificar fácilmente cada uno de estos elementos. Nuestro formulario *BusquedaCombinadaForm*, consta de 10 *widgets* del tipo *sfWidgetFormInput*. A cada uno de esos *widgets* le hemos asociado un validador del tipo *sfValidatorNumber* para garantizar que los valores introducidos por el usuario sean numéricos. Además, para todos los casos, se exige mediante cada validador, que el campo no se puede dejar vacío y que el valor introducido debe ser mayor o igual a 0.

Los objetos de tipo formulario, es decir las instancias de clases que derivan de la clase base *sfForm* como por ejemplo *BusquedaCombinadaForm*, nos ofrecen dos operaciones básicas: el renderizado, y la validación. La primera consiste en desplegar el código *HTML* que le corresponde a cada *widget*, y se utiliza en las plantillas donde se definen los formularios. La segunda consiste en comprobar si los datos enviados en la petición *HTTP* (*request*) cumplen los criterios especificados para cada *widget* a través de los validadores. Esta última operación se realiza durante el procesado de los datos del formulario.

La plantilla que dibuja el formulario definido en la acción *buscarAlimentosCombinada* se llama *buscarAlimentosCombinadaSuccess.php* y el código es el que sigue:

```
<form name="formBusqueda" action="php echo url_for('alimentos/procesarFormBusquedaCombinada') ?&gt;" method="POST"&gt;

&lt;?php echo $formulario -&gt; renderGlobalErrors() ?&gt;
&lt;?php echo $formulario -&gt; renderHiddenFields() ?&gt;

<table border="1"| propiedad | mínimo | máximo |
| --- | --- | --- |
| energía (Kcal): | <?php echo $formulario['energia_min'] -> renderError() ?><?php echo $formulario['energia_min']-> render() ?> | <?php echo $formulario['energia_max'] -> renderError() ?><?php echo $formulario['energia_max']-> render() ?> |
| proteína (g): | <?php echo $formulario['proteina_min'] -> renderError() ?><?php echo $formulario['proteina_min']-> render() ?> |  |

```

Búsqueda combinada.

```
</tr>
<tr>
    <td><?php echo $formulario['proteina_max'] -> renderError() ?><?php echo $formulario['proteina_max']-> render() ?>
    <td>hidratos de carbono (g):</td>
    <td><?php echo $formulario['hc_min'] -> renderError() ?><?php echo $formulario['hc_min']-> render() ?>
    <td><?php echo $formulario['hc_max'] -> renderError() ?><?php echo $formulario['hc_max']-> render() ?>
</tr>
<tr>
    <td>fibra (g):</td>
    <td><?php echo $formulario['fibra_min'] -> renderError() ?><?php echo $formulario['fibra_min']-> render() ?>
    <td><?php echo $formulario['fibra_max'] -> renderError() ?><?php echo $formulario['fibra_max']-> render() ?>
</tr>
<tr>
    <td>grasa (g):</td>
    <td><?php echo $formulario['grasa_min'] -> renderError() ?><?php echo $formulario['grasa_min']-> render() ?>
    <td><?php echo $formulario['grasa_max'] -> renderError() ?><?php echo $formulario['grasa_max']-> render() ?>
</tr>
</table>
<input type="submit" value="buscar">
</form>
```

Se trata de un formulario *HTML* maquetado como una tabla en la que cada fila contienen el nombre de la magnitud y dos casillas de texto para que el usuario inserte el valor mínimo y máximo que se utilizarán en la búsqueda. Observa el uso de los métodos *render()* y *renderError()*. El primero se encarga de construir el código *HTML* que le corresponde al *widget* según su tipo. El segundo, *renderError()*, devuelve un valor vacío la primera vez que se dibuja el formulario, pero cuando la validación de los datos del formulario no ha sido satisfactoria, devuelve mensajes de error que indican al usuario porqué se ha rechazado la petición.

El método *renderGlobalErrors()*, como su nombre indica, mostraría errores asociados al formulario en su conjunto. Por último el método *renderHiddenFields()* dibuja los elementos *HTML* ocultos que pueda tener el formulario. Aunque no hemos definido ningún tipo de elemento oculto en nuestro formulario, es necesario utilizar el método *renderHiddenFields()* pues el formulario genera automáticamente un campo oculto denominado *_csrf* que sirve para evitar un tipo de ataque conocido como “*cross site retrieve forgery*”. Este comportamiento podemos anularlo eliminando del archivo de configuración *apps/dietetica/config/settings.yml* el parametro *csrf_secret*. No obstante lo recomendamos pues de esa manera disminuiríamos el grado de seguridad de la aplicación.

El flujo de operaciones que tienen lugar en la búsqueda combinada es el siguiente:

1. En la acción *buscarAlimentosCombinada* se define el objeto *\$this->formulario* como una instancia de la clase *BusquedaCombinadaForm*.
2. La plantilla *buscarAlimentosCombinada* dibuja el formulario *HTML* mediante el renderizado de los *widgets* que componen el objeto formulario anterior. Observa el uso del método *render* en el código anterior.
3. En la parte cliente (navegador *web*), el usuario de la aplicación introduce los datos y envía el formulario al servidor.
4. Se ejecuta la acción *procesarFormBusquedaCombinada* tal y como se ha indicado en el atributo *action* del elemento *form* de la plantilla. Esta acción se encarga de validar los datos de la petición *HTTP* (*POST request*) que provienen del formulario y realizar la búsqueda en caso de que los datos sean válidos.

Observemos más de cerca este último punto para comprender como se realiza la validación:

1. En primer lugar se crea un nuevo formulario, es decir un objeto de la clase *BusquedaCombinadaForm*. Dicho formulario se declara como miembro de la clase *alimentosActions* para que pueda estar disponible en la plantilla. Dicho de otro modo, se utiliza el identificador *\$this*.
2. Se asocia el formulario recién creado a los parámetros de la petición *HTTP* mediante el método *bind()* del formulario. Esta operación permitirá delegar la validación de los datos al propio objeto formulario, aliviando al programador de realizar una de las tareas más enrevesadas de la programación de aplicaciones *web*.

Inserción de registros

3. Mediante el método *isValid()* del formulario se comprueba la validez de los datos recibidos según lo especificado en la sección de validadores de la clase *BusquedaCombinadaForm* donde se definió nuestro formulario. Reincidimos en el hecho de que es el propio objeto formulario quién realiza la compleja tarea de validar los datos.
4. Si los datos pasan el test de validez, se realiza la consulta construyendo el criterio adecuado con los datos del formulario y utilizando el método *doSelect()*. El conjunto de alimentos encontrados serán visualizados por la plantilla *mostrarAlimentoSuccess.php* que ya hemos utilizado en las otras búsquedas. Si los datos no son válidos, la acción se dibujará mediante la plantilla *buscarAlimentosCombinadasSuccess.php*, es decir, la misma que dibuja el formulario, pero esta vez el objeto formulario que la acción le pasa está asociado a unos datos determinados (los que venían en la petición *HTTP*) y, debido a que no ha pasado el test de validez, contiene unos mensajes de error que informan sobre las causas del rechazo del formulario. Así pues el formulario que se entrega en esta ocasión al cliente irá lleno con los datos que el usuario introdujo anteriormente y, gracias al método *renderError()*, con los mensajes de error.

Nota

El borrado de la caché. Para mejorar el tiempo de ejecución de los *scripts*, *symfony* incorpora un mecanismo de caché mediante el cual, tanto la configuración como las funciones de librería son cacheadas la primera vez que se ejecuta la aplicación. Eso significa que si, después de haber ejecutado la aplicación haces un cambio en algún fichero de configuración o en alguna librería, estos cambios no se verán reflejados en las siguientes ejecuciones pues la ejecución del *framework* "tira" de los ficheros cacheados. La solución a este problema es lo que se denomina borrar la caché, lo cual se hace mediante el comando:

```
# symfony cc
```

el cual, al eliminar los datos de la caché, obliga a *symfony* a reconstruirlos utilizando los cambios que hayas realizado en la configuración y/o las librerías.

Como acabamos de hacer un cambio en la librería consistente en añadir un formulario al módulo de alimentos, debemos borrar la caché para que este cambio surta efecto.

Atención: muchas veces, especialmente cuando se comienza a utilizar *symfony*, nos olvidamos de este detalle y podemos volvemos locos buscando un error que no existe. Así que cuando algo no te funcione piensa inmediatamente si los cambios que has realizados exigen borrar la caché.

Ahora es el momento de probar la funcionalidad recién implementada. Puedes acceder a ella mediante el menú de la aplicación o a través de la *url*:

<http://localhost/unidad3/web/index.php/alimentos/buscarAlimentosCombinada>

Para probar el funcionamiento de la validación de formulario deja alguna/s casilla/s sin llenar, introduce en otra/s un valor no numérico, en otra/s un valor numérico negativo, y en otra/s un valor numérico positivo. Observa como la acción que procesa el formulario rechaza los datos y muestra de nuevo el formulario con los mensajes de error que le corresponde a cada casilla. ¡Y todo esto con tres líneas de código!, una para definir el formulario, otra para asociarlo a los datos de la petición *HTTP* (*bind*) y otra para realizar la validación (*isValid*).

Inserción de registros

Inserción de registros

Finalizamos la aplicación implementando la funcionalidad de inserción de alimentos en la base de datos. Ahora necesitaremos un formulario con 6 casillas de texto que se correspondan con los campos de la tabla alimento: *nombre*, *energia*, *proteina*, *hidratocarbono*, *fibra* y *grasatotal*. Una vez que se envíen estos datos al servidor, la aplicación los procesará: si son válidos, los insertará en la tabla correspondiente, si no lo son mostrará al usuario los mensajes de error correspondientes sobre el propio formulario.

Según lo que hemos visto en el apartado anterior, debemos definir una clase derivada de *sfForm* para definir dicho formulario. Sin embargo *symfony* ya hizo esto por nosotros cuando comenzamos el proyecto; en efecto, la tarea

```
$ symfony propel:build-forms
```

se encarga de construir para cada tabla de la base de datos, un formulario con los campos que le corresponda. Además estos formularios, al estar asociados a la base de datos, ofrecen métodos automáticos para modificar y añadir registros a sus tablas asociadas. Puedes ver los archivos generados en el directorio *lib/form* del proyecto. ¡Así pues una tarea menos que hacer!

Las acciones que añadiremos para implementar esta funcionalidad son las que siguen:

Trozo del archivo: *apps/dietetica/modules/alimentos/actions/actions.class.php*

```
public function executeInsertarAlimento(sfWebRequest $request)
{
    $this -> formulario = new AlimentosForm();
}

public function executeProcesarFormInsertarAlimento(sfWebRequest $request)
{
    $this -> formulario = new AlimentosForm();

    $datos = $request->getParameter('alimentos');
    $this->formulario->bind($datos);
    if ($this->formulario->isValid())
    {
        $this -> formulario ->save();
        $this -> getUser() -> setFlash('alimento', $datos['nombre']);

        $this -> redirect('alimentos/insertarAlimento');
    }
    else
    {
        $this -> setTemplate('insertarAlimento');
    }
}
```

La primera de ellas define el formulario de inserción que se dibuja mediante la plantilla *insertarAlimentoSuccess.php*, y la segunda procesa los datos que se envían a través de dicho formulario.

Presentamos el código de la plantilla *insertarAlimento* (fichero *insertarAlimentoSuccess.php*):

```
<?php if($sf_user -> hasFlash('alimento')): ?>
<div class="notice">
    el alimento "<?php echo $sf_user -> getFlash('alimento') ?>" se ha insertado correctamente
</div>
```

Ejercicios de ampliación.

```
</div>
<?php endif; ?>

<form name="formInsertar" action="php echo url_for('alimentos/procesarFormInsertarAlimento') ?&gt;" method="POST"&gt;
&lt;?php echo $formulario -&gt; renderGlobalErrors() ?&gt;
&lt;?php echo $formulario -&gt; renderHiddenFields() ?&gt;
&lt;table&gt;
    &lt;tr&gt;
        &lt;th&gt;Nombre&lt;/th&gt;
        &lt;th&gt;Energía (Kcal)&lt;/th&gt;
        &lt;th&gt;Proteína (g)&lt;/th&gt;
        &lt;th&gt;H. de carbono (g)&lt;/th&gt;
        &lt;th&gt;Fibra (g)&lt;/th&gt;
        &lt;th&gt;Grasa total (g)&lt;/th&gt;
    &lt;/tr&gt;
    &lt;tr&gt;
        &lt;td&gt;&lt;?php echo $formulario['nombre'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['nombre'] -&gt; render() ?&gt;&lt;/td&gt;
        &lt;td&gt;&lt;?php echo $formulario['energia'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['energia'] -&gt; render() ?&gt;&lt;/td&gt;
        &lt;td&gt;&lt;?php echo $formulario['proteina'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['proteina'] -&gt; render() ?&gt;&lt;/td&gt;
        &lt;td&gt;&lt;?php echo $formulario['hidratocarbono'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['hidratocarbono'] -&gt; render() ?&gt;&lt;/td&gt;
        &lt;td&gt;&lt;?php echo $formulario['fibra'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['fibra'] -&gt; render() ?&gt;&lt;/td&gt;
        &lt;td&gt;&lt;?php echo $formulario['grasatotal'] -&gt; renderError() ?&gt;&lt;?php echo $formulario['grasatotal'] -&gt; render() ?&gt;&lt;/td&gt;
    &lt;/tr&gt;
&lt;/table&gt;
&lt;input type="submit" value="insertar" name="insertar" /&gt;
&lt;/form&gt;
* Los valores deben referirse a 100 g del alimento</pre
```

Observa de nuevo el uso de los métodos *render* y *renderError* para construir el formulario *HTML*.

Otra singularidad de esta plantilla es que utiliza, al principio del todo, un bloque *if - endif* al estilo de plantillas, es decir sin llaves ({}) para delimitar el bloque. El funcionamiento es exactamente el mismo que los bloques *foreach - endforeach* que ya hemos utilizado anteriormente, pero en este caso con la lógica de un elemento condicional. La condición que se evalúa para ejecutar el bloque es la existencia o no de un parámetro *flash*. Estudiaremos más adelante qué son estos parámetros, por lo pronto baste por decir y comprobar que dicho parámetro es definido en la acción *procesarFormInsertarAlimento* cuando la inserción se ha llevado a cabo con éxito.

Cuando el usuario envía el formulario al servidor, sus datos son procesados, tal y como indica el atributo *action* del elemento *form*, por la acción *procesarFormInsertarAlimento*. Al igual que en el caso de la búsqueda condicional, se define un objeto formulario de la clase *AlimentosForm* y, mediante el método *bind()*, se le asocian los datos enviados en la petición *HTTP*. A continuación se comprueba la validez mediante el método *isValid()*. Si los datos son válidos se insertan el registro en la tabla correspondiente a través de la acción *save()* del formulario. De nuevo nos hemos ahorrado un trabajo, cuando menos, pesado: la inserción de un registro en la tabla *alimentos*. Observa que no necesitamos saber nada acerca de la base de datos, el formulario se encarga de todo.

Ya sólo queda probar la funcionalidad que acabamos de implementar. Accede a ella a través del menú de la aplicación o a través de la *URL*:

```
http://localhost/unidad3/web/index.php/alimentos/buscarAlimentosCombinada
```

Prueba a introducir campos no numéricos en las casillas de los parámetros y a dejar alguna/s casilla/s vacía/s para comprobar el funcionamiento de la validación de formularios.

Ejercicios de ampliación.

La plantilla *mostrarAlimentosHistoEnergiaSuccess.php* con la que se muestran los alimentos encontrados mediante un sencillo histograma:

```
<table>
    <tr>
        <th>alimento (por 100g)</th>
        <th>energía (Kcal)</th>
```

Conclusión

```
</tr>
<?php foreach ($param_alimentos as $alimento) :?>
<tr>
    <td><?php echo $alimento -> getNombre() ?></td>
    <td><?php echo Utilidades::pintaBarra($alimento -> getEnergia())?></td>

</tr>
<?php endforeach; ?>

</table>
```

La función `Utilidades::pintaBarra` la hemos definido en un archivo denominado `Utilidades.class.php` y ubicado en el directorio `apps/dietetica/lib`. Su contenido se detalla a continuación:

```
<?php

class Utilidades
{
    static public function pintaBarra($num)
    {
        $numCaracteres = ceil($num/10.0);
        $barra = str_repeat('*', $numCaracteres);
        return $barra;
    }
}

?>
```

Lo que sucede cuando pedimos al controlador central la ejecución de la acción de un módulo es que, en un determinado momento de la ejecución del *framework*, éste crea un objeto de la clase `{nombreModulo}Actions`, y ejecuta el método `execute{NombreAccion}` de dicho objeto.

Conclusión

Llegamos al final del capítulo y hemos concluido la primera vuelta alrededor del *framework*, la más externa y panorámica. Para ello hemos reescrito la aplicación de la unidad 2 sobre *symfony*. Un ejercicio que nos mostró de forma sencilla los fundamentos del patrón de diseño *Modelo-Vista-Controlador* y los conceptos *acción*, *plantilla* y *layout* de la aplicación. En esta unidad hemos utilizado tales conceptos en el contexto de un *framework* profesional, razón por la que el grado de complejidad ha aumentado considerablemente. Pero el estudiante también puede intuir una mayor solidez estructural y un aumento notable en las posibilidades ofrecidas por *symfony* en relación al mini-*framework* de la unidad 2.

Es muy posible que al final de este capítulo hayas quedado exhausto. Es normal, ya que se han introducido prácticamente todos los conceptos básicos para el desarrollo de aplicaciones en *symfony* sin entrar en profundos detalles, aunque tratándolos de una manera operativa, ya que nuestro propósito ha sido ofrecer una vista panorámica del *framework*. Así que no te preocupes demasiado si piensas que aún no lo tienes todo bien asimilado y controlado, a medida que avances en el curso irás profundizado en los conceptos introducidos en este capítulo. Lo que si te debe quedar claro es la estructura que *symfony* utiliza para organizar sus archivos: el controlador frontal, las acciones del controlador, las plantillas de la vista, el *layout* de la aplicación, el modelo en los directorios *lib* y los archivos de configuración en los directorios *config*.

Conclusión

También es recomendable que, una vez finalizada la unidad, estudies con más rigor el código desarrollado. Modifícalo y comprueba el resultado obtenido con el que has previsto, si no coinciden rectifica y vuelve a probar. Pon a trabajar duro a la intuición y al razonamiento deductivo y lógico, de esa manera irás asimilando progresivamente y casi sin darte cuenta la lógica de funcionamiento de *symfony*. Seguramente, y aunque no conozcas en profundidad el *framework*, si unes tu experiencia a lo que has aprendido en esta unidad, ya podrías desarrollar cualquier tipo de aplicación *web* sobre *symfony*, aunque, por desconocimiento, no uses toda la artillería pesada que el *framework* ofrece y que mejoraría sustancialmente la calidad de la aplicación.

Unidad 4: Análisis de la aplicación “Gestor Documental”

Continuaremos el estudio de *symfony* desarrollando una aplicación *web* con unas funcionalidades que permitan aplicar la mayor parte de las herramientas que el *framework* ofrece. Tal aplicación requiere de un análisis previo que, mediante los modelos apropiados, describa con detalle las características de la misma. El primer objetivo de este capítulo es la exposición de dicho análisis.

Por otra parte, cuando nos dispongamos a construir la aplicación *web* descrita en el análisis y según los principios de diseño impuestos por el *framework symfony*, necesitaremos utilizar una serie de recursos que van más allá del código fuente de las acciones y plantillas que conforman el grueso de la aplicación. En concreto necesitaremos:

- Unas *CSS*'s para dotar de estilo a la aplicación.
- Un modelo de maquetación *HTML* para construir las plantillas y el *layout* de la aplicación.
- Algunos recursos gráficos (logotipo, iconos, ...)

Una base de datos con datos de ejemplo.

El segundo objetivo de esta unidad será construir los recursos que acabamos de citar teniendo en cuenta los dictados del análisis de la aplicación.

En definitiva, en esta unidad desarrollaremos el soporte teórico y los recursos prácticos que necesitaremos en el resto del curso.

Descripción de la aplicación

Vamos a construir un *gestor documental multiusuario*, es decir una aplicación *web* mediante la que los usuarios de la misma puedan almacenar y compartir ficheros, organizarlos, etiquetarlos, buscarlos y descargarlos. Se trata de una herramienta muy útil en cualquier grupo de trabajo que produzca cierta cantidad de documentación.

Un equipo de desarrollo de *software* es un buen ejemplo de usuarios a los que resultaría muy útil la aplicación. En efecto, el ciclo de desarrollo de *software* pasa por varias fases a lo largo de las que se generan distintos tipos de documentos: análisis, diseños, manuales de usuario y otros tantos. Uno de los problemas más comunes es la organización y localización rápida de los documentos. La aplicación que vamos a construir tiene como objetivo principal facilitar dicha tarea.

Existen varias soluciones profesionales en el mercado que cubren esta necesidad. Las más conocidas son *Alfresco*, *Nuxeo* y *KnowledgeTree*, las dos primeras utilizan tecnología Java y se presentan como un *ECM (Entreprise Content Management)* incluyendo el gestor de documentos como un componente más, mientras que la última se ha construido sobre *PHP* y se presenta exclusivamente como gestor de documentos. Obviamente la aplicación que construiremos durante el resto del curso no pretende competir con ninguna de aquellas; nuestro propósito es fundamentalmente pedagógico. Sin embargo, tales aplicaciones profesionales nos han servido como modelo para la extracción de los requisitos fundamentales de nuestro desarrollo.

Por último hemos optado por un gestor documental como aplicación vertebradora del curso porque engloba un amplio conjunto de funcionalidades para tratar los aspectos más relevantes de *symfony*. Además es una aplicación eminentemente práctica, a pesar de que la hemos concebido desde una perspectiva pedagógica. De hecho, y gracias a las posibilidades de expansión de *symfony*, una vez finalizado el curso, la aplicación puede ser mejorada, modificada y/o adaptada hasta el punto que uno desee. También podrá ser utilizada como modelo para el desarrollo de otras aplicaciones que, en principio, no tienen

Catálogo de requisitos

nada que ver con la gestión de documentos; basta con que el programador tenga la suficiente capacidad de abstracción para identificar estructuras análogas en dominios distintos.

Catálogo de requisitos

Hemos clasificado en 4 grupos los requisitos de la aplicación:

- Gestión de usuarios
- Gestión de documentos
- Gestión de comentarios y
- Gestión de puntuaciones

Gestión de usuarios

U.01	La aplicación contemplará 4 tipos de usuarios: <ul style="list-style-type: none">• invitado, que podrá realizar búsquedas y descargas de documentos públicos.• lector, que podrá realizar búsquedas y descargas de todos los documentos• autor, que además podrá subir documentos• administrador, que además podrá administrar todos los aspectos de la aplicación.
U.02	La aplicación presentará una parte pública (perfil invitado) en la que cualquier persona podrá realizar búsquedas de documentos públicos. Para todas las demás acciones el usuario debe estar registrado.
U.03	Los usuarios registrados tendrán asociado un único perfil
U.04	La aplicación permitirá a los usuarios que dispongan del perfil administrador gestionar los usuarios, esto es, darlos de alta, modificarlos y eliminarlos.
U.05	Se almacenarán los siguientes datos de los usuarios registrados: <ul style="list-style-type: none">• nombre (requerido)• apellidos• nombre de usuario (requerido)• <i>password</i> con encriptación MD5 (requerido)• perfil
U.06	La aplicación presentará en la cabecera de cada pantalla el nombre y apellidos del usuario que la está utilizando, así como el perfil que tiene asociado y un botón para salir de la misma.

Gestión de documentos

D.01	Por cuestiones de seguridad, la aplicación permitirá al administrador decidir qué tipo de archivos se podrán subir al repositorio. Se utilizará para ello la comprobación de tipos MIME.
------	--

Comentarios

D.02	<p>Los documentos tendrán asociados los siguientes <i>metadatos</i>:</p> <ul style="list-style-type: none">• título• descripción• fecha de subida al servidor• autor del documento• ¿es público?• categorías
D.03	El administrador de la aplicación podrá añadir, modificar y eliminar las categorías
D.04	Las versiones de los documentos, una vez subidas al servidor, no se podrán borrar ni modificar
D.05	Los ficheros de las versiones deben ser del tipo indicado por el documento al que pertenecen.
D.06	El autor podrá subir nuevas versiones de sus documentos. La numeración de las versiones será automática.
D.07	Cada autor tendrá asociado un directorio físico en el servidor donde serán alojadas las versiones de sus documentos.
D.08	Las búsquedas podrán ser realizadas por título, descripción, tipo, autor y categorías
D.09	<p>Los resultados de las búsquedas se presentarán como listados paginados. Cada fila corresponderá a un documento y sobre la misma fila se indicarán las operaciones que se pueden realizar sobre el documento en función del perfil que esté registrado. Las operaciones son las siguientes, entre paréntesis se muestran los perfiles que pueden realizarlas:</p> <ul style="list-style-type: none">• Descargar (lector, autor, administrador)• Subir nueva versión (autor, administrador)• Ver metadatos (todos los perfiles)• Modificar metadatos (autor, administrador)

Comentarios

C.01	Los usuarios registrados podrán enviar comentarios a los documentos.
C.02	Los usuarios registrados podrán ver los comentarios de los documentos.
C.03	Cada comentario consistirá en: <ul style="list-style-type: none">• autor• fecha de publicación• texto del comentario (máximo 250 caracteres)
C.04	Los comentarios solo podrán ser borrados por el administrador y nunca modificados.

Puntuación

Modelo de datos.

P.01	Los usuarios registrados podrán votar sólo una vez cada documento consultado
P.02	La puntuación se hará de 1 a 10 sin decimales
P.03	La puntuación del artículo consistirá en la media aritmética de todas las votaciones junto con el nº de votaciones que tiene.
P.04	Los listados de las búsquedas podrán ordenarse por puntuación.

Modelo de datos.

La figura 1 representa mediante un diagrama de clases *UML* el modelo de datos que utilizaremos en el desarrollo de la aplicación y que satisface los requisitos anteriores.

Los usuarios deben tener asociado un sólo perfil. Además podrán poseer ninguno, uno o varios documentos y podrán realizar ninguno, uno o varios comentarios a las versiones de los documentos almacenados de otros autores. Todo ello según las reglas especificadas en los requisitos.

Cada documento tiene asociado, como mínimo una versión (la primera), y puede pertenecer a un número cualquiera de categorías.

Las categorías se pueden asociar a cualquier número de documentos.

Cada versión tendrá asociada una votación que se calculará como se indica en los requisitos. Además se les puede asignar cualquier número de comentarios que, a su vez, proceden de los usuarios.

Modelo de datos.

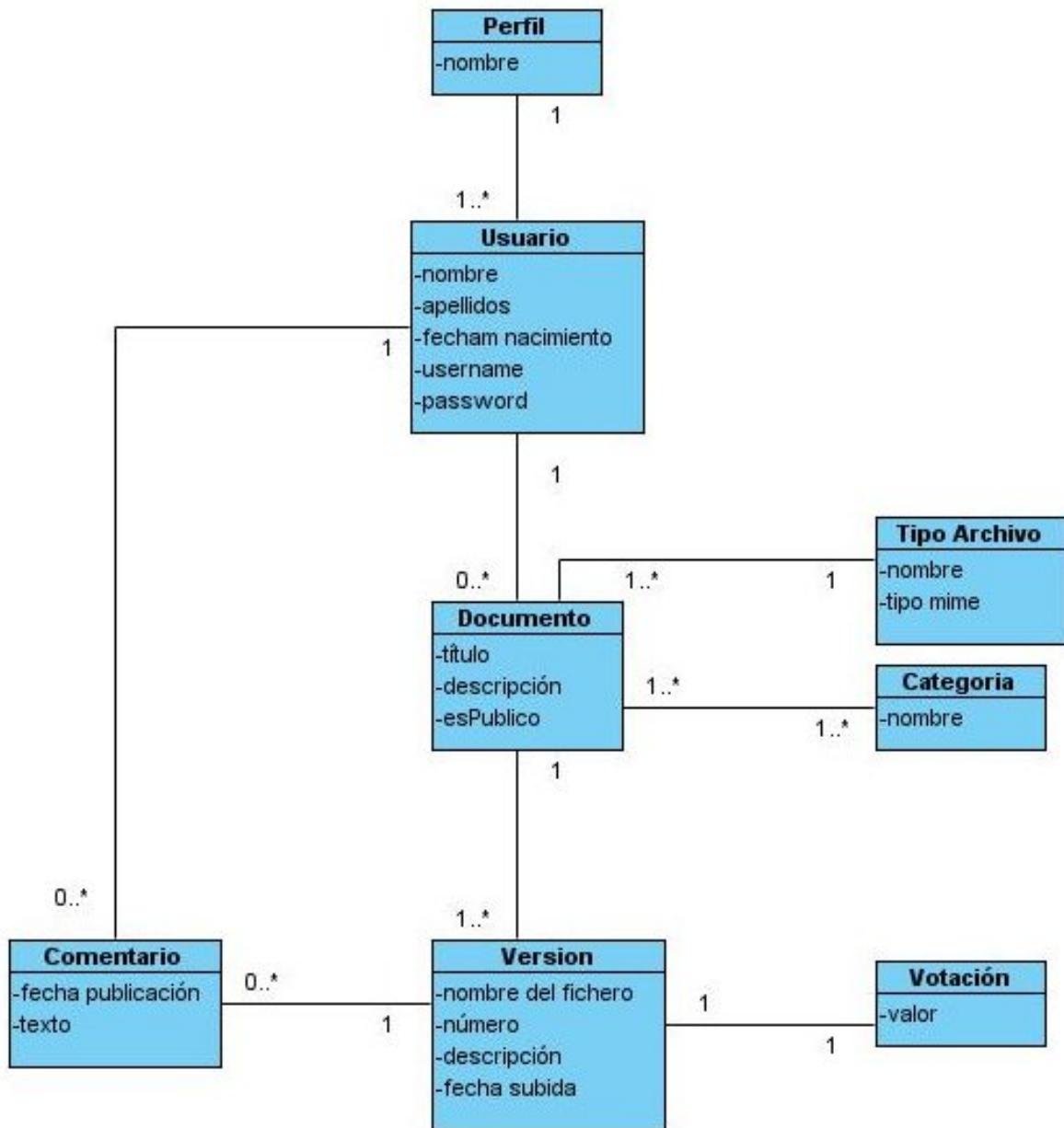


Figura 1. Modelo de datos

Implementaremos este modelo en una base de datos según lo especificado en el diagrama *entidad-relación* de la figura 2. Buscando la simplificación por motivos pedagógicos, hemos sacrificado cierta flexibilidad y capacidad de crecimiento de la aplicación al considerar los perfiles como un campo de la tabla *usuarios*, y las votaciones como un campo de la tabla *versiones*.

Descripción del proceso de subida de archivos

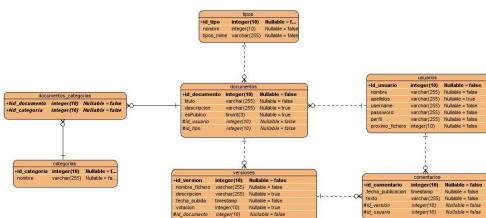


Figura 2. Diagrama Entidad-Relación

Descripción del proceso de subida de archivos

Es importante que no haya colisión en los nombres de los ficheros cuando sean alojados en las carpetas asociadas a cada autor. Por ello hemos diseñado un sencillo procedimiento para nombrar los ficheros enviados al servidor antes de ser guardados en su carpeta correspondiente. Se trata de nombrarlos mediante una combinación de la fecha actual y una cadena generada aleatoriamente, manteniendo la extensión original. Se trata de un sencillo proceso que garantiza la unicidad del nombre del fichero.

Una vez almacenado el fichero en la carpeta que le corresponde según su autor, hay que actualizar la base de datos añadiendo un registro en la tabla *versiones* con el nombre del fichero. Además, si se trata de la primera versión de un documento también hay que insertar el registro correspondiente en la tabla *documentos* con los *metadatos* que el usuario ha introducido en el formulario para la subida de documentos.

La figura 3 muestra un diagrama de actividad modela dicho proceso.

Escenarios

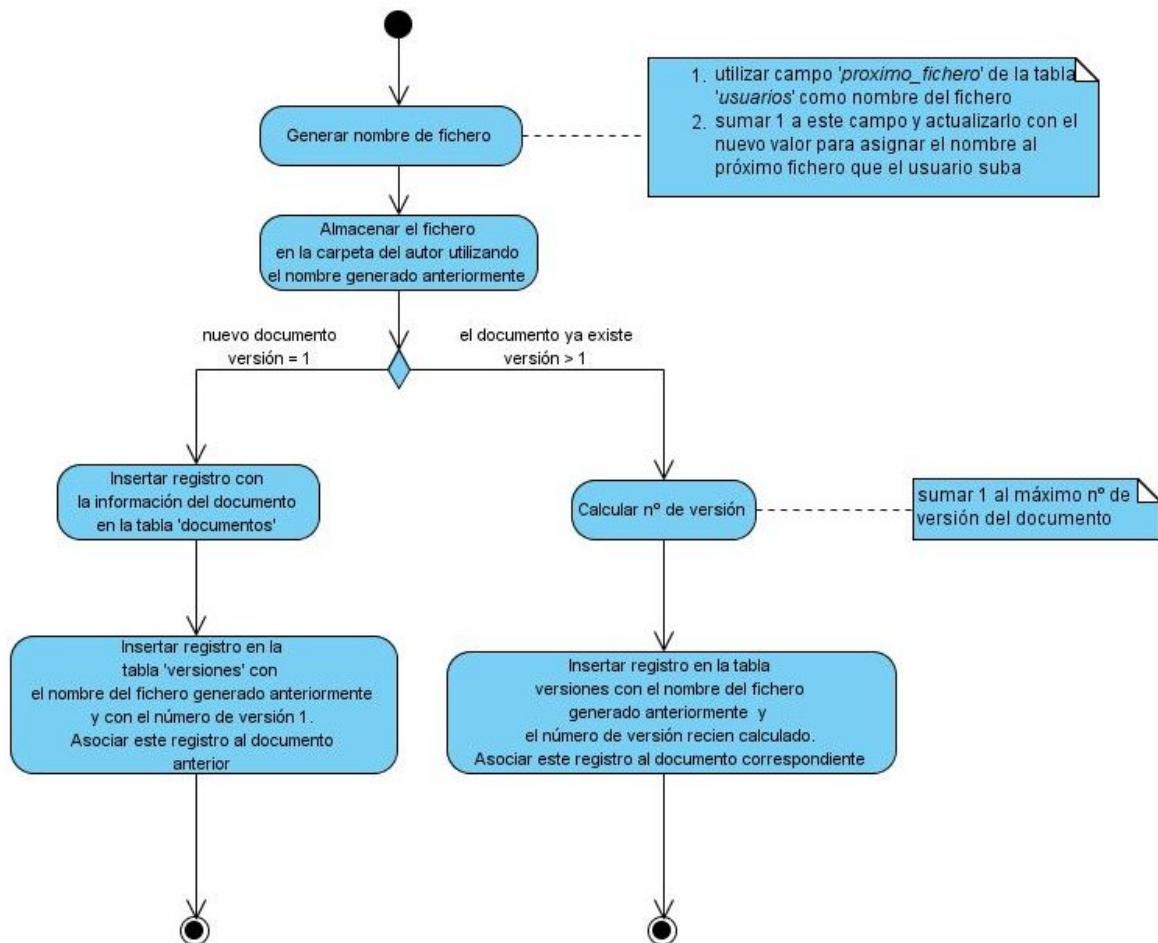


Figura 3. Diagrama de actividad del proceso de subida de archivos

Escenarios

En este apartado mostraremos mediante bocetos de pantallas, los distintos escenarios que presentará la aplicación.

Búsqueda y listado de documentos

Subida de documentos

Browser X

http://localhost/gestorDocumental/index.php/gesdoc/buscarDocumentos 🔍



[desconectar](#) José Valenzuela , perfil: autor

Buscar [Nuevo Documento](#) [Ayuda](#)

Búsqueda de documentos

título:	<input type="text" value="análisi%"/>	autor:	<input type="text" value="Diego Velázquez"/> ▾
categorías:	Combo	<input type="button" value="buscar >>"/>	
	técnicos		
	análisis		
	diseños		
	manuales		
	web		
	css		
	php		
	symfony		

Subida de documentos

Subida de nuevas versiones de los documentos

Browser X

http://localhost/gestorDocumental/index.php/gesdoc/nuevoDocumento



[desconectar](#)

José Valenzuela , perfil: autor

Buscar [Nuevo Documento](#) [Ayuda](#)

Nuevo Documento

título: es público:

descripción del documento

descripción de la primera versión

fichero: buscar fichero

categorías:

- técnicos
- análisis
- diseños
- manuales
- web
- css
- php
- symfony

Subida de nuevas versiones de los documentos

Modificación de los metadatos de un documento

Browser X

http://localhost/gestorDocumental/index.php/gesdoc/nuevaVersion/id/5 🔍



[desconectar](#)

José Valenzuela , perfil: autor

[Buscar](#) [Nuevo Documento](#) [Ayuda](#)

Nueva Versión

descripción:

Modificación de los metadatos de un documento

Añadir un comentario a una versión

Browser X

http://localhost/gestorDocumental/index.php/gesdoc/editarMetadatos/id/5 🔍



desconectar

José Valenzuela , perfil: autor

Buscar Nuevo Documento Ayuda

Modificar Metadatos Documento

título: es público:

descripción del documento:

categorías:

- técnicos
- análisis
- diseños
- manuales
- web
- css
- php
- symfony

Añadir un comentario a una versión

Valorar una versión.

Browser

http://localhost/gestorDocumental/index.php/gesdoc/nuevoComentario/id/5



GesDoc

desconectar

José Valenzuela , perfil: autor

Buscar Nuevo Documento Ayuda

Añadir Comentario

texto:

enviar

Valorar una versión.

Browser

http://localhost/gestorDocumental/index.php/gesdoc/valorarDocumento/id/5



GesDoc

desconectar

José Valenzuela , perfil: autor

Buscar Nuevo Documento Ayuda

Valorar

nota: 8

1
2
3
4
5
6
7
8
9
10

enviar

Gestión de Usuarios

Valorar una versión.

Listado

Browser X

http://localhost/gestorDocumental/backend.php/gesusu/index



Administración

[desconectar](#)

José Valenzuela , perfil: administrador

[Gestion Usuarios](#) [Gestion Tipos](#) [Gestión Categorias](#)

Listar Usuarios

[nuevo](#)

Nombre	Apellidos	Perfil	User Name	Acciones
Pedro	López	autor	pedro	Editar Borrar
Cristina	Martínez	lector	cristina	Editar Borrar
José	Valenzuela	administrador	jose	Editar Borrar

Edición/Creación

Browser X

http://localhost/gestorDocumental/backend.php/gesusu/new



Administración

[desconectar](#)

José Valenzuela , perfil: administrador

[Gestion Usuarios](#) [Gestion Tipos](#) [Gestión Categorias](#)

Editar Usuario

Nombre:

Apellidos:

Fecha Nacimiento:

Perfil:

Nombre Usuario:

Password:

Gestión de Categorías y Gestión de Tipos de Archivos

La gestión de categorías y de tipos de archivos tendrá un aspecto análogo a la gestión de usuarios, diferenciándose simplemente en que los datos que se manipulan son los de la tabla que corresponda.

Diseño arquitectónico.

El diseño arquitectónico de la aplicación está determinado por la arquitectura de *symfony*. En este apartado decidiremos como agrupar las distintas acciones de la aplicación en módulos y aplicaciones en el sentido que les da *symfony* y que ya hemos estudiado en el tema anterior.

Los requisitos de la aplicación sugieren la existencia de dos partes diferenciadas: por un lado tenemos el gestor documental en sí, cuyas funcionalidades pueden ser utilizadas en mayor o menor medida según el perfil que ostente el usuario, y por otro una parte de administración de entidades que utiliza dicho gestor documental, a saber: *usuarios*, *tipos de archivos permitidos* y *categorías*, y que sólo puede ser accedida por los usuarios administradores.

Esta división en dos partes; la aplicación en sí y la administración de la aplicación, es muy típica en el universo de la *web*, siendo los gestores de contenido los ejemplos más claros de aplicaciones *web* que presentan tal organización. De hecho existe una terminología indicada para designar a cada una de estas dos partes: a la parte de administración se le denomina **backend**, mientras que a la aplicación en sí, a la parte pública, entendiendo como tal la que puede ser usada por usuarios no administradores, se conoce como **frontend**.

Este hecho nos sugiere que organicemos los módulos en dos aplicaciones distintas que obviamente denominaremos *frontend* y *backend*. Como ya vimos en la unidad anterior, cada aplicación tiene su controlador frontal, su configuración, su *layout* y, por supuesto, sus módulos propios, aunque las dos operen sobre la misma base de datos que representa el nexo entre ambas.

Frontend: el gestor documental

Esta aplicación constará de un sólo módulo que vamos a llamar *gesdoc* donde se implementarán las acciones siguientes:

- buscar documentos
- listar documentos
- añadir nuevos documentos
- añadir nuevas versiones de documentos
- comentar documento
- valorar documento

Backend: administración del gestor documental

Esta aplicación albergará los módulos dedicados a la administración o gestión de las tablas *usuarios*, *tipos* y *categorías*. En todos los casos la gestión consiste en posibilitar al administrador la creación, recuperación, modificación y eliminación de registros. Estas son las operaciones típicas que ofrece cualquier *backend* sobre sus datos. De nuevo, dado que el patrón se repite extensivamente en multitud de aplicaciones, existe un término indicado para designar a este tipo de módulos: *CRUD*, que es un acrónimo de *Create*, *Retrieve*, *Update* y *Delete*, es decir, crear, recuperar, actualizar y borrar; las cuatro operaciones básicas que acabamos de proponer.

Otra de las herramientas que *symfony* ofrece es un generador automático de módulos *CRUD* sobre las tablas del proyecto. Con una sola instrucción la tarea *propel:generate-admin*

Inicio de sesión. Los plugins de symfony

construye un módulo completamente funcional mediante el cual podemos realizar las cuatro operaciones anteriores sobre la tabla que hayamos seleccionado. Modificando mínimamente el código generado automáticamente adaptaremos los módulos a nuestras necesidades.

Teniendo en consideración la existencia de esta potente herramienta, organizaremos la aplicación *backend* en tres módulos:

- gesusu: para la gestión de usuarios (*CRUD* sobre la tabla *usuarios*)
- gescat: para la gestión de categorías (*CRUD* sobre la tabla *categorias*)
- gestip: para la gestión de tipos de archivos (*CRUD* sobre la tabla *tipos*)

Estructuralmente los tres módulos serán idénticos e implementarán las siguientes operaciones:

- listar registros
- filtrar registros (búsqueda)
- modificar un registro determinado
- eliminar un registro determinado
- añadir un nuevo registro.

Inicio de sesión. Los plugins de symfony

Tanto para acceder a una y a otra aplicación; *frontend* y *backend*, es preciso autenticarse facilitando el nombre de usuario y contraseña, a excepción del acceso invitado al gestor documental (*frontend*) que permite realizar búsquedas y mostrar los resultados de documentos públicos. Es decir, se requiere alguna acción (o conjunto de acciones) que realice el proceso de autenticación y construya una sesión en el servidor con los datos persistentes relativos al usuario. De esa manera la aplicación sabrá qué puede permitir al usuario en función de su perfil, dónde debe almacenar los archivos que este envíe al repositorio y otras decisiones que dependen del usuario que realiza la petición.

El proceso que acabamos de esbozar se denomina **inicio de sesión**, y el conjunto de acciones, es decir el módulo, que lo implemente debería ser común a las dos aplicaciones. El problema es que, por lo que hasta ahora sabemos, cada aplicación tiene sus propios módulos, no hay módulos comunes a dos aplicaciones. Así pues la solución podría ser construir el módulo de inicio de sesión en una de las aplicaciones (*frontend*, por ejemplo) y copiarlo tal cual en la otra (*backend*, entonces). Pero esta solución, aunque funcionaría perfectamente, chirría a cualquier programador que respete ciertas normas básicas, conocidas como **buenas prácticas**, cuando realiza su trabajo. *Don't Repeat Yourself* (principio *DRY*) reza la norma que nos indica que la anterior solución no es del todo buena. No te repitas. Si más adelante haces un cambio en el módulo de inicio de sesión, resolviendo un *bug* por ejemplo, tendrás que propagarlo a su módulo gemelo una vez resuelto, con el coste de mantenimiento que eso conlleva.

De nuevo *symfony* nos ofrece una solución que encaja bien con las buenas prácticas de programación: los *plugins*. Esta facilidad permite extender las aplicaciones *symfony* con nuevas funcionalidades que han podido ser programadas, incluso, por terceros. Por lo general un *plugin* consiste en uno o más módulos autónomos que pueden ser utilizados y **compartidos** por las aplicaciones de un proyecto. Los *plugins* presentan la misma estructura que una aplicación, de hecho, la mayoría de los *plugins* comenzaron siendo aplicaciones que, dada su utilidad, fueron generalizándose y terminaron convertidas en *plugins*, dando servicio a la comunidad de usuarios *symfony*, ya que estas criaturas pueden enviarse a un repositorio común y pueden incorporarse a nuestros proyectos *symfony* a través de la tarea *plugin:install* de la instrucción *symfony*.

Resumen del diseño arquitectónico

Por tanto, además de las aplicaciones *frontend* y *backend*, construiremos un *plugin* consistente en un único módulo encargado de llevar acabo el inicio de la sesión. Las dos aplicaciones harán uso del mismo módulo de inicio de sesión. De esta manera hemos evitado duplicar el código. Denominaremos al *plugin* *IniSesPlugin*, y al módulo *inises*.

Resumen del diseño arquitectónico

En la tabla siguiente realizamos un resumen del diseño arquitectónico de nuestro gestor documental, es decir, la manera en que hemos organizado las acciones del proyecto en módulos, aplicaciones y *plugins*, que es básicamente lo que *symfony* nos permite decidir.

aplicación/plugin	Descripción	Módulos	Acciones
frontend(aplicación)	Es el gestor documental en sí, tal y como lo ven sus usuarios.	gesdoc	búscar documentos
			listar documentos
			añadir nuevos documentos
			añadir nuevas versiones de documentos
			comentar documento
			Valorar documento
backend (aplicación)	Es la aplicación que utiliza el administrador para configurar el gestor documental. Concretamente para administrar los usuarios, las categorías y los tipos de documentos que se pueden subir al sitio.	gesusu	Listar y buscar usuarios
			Modificar usuario
			Eliminar usuario
		gescat	Añadir nuevo usuario
			Listar, buscar, modificar, eliminar, añadir categorias
			Añadir nuevo tipo de
sesionPlugin(plugin)	Es un plugin con un módulo para realizar el inicio de sesión y que será utilizado por las dos aplicaciones anteriores.	inises	Comprobar autentificación del usuario
			Construir la sesión de usuario
			Cerrar la sesión

Recursos para la construcción de la aplicación

Aquí tienes disponible los siguientes recursos que necesitarás para seguir el resto del curso:

- Un volcado de la base de datos con la estructura y algunos datos de ejemplo.
- Algunos documentos que se corresponden con los ficheros asociados a las versiones de ejemplo de la anterior base de datos.
- Las CSS's que utilizaremos para presentar las pantallas de la aplicación de una manera similar a la propuesta en los bocetos de esta unidad.
- Las imágenes utilizadas por las CSS's y por algunas plantillas de la aplicación.

Conclusión

Resumen del diseño arquitectónico

En esta unidad hemos realizado un análisis de la aplicación cuya construcción nos servirá durante el resto del curso para aprender cómo utilizar *symfony* para construir aplicaciones *web* de *calidad*. Hemos planteado un catálogo de requisitos, un modelo de datos, un procedimiento para la subida de archivos y hemos propuesto unos bocetos de las pantallas de la aplicación. También hemos planteado la estructura organizativa del código en dos aplicaciones; el *frontend* que es la parte que se muestra a invitados, lectores y autores y que constituye la aplicación en sí, el *backend*, que es la parte de administración, y un *plugin* que implementará un procedimiento de inicio de sesión que será compartido por ambas aplicaciones.

Unidad 5: Profundizando en el modelo.

Unidad 5: Profundizando en el modelo.

Tras la pausa necesaria para definir la aplicación que vamos a construir, volvemos en esta unidad al dominio de *symfony*. Recordemos la estrategia de acercamiento en espiral que estamos siguiendo; podemos considerar que comenzamos la segunda vuelta. Nos acercamos al centro del problema, lo cual impone ampliar los detalles de algunos conceptos que ya se han tratado y a introducir algunos nuevos.

El desarrollo de aplicaciones se realiza añadiendo progresivamente, e incluso de manera incompleta, las funcionalidades requeridas de manera que, en ocasiones, una vez iniciado el trabajo, hay que volver a algún punto que ya se había comenzado a desarrollar para finalizarlo definitivamente. Este trabajo es parecido al del artesano que elabora una pieza de, digamos, alfarería; comienza por modelar el material de una forma aproximada según formas geométricas sencillas como esferas o cubos, y progresivamente, volviendo sobre las partes que ya comenzó, elimina o añade material para conferir al objeto la forma definitiva. Si esto hace el programador experto en una tecnología de desarrollo, con más razón será necesario proceder así en el desarrollo de un curso donde presentamos los conceptos de forma gradual, volviendo sobre ellos a medida que avanzamos.

Tomando como referencia el análisis de la unidad 4 y los recursos que allí se facilitaron, construiremos en primer lugar el proyecto, la aplicación *frontend* y su módulo *gesdoc* (vacío por lo pronto). A continuación enlazaremos con la base de datos y generaremos, mediante la herramienta *symfony*, el esquema, el modelo, los formularios y los filtros y ubicaremos en su lugar las *CSS*'s y el *layout* de la aplicación con su respectivo menú. Es decir, construiremos los pilares sobre los que se construirá la aplicación tal y como se estudió en la unidad 3.

En segundo lugar iniciaremos el desarrollo del módulo de gestión documental (*gesdoc*) de la aplicación *frontend*, implementando parcialmente las funcionalidades relativas a la generación de listados paginados y sus respectivos filtros de búsqueda (requisitos D.08 y D.09). Dicho trabajo nos obligará a profundizar en el modelo; concretamente en el estudio de la capa de abstracción de base de datos (*ORM*) que utiliza *symfony* para acceder a la misma y que ya hemos presentado y utilizado en la unidad 3.

Construcción del proyecto

Esta parte requiere pocas explicaciones; seguimos el procedimiento indicado en la unidad 3: Creamos el directorio accesible al servidor *web* donde se alojará el proyecto:

```
# mkdir /opt/lamp/htdocs/gestordocumental
```

Y generamos el proyecto en él:

```
# cd /opt/lamp/htdocs/gestordocumental
# symfony generate:project gestordocumental --orm=Propel
```

Definimos los parámetros de conexión a la base de datos en los ficheros *config/databases.yml* y *config/propel.ini*:

Fichero config/databases.yml

```
# You can find more information about this file on the symfony website:
# http://www.symfony-project.org/reference/1_4/en/07-Databases
```

```
dev:
  propel:
    param:
      classname: DebugPDO
    debug:
      realmemoryusage: true
      details:
        time: { enabled: true }
        slow: { enabled: true, threshold: 0.1 }
        mem: { enabled: true }
        mempeak: { enabled: true }
        memdelta: { enabled: true }

test:
  propel:
    param:
      classname: DebugPDO

all:
  propel:
    class: sfPropelDatabase
    param:
      classname: PropelPDO
      dsn: mysql:dbname=gestordocumental;host=localhost
      username: root
      password: root
      encoding: utf8
      persistent: true
      pooling: true
```

Líneas 6 - 9 del fichero config/propel.ini

```
propel.database.url      = mysql:dbname=gestordocumental;host=localhost
propel.database.creole.url = ${propel.database.url}
propel.database.user      = root
propel.database.password  = root
```

Generamos el *schema*, el modelo, los formularios y los filtros:

```
# symfony propel:build-schema
```

Nota

Debido a un error de *Propel* con el tratamiento de los tipos de datos *Timestamps* es necesario modificar en el archivo *config/schema.yml* que se acaba de generar las líneas que hacen referencia a este tipo de datos de la siguiente manera:

```
fecha_publicacion: { phpName: FechaSubida, type: TIMESTAMP, required: true, Value: CURRENT_TIMESTAMP }
```

Unidad 5: Profundizando en el modelo.

debe ser:

```
fecha_publicacion: { phpName: FechaSubida, type: TIMESTAMP, required: true }
```

y

```
fecha_subida: { phpName: FechaSubida, type: TIMESTAMP, required: true, Value: CURRENT_TIMESTAMP }
```

debe ser:

```
fecha_subida: { phpName: FechaSubida, type: TIMESTAMP, required: true }
```

```
# symfony propel:build-model  
# symfony propel:build-forms  
# symfony propel:build-filters
```

Generamos la aplicación y su módulo:

```
# symfony generate:app frontend  
# symfony generate:module frontend gesdoc
```

A continuación colocamos los archivos *admin.css*, *default.css* y *menu.css* que hemos construido en la sección de recursos de la unidad 4 en la carpeta *web/css* del proyecto, y los recursos de imágenes correspondientes en la carpeta *web/images*. De esta forma ya disponemos de los estilos y las imágenes que utilizaremos en la elaboración de las plantillas de nuestras aplicaciones. También debemos colocar, bajo el directorio *web/uploads*, los documentos de ejemplo correspondiente a las versiones de los documentos que han subido los usuarios. Para ello descomprime el archivo *documentos.zip* que encontrarás en los recursos de la unidad 4 en el directorio *web/uploads*.

Por último modificaremos el fichero *apps/frontend/templates/layout.php* de acuerdo al siguiente código que tienen en cuenta las CSS's anteriores para la aplicación de los estilos gráficos.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">  
    <head>  
        <?php include_http_metas() ?>  
        <?php include_metas() ?>  
        <?php include_title() ?>  
        <link rel="shortcut icon" href="/favicon.ico" />  
        <?php include_stylesheets() ?>  
        <?php include_javascripts() ?>  
    </head>  
    <body>  
        <div id="contenedor_general">  
            <div id="cabecera">  
                <div id="logo"></div>  
            </div>
```

```
        <div id="wrapper">  
            <div id="perfil"></div>
```

La capa de abstracción de base de datos en symfony

```
<div id="menuprincipal"></div>
<div id="sf_admin_container"><?php echo $sf_content ?></div>
</div>
<div class="PiePagina">
    <ul>
        <li><a href="#" title="Aviso legal" target="">Aviso legal</a>|</li>
        <li><a href="http://www.w3.org/WAI/" title="Accesibilidad" target="">Accesibilidad</a>|</li>
        <li>
            <a href="http://www.w3.org/WAI/" title="Logo de la WAI" target="">
                <?php echo image_tag('valid-xhtml10.png', array('alt' => 'Accesibilidad web', 'width' => '50'))?>
            </a>
        </li>
    </ul>
    <p>
        <a href="#" title="© Juan David Rodríguez" target="">© Juan David Rodríguez</a>
        <br/>
        <a href="#" title="Mentor Soft" target="_blank">Mentor Soft</a>
    </p>
    <p>
        Información general : <a href="mailto:#" title="Contacte con el webmaster">webmaster at gmail dot com</a>
        <br/>
    </p>
</div>
</div>
</body>
</html>
```

Ya sólo nos queda indicar en el ficheros *apps/frontend/config/view.yml* las hojas de estilo que deseamos utilizar para ver el resultado de lo que vamos construyendo con estilos gráficos aplicados.

```
default:
    http_metas:
        content-type: text/html

metas:
    title: Gestor Documental
    description: Un gestor documental construido con symfony para un curso de Mentor
    keywords: symfony, gestor_documental, mentor
    language: es
    robots: index, follow

stylesheets: [default.css, admin.css, menu.css]
javascripts: []
has_layout: true
layout: layout
```

La capa de abstracción de base de datos en symfony

La organización de una aplicación según el patrón *MVC* implica agrupar en el modelo aquellos componentes que tengan que ver con la lógica de negocio del dominio en que la aplicación se desarrolla. En las aplicaciones *web* gran parte de los procesos que conforman la lógica de negocio requieren del acceso a una o varias bases de datos, por ello el modelo está estrechamente relacionado con los componentes que se utilicen para realizar dicho acceso. Por otro lado *symfony* es un *framework* orientado a objetos, por lo que es obvio que utilice una capa orientada a objetos para la manipulación de las bases de datos: *la capa de abstracción de base de datos*. A partir de ahora nos referiremos a ella con el término anglosajón *ORM*, *Object Relational Mapping*, algo así como mapeado de bases de datos relacionales a objetos. Y es que los sistemas gestores de base de datos más utilizados (*MySQL*, *PostgresSQL*, *Oracle*) utilizan el modelo relacional para la administración de los datos. El mapeado relacional consiste en abstraer las entidades de la base de datos (tablas, vistas y atributos) en objetos que las representan.

La gran ventaja del *ORM* es la reutilización de los objetos en distintas partes de la aplicación e incluso en distintas aplicaciones, proporcionando un modelo que encaja perfectamente en un desarrollo orientado a objetos. La manera de recuperar, introducir y modificar datos es

Generación del ORM de Propel (Modelo)

independiente del sistema gestor de base de datos que utilicemos; la sintaxis utilizada es siempre la misma y es el propio *ORM* quien construye la *query (SQL)* adecuada al sistema gestor de base de datos que en cada momento utilice la aplicación. Esto proporciona una gran portabilidad a nuestros proyectos, ya que si deseamos hacerlos funcionar con otro sistema de base de datos distinto bastará con indicarlo en los ficheros de configuración del *ORM*.

Symfony incorpora los dos *ORM* más potentes que actualmente se ofrecen en el mundo del *PHP*: *Propel* y *Doctrine*. En este curso utilizaremos el primero de ellos. La razón de esta elección no es que *Propel* sea más potente que *Doctrine*, sino que fue el primero que se acopló a *symfony* y es el que mejor conocemos. De hecho, a la luz de las numerosas comparativas que presentan los *blogs* dedicados al mundo del desarrollo de aplicaciones *web*, podemos concluir que ambos *ORM* están al mismo nivel. También podemos decir que una vez que se conoce uno de ellos, migrar al otro no es una tarea complicada.

En la unidad 3 ya hicimos uso de las clases de *Propel* para recuperar e insertar registros en la base de datos, pero nos acercamos a ellas de una manera puramente operativa, describiendo muy resumidamente, más que su funcionamiento, su uso. En este apartado presentaremos una explicación que, aunque sigue estando centrada en lo operativo, criterio principal utilizado en el desarrollo de este curso, profundiza lo suficiente para que el programador se sienta cómodo con *Propel*.

Generación del ORM de Propel (Modelo)

Ya sabemos que el modelo de *Propel* se genera automáticamente mediante la tarea `generate:model` de *symfony*, y que se ubica en el directorio *lib/model* del proyecto. Ahora explicaremos el proceso de generación. Cada base de datos que utilice nuestro proyecto debe tener asociada una **conexión**, la cual es un conjunto de parámetros definidos bajo un mismo nombre en el archivo *config/databases.yml*. Por defecto el nombre de la conexión es *propel*, como puedes ver indicado en el siguiente código. Si, por ejemplo necesitamos utilizar otra base de datos adicional en el proyecto debemos añadir sus datos de conexión como se indica en el texto sombreado:

Ten en cuenta que el código anterior es un ejemplo para la explicación y no debes añadirlo al proyecto.

Por otro lado la descripción de cada base de datos (sus tablas, atributos, y referencias) se define en los ficheros *schema*. Lo normal es trabajar con tantos ficheros *schema* como conexiones se hayan definido en el *databases.yml*; si únicamente tenemos una conexión el fichero *schema* se denominará *schema.yml*, si son varias las conexiones cada fichero se denominará según el siguiente patrón: *{nombreconexion}.schema.yml*.

Cada fichero *schema* define en su primera línea el nombre de la conexión asociada a la base de datos que describe. Además podemos indicar en los ficheros *schema* el directorio donde deseamos que se almacene el modelo que va a generarse a partir de ellos a través de la directiva *package* (mira el fichero *schema.yml* de tu proyecto). Por defecto su valor es *lib.model*, por eso se genera el modelo en el directorio *lib/model*, pero podemos cambiar esta ubicación, lo cual nos permitirá, si trabajamos con varias bases de datos, organizar los modelos respectivos en distintas ubicaciones. Incluso con un poco de imaginación podemos generar varios modelos y ubicarlos en distintos paquetes (directorios donde se ubican los modelos) a partir de una única base de datos. Para ello podemos definir dos conexiones con nombres distintos pero con los mismos parámetros de conexión, crear dos archivos *schema* (por ejemplo *modelo1.schema.yml* y *modelo2.schema.yml*) cada uno con una parte de la base de datos haciendo referencia a la conexión correspondiente y definiendo la directiva *package* con la ubicación donde deseamos colocar el modelo generado. Todo esto puede venir muy bien para organizar proyectos con bases de datos con muchas tablas o con varias bases de datos.

Jerarquía de clases del modelo con Propel.

No obstante en la mayoría de los proyectos bastará con utilizar una sola base de datos, una única conexión y un único modelo ubicado en el directorio por defecto *lib/model*.

Volvamos al proceso de generación del modelo. Con las conexiones definidas en *databases.yml* y los *schemas* que definen las entidades de la base de datos, *symfony* genera a través de la tarea *generate:model* un conjunto de clases que el programador utilizará para realizar operaciones con las bases de datos.

Jerarquía de clases del modelo con Propel.

Veamos ahora la pinta que tiene el modelo generado en los directorios que hayamos especificado con la directiva *package* de los ficheros *schemas*. Cada tabla da lugar a 5 clases definidas en otros tantos archivos. Para hacer la explicación más sencilla y concreta tomemos la tabla *usuarios* de nuestro proyecto como ejemplo. Bajo el directorio *lib/model* vemos que se han generado los siguientes archivos asociados a esta tabla:

- *Usuarios.php*
- *om/BaseUsuarios.php*
- *UsuariosPeer.php*
- *om/BaseUsuariosPeer.php*
- *map/UsuariosMapBuilder.php*

El último fichero podemos obviarlo pues es utilizado internamente por *Propel* y el programador no lo necesita para su trabajo.

Si echamos un vistazo al fichero *Usuarios.php* veremos que define la clase *Usuarios* que deriva de la clase *BaseUsuarios* la cual, como podrás adivinar y comprobar se define en el fichero *om/BaseUsuarios.php*. Además puedes ver que la clase *Usuarios* no implementa ningún método, simplemente realiza la derivación que hemos indicado. Por lo tanto, en un principio, la clase *Usuarios* es equivalente a *BaseUsuarios*, su clase base. Esta última sí que está repleta de métodos que podremos utilizar en el desarrollo de nuestro proyecto. Lo mismo ocurre con el fichero *UsuariosPeer.php* y *om/BaseUsuariosPeer.php*. La razón de esta aparente complejidad es que las clases hijas están pensadas para que el programador las utilice en lugar de las clases bases y, además, si lo requiere, defina en ellas nuevos métodos que modelen la lógica particular del negocio que anda desarrollando. ¿Vale y qué? preguntarás, ¿no podemos definir, si fuese necesario, los nuevos métodos directamente en las clases bases?. Pues sí, podríamos hacerlo, pero si más adelante, por la razón que fuese, modificamos y/o añadimos nuevos campos de la tabla tendríamos que volver a generar el modelo, y en esa regeneración iPERDERÍAMOS los métodos que hubiésemos añadido a la clase base!, ya que volver a generar el modelo implica volver a generar los archivos de las clases bases. Sin embargo, si nuestros nuevos métodos los implementamos en las clases hijas, este desastre no ocurrirá, ya que estas clases, una vez generadas la primera vez que se invoca la tarea *generate:model*, no vuelven a ser tocadas por dicha tarea en lo sucesivo. De esta forma podemos extender el modelo a nuestro antojo y regenerarlo tantas veces como sea necesario sin que perdamos nuestras extensiones. En realidad se trata de una organización de los métodos de las clases; por una parte tenemos los métodos generados automáticamente a partir de la definición de la tabla que son implementados en la clase base, y por otro lado los métodos propios del programador que los define en la clase hija. Así pues, y como la teoría de la orientación a objetos enseña, la clase hija comprende a la clase base y la amplía.

Aclarado este importante detalle acerca de la organización de los métodos, veremos ahora qué representan las clases *Usuarios* y *UsuariosPeer*. La primera de ellas, *Usuarios*, es una clase instanciable, es decir, que se pueden declarar variables que son objetos de dicha clase. Cada objeto así declarado **representa un registro de la tabla usuarios** y podemos acceder y modificar sus valores mediante una serie de métodos, denominados *getters* y

Métodos de las clases registro

setters, que describiremos en el próximo apartado.

La segunda, *UsuariosPeer*, es una clase estática, lo cual significa que no puede ser instanciada, es decir, no se pueden definir variables que sean objetos de dicha clase. Este tipo de clases se puede concebir como un conjunto de funciones (sus métodos), es decir como una librería. La funcionalidad de esta clase es proporcionar métodos para recuperar registros de la tabla *usuarios* que satisfagan criterios definidos por el programador, en la forma de objetos *Usuarios*. Una vez que dispongamos de dichos objetos podremos manipularlo a través de sus métodos.

Obviamente, todo lo que hemos contado acerca de las clases *Usuarios* y *UsuariosPeer*, que representan el *ORM* de la tabla *usuarios*, es válido para el resto de tablas de la base de datos. A partir de ahora llamaremos **clases registro** a las clases del modelo que representan registros de las tablas, y **clases peer** a las clases estáticas que implementan los métodos para recuperar registros encapsulados como objetos.

Métodos de las clases registro

Antes de nada hay que indicar que la mejor forma de saber qué métodos implementa una determinada clases es viendo su código fuente y la documentación correspondiente. Tanto en este apartado como en el siguiente mostraremos los métodos más relevantes de las clases de *Propel*. El estudiante interesado en profundizar siempre puede analizar el código generado y complementar el estudio con las referencias sugeridas.

Como cualquier objeto en *PHP*, los objetos de las clases registro se instancian de la siguiente manera:

```
$usuario = new Usuarios();
```

La línea anterior declara la variable *\$usuario* como un objeto (o instancia) de la clase *Usuarios*, el cual, como ya se ha dicho antes, representa un registro de la tabla *usuarios*. Por lo pronto es un registro inexistente en dicha tabla, de hecho aún es un objeto vacío de contenido. ¿Qué podemos hacer con él?

Definir sus atributos

Mediante los métodos llamados *setters* podemos definir el valor de los atributos del objeto. Estos atributos coinciden con los campos de la tabla que representa la clase. Siguiendo con nuestro ejemplo, los atributos del objeto *\$usuario* serían *id_usuario*, *nombre*, *apellidos*, *username*, *password* y *perfil*, sin embargo, como dicta la norma de la encapsulación, el acceso a los miembros del objeto debe realizarse a través de métodos construidos para tal fin. Los métodos que sirven para definir valores se denominan *setters*, y los que se utilizan para recuperar valores *getters*. Así pues podemos dotar de contenido al objeto que hemos construido de la siguiente manera:

```
$usuario → setNombre('Anselmo');  
$usuario → setApellidos('González García');  
$usuario → setUsername('anselmo');  
$usuario → setPassword('s8udR3');  
$usuario → setPerfil('administrador');
```

Es decir los métodos *setters* se nombran anteponiendo el prefijo *set* al nombre del campo reescrito mediante la notación *UpperCamelCase*, lo cual significa colocar en mayúsculas la primera letra del nombre del campo y las letras que van tras el carácter *'_'* (*underscore*), eliminando de la cadena este último.

Métodos de las clases peer y el objeto Criteria.

Tras escribir el código anterior tenemos un objeto con sus atributos definidos. Sin embargo **aún no existe como registro** en la base de datos.

Obtener el valor de los atributos

Ahora podemos obtener los valores de los campos mediante los métodos *getters*:

```
$nombre    = $usuario → getNombre();  
$apellidos = $usuario → getApellidos();  
$userName = $usuario → getUsername();  
$password  = $usuario → getPassword();  
$perfil    = $usuario → getPerfil();
```

El nombre de los métodos *getters* se construye de la misma manera que el de los *setters* pero utilizando el prefijo *get* en lugar de *set*.

Insertar en la base de datos un nuevo registro

Cuando queramos grabar en la base de datos la información que contiene el objeto utilizamos el método *save()*:

```
$usuario → save();
```

Una vez invocado este método, el *ORM* crea un nuevo registro en la tabla *usuarios*, y por tanto se asigna un valor numérico al campo correspondiente a la clave principal, *id_usuario* en el ejemplo. Esta operación se corresponde con una inserción en la base de datos (*INSERT*).

Actualizar un registro existente

Ahora si hacemos:

```
id_usuario = $usuario → getIdUsuario();
```

obtenemos el valor que el sistema gestor de base de datos ha asignado a la clave principal del registro insertado. Dicho valor no está disponible hasta que no se graba en la base de datos mediante *save()*.

Podemos volver a cambiar los valores de los atributos mediante los métodos *getter*, pero recuerda que solo se grabarán en la base de datos cuando se invoque el método *save()*. Si cambiamos algún atributo del objeto mediante los *getters* correspondientes y volvemos a utilizar el método *save()*, se actualiza en la base de datos el registro en cuestión. Por tanto, en esta situación, el método *save()* se corresponde con una operación de actualización (*UPDATE*) en la base de datos.

Métodos de las clases peer y el objeto Criteria.

Ya te habrás dado cuenta de que nos faltan dos operaciones fundamentales de acceso a base de datos; la recuperación de registros (*SELECT*) y su eliminación (*DELETE*). Las clases peer definen una serie de métodos estáticos que, en combinación con el objeto *Criteria*, nos proporcionan colecciones (*arrays*) de objetos-registro que satisfacen un criterio dado y otros métodos para realizar la eliminación de registros. También constituyen en sí mismas una descripción de las tablas que representan gracias a la definición de constantes que representan los nombre de los campos de la tabla que modelan. Estas constantes, como veremos en breve, son muy utilizadas para construir los criterios de selección. Así por

Métodos de las clases peer y el objeto Criteria.

Por ejemplo el nombre de los campos de la tabla *usuarios* puede accederse mediante las siguientes constantes:

- *UsuariosPeer::ID_USUARIO*
- *UsuariosPeer::NOMBRE*
- *UsuariosPeer::APELLOS*
- *UsuariosPeer::USERNAME*
- *UsuariosPeer::PASSWORD*
- *UsuariosPeer::PERFIL*

Presentaremos ahora las operaciones más corrientes que se realizan para recuperar registros en forma de objetos de *Propel*.

Obtener un registro específico

El método *retrieveByPK()* nos permite recuperar un registro cuya clave principal se conoce:

```
$usuario = UsuariosPeer::retrieveByPK(21);
```

La línea anterior construye un objeto de la clase *Usuarios* a partir del registro nº 21 de la tabla *usuarios* siempre que dicho registro exista, en caso contrario la variable *\$usuario* sería definida como *null*. A partir de este momento podemos obtener los valores de sus campos, modificarlos y actualizarlos en la base de datos según hemos visto en el apartado anterior.

Obtener todos los registros de una tabla

El objeto *Criteria* modela la parte *WHERE* de una consulta *SQL*. La siguiente línea declara un objeto *Criteria* vacío:

```
$c = new Criteria();
```

Dicho objeto se utiliza como argumento de la función *doSelect()* de las clases peer para recuperar los objetos que satisfacen un criterio determinado. Si utilizamos un criterio vacío como el anterior obtendremos todos los registros de la tabla:

```
$usuarios = UsuariosPeer::doSelect($c);
```

La línea anterior define un *array* de objetos *Usuarios* con todos los registros de la tabla *usuarios*. Como es un *array*, puede ser iterado mediante las instrucciones que *PHP* ofrece para la iteración, siendo *foreach* la más usada con diferencia:

```
foreach($usuarios as $u)
{
    echo '<b>' . $usuario -> getNombre() . ' ' . $usuario -> getApellidos(). '</b>';
    echo '<br/>';
}
```

El código anterior imprimiría en el navegador una lista con los nombres y apellidos de todos los usuarios en negrita.

Obtener un subconjunto de registros que satisface un criterio

Métodos de las clases peer y el objeto Criteria.

Se haría como en el ejemplo anterior, mediante la función `doSelect()`, la diferencia es que el criterio (`$c`) que pasamos por argumento no está vacío y modela un filtro (*WHERE*) de selección de registros. La definición del criterio se realiza utilizando los métodos del objeto *Criteria*. Aunque son muchos, en la práctica se puede hacer casi de todo con el subconjunto que a continuación describiremos.

El método `add()` del objeto *Criteria* se utiliza para añadir condiciones de búsqueda sobre la tabla especificada en su primer argumento:

```
$c = new Criteria();
$c → add(UsuariosPeer::NOMBRE, 'Anselmo');

$usuarios = UsuariosPeer::doSelect($c);
```

El código anterior declara un objeto *Criteria* y le añade la condición de que el campo nombre de la tabla *usuarios* sea igual '*Anselmo*'. De esa manera, la variable `$usuarios` recibirá a través de la función `doSelect()` de la clase *UsuariosPeer* un array con todos los objetos usuarios cuyo nombre es igual a '*Anselmo*'.

Podemos realizar otras operaciones de comparación que tienen correspondencia en las condiciones de la cláusula *WHERE* de una consulta *SQL*. Las más usuales son *LIKE*, *NOT_EQUAL*, *NOT_LIKE*, *IS_NULL*, *IS_NOT_NULL*.

Así por ejemplo, si en el código anterior sustituimos la segunda línea por la siguiente:

```
$c → add(UsuariosPeer::NOMBRE, 'An%', Criteria::LIKE);
```

El array `$usuarios` contendría una colección de objetos *Usuarios* cuyo nombre satisface el patrón de búsqueda '*An%*', teniendo el carácter '%' el mismo significado que en las consultas *SQL*.

Si utilizamos varias veces el método `add()`, el criterio final será el producto lógico (*AND*) de cada uno de las comparaciones:

```
$c = new Criteria();

$c → add(UsuariosPeer::NOMBRE, 'An%', Criteria::LIKE);
$c → add(UsuariosPeer::PERFIL, 'lector');

$usuarios = UsuariosPeer::doSelect($c);
```

Con este código obtenemos todos los usuarios que tienen el perfil *lector* y cuyo nombre comienza por '*An*'.

Si queremos realizar sumas lógicas (*OR*), utilizamos el método `addOr()` en lugar de `add()`. Su forma de uso es exactamente la misma.

También podemos añadir uniones (*joins*) con otras tablas para recuperar registros. Esto se hace añadiendo al criterio condiciones de unión mediante el método `addJoin()`:

```
$c = new Criteria();

$c → add(UsuariosPeer::NOMBRE, 'An%', Criteria::LIKE);
```

Métodos de las clases peer y el objeto Criteria.

```
$c → addJoin(UsuariosPeer::ID_USUARIO, DocumentosPeer::ID_USUARIO);  
$documentos = DocumentosPeer::doSelect($c);
```

Con este código obtendríamos en \$documentos un array con todos los objetos *Documentos* que pertenecen a los usuarios cuyo nombre satisface el patrón de búsqueda 'An%'.

De la misma manera que podemos recuperar registros mediante el método *doSelect()* de las clases peer, podemos borrarlos con el método *doDelete()*:

```
$c = new Criteria();  
$c-> add(UsuariosPeer::NOMBRE, 'An%', Criteria::LIKE);  
UsuariosPeer::doDelete($c);
```

El código anterior eliminaría de la tabla *usuarios* todos los registros cuyo nombre satisface el patrón de búsqueda 'An%'.

Recuperación de objetos relacionados con otros objetos.

Cualquier base de datos, por muy sencilla que sea, define relaciones entre sus tablas que permiten relacionar registros de unas tablas con otras mediante claves foráneas. En el dominio del ORM decimos que los objetos de una clase (registros) pueden estar relacionados con los de otra. Por ello cada objeto-registro, implementa unos métodos que nos permiten obtener todos los objetos que con él se relacionan.

Cuando una tabla contiene un índice que se relaciona con la clave principal de otra tabla, cada objeto de la primera tiene asociado un objeto de la segunda. Por ejemplo, en nuestra base de datos cada registro de la tabla *documentos* tiene asociado un registro de la tabla *usuario*, lo que equivale a decir que cada objeto de la clase *Documentos* tiene asociado un objeto de la clase *Usuarios*. Podemos obtener el objeto *Usuarios* asociado a un objeto *Documentos* dado de la siguiente manera:

```
$documento = DocumentosPeer::retrieveByPk(2);  
$usuario = $documento → getUsuarios();
```

El código anterior recupera el documento con clave principal nº 2, y posteriormente obtiene su usuario asociado. Es decir, los objetos que están en relación N - 1 con otros objetos, implementan un *getter* de la forma *get{NombreClaseRelacionada}()*, para recuperar su objeto asociado.

También podemos estar en la situación opuesta, un objetos tiene otros muchos objetos relacionados; se trata de la relación contraria 1 - N. Por ejemplo un objeto *Usuarios* puede tener asociados muchos objetos *Documentos*. La forma de obtenerlos sería la siguiente:

```
$usuario = UsuariosPeer::retrieveByPK(5);  
$documentos = $usuarios → getDocumentoss();
```

Implementación del listado documentos

El código anterior recupera el usuario con clave principal nº 5 y posteriormente obtiene un array (`$documentos`) con todos los objetos *Documentos* asociados a él. Es decir, los objetos que están en relación 1 - N con otros objetos, implementan un *getter* de la forma `get{NombreClaseRelacionada}s()`, para recuperar un array con sus objetos asociados.

De esta manera podemos navegar por toda la estructura de datos, de objeto en objeto a través de sus relaciones, y obtener todos los atributos referidos directa e indirectamente con él.

Son muchas más las funcionalidades y métodos que ofrece *Propel*. No obstante paramos aquí, por lo pronto, el estudio teórico del este *ORM*. Con lo dicho hasta el momento se cubre una amplia gama de problemas reales. Remitimos al estudiante a los recursos que indicamos sobre *Propel* para profundizar en su estudio y para utilizarlo como referencia en caso de necesitar nuevas características del mismo. Así pues continuamos la implementación de nuestro gestor documental.

Nota

Recursos sobre Propel:

Sitio oficial:

<http://www.propelorm.org/>

Chuleta sobre el objeto *Criteria*:

http://www.cheat-sheets.org/saved-copy/sfmodelcriteriacriterionrsrefcard_enus.pdf

Chuleta sobre el *Schema*:

<http://www.cheat-sheets.org/saved-copy/sfmodelsecondpartrefcard.pdf>

Implementación del listado documentos

Una vez que tenemos montado el marco de nuestro cuadro ya podemos empezar a pintarlo; es la hora de picar el código propio de nuestra aplicación sobre los pilares del proyecto que acabamos de definir. La primera funcionalidad que implementaremos será la generación de listados de documentos filtrados mediante los campos de un formulario de búsqueda.

Listado de todos los documentos

El listado de documentos forma parte del módulo *gesdoc* de la aplicación *frontend*. Llamaremos *index* a la acción encargada de mostrar dicho listado. La elección del nombre no es arbitraria; ya se ha dicho en otra ocasión que *symfony* interpreta como acción por defecto de cualquier módulo la que se llame de esta manera. Como en el caso del gestor documental el listado de documento es la acción de inicio del módulo, pues su nombre está cantado.

Por lo pronto, la acción de inicio (*index*), recogerá todos los documentos, y su plantilla asociada, *indexSuccess.php*, los pintará en una tabla diseñada según el boceto que se ha planteado en el análisis de la aplicación. Para recoger todos los documentos debemos acceder a la base de datos, es decir, debemos usar los objetos de *Propel* tal y como se ha explicado en esta misma unidad: creamos un criterio vacío y lo pasamos como argumento al método **doSelect()* de la clase peer correspondiente a la tabla *documentos*.

Esto es, añadimos al fichero de acciones del módulo el siguiente código:

Trozo del archivo: *apps/frontend/modules/gesdoc/actions/action.class.php*

Implementación del listado documentos

```
public function executeIndex(sfWebRequest $request)
{
    $c = new Criteria();

    $this -> documentos = DocumentosPeer::doSelect($c);
}
```

el atributo `$this -> documentos` recibe un *array* con todos los objetos *Documentos* existentes, y estará disponible en la plantilla correspondiente para ser iterado y mostrar los atributos de los documentos que nos interese. Agregamos el siguiente código a la plantilla *indexSuccess.php*:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/indexSuccess.php*

```
<table>
    <thead>
        <tr>
            <th>Título</th>
            <th>Autor</th>
            <th>Versiones</th>
            <th>Acciones</th>
        </tr>
    </thead>

    <tbody>
        <?php foreach ($documentos as $d): ?>
        <tr>
            <td><?php echo $d -> getTitulo() ?></td>
            <td></td>
            <td></td>
            <td></td>
        </tr>
        <?php endforeach; ?>
    </tbody>
</table>
```

Ya puedes probar desde el navegador el resultado de esta acción:

http://localhost/gestordocumental/web/frontend_dev.php/gesdoc1

Y verás una tabla (aún por completar) con el título de todos los documentos, los cuales se han obtenido en la plantilla accediendo a cada uno de los objetos Documentos del array `$documentos` y pidiendo su título a través de su getter `getTitulo()`.

Para completar la tabla debemos acceder a los autores y versiones de cada documento. Según nuestro modelo de datos, Usuarios (que pueden ser autores) y Versiones son objetos (registros) relacionados con los documentos de forma que un documento sólo puede tener un autor, y puede tener asociadas varias versiones. Se trata por tanto de aplicar los métodos de acceso explicados en el apartado 2.5, para realizar dicha tarea.

Si `$d` representa un objeto Documentos, entonces:

```
$autor = $d -> getUsuarios();
```

`$autor` es un objeto Usuarios asociado al documento y `$versiones = $d -> getVersioness();`

Implementación del listado documentos

\$versiones es un array de objetos Versiones asociados al documento.

Con esto podemos mostrar los datos que nos faltan en las columnas:

Contenido del archivo: apps/frontend/modules/gesdoc/templates/indexSuccess.php

```
<table>
    <thead>
        <tr>
            <th>Título</th>
            <th>Autor</th>
            <th>Versiones</th>
            <th>Acciones</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($documentos as $d): ?>
        <tr>
            <td><?php echo $d -> getTitulo() ?></td>
            <td><?php echo $d -> getUsuarios() -> getNombre().' '. $d -> getUsuarios() -> getApellidos()?></td>
            <td>
                <?php foreach ($d -> getVersioness() as $v): ?>
                <?php echo $v -> getNumero() ?> |
                <?php endforeach; ?>
            </td>
            <td>
                <?php echo link_to('modificar', 'gesdoc/modificar?id_documento='.$d -> getIdDocumento()) ?> |
                <?php echo link_to('subir versión', 'gesdoc/subirVersion?id_documento='.$d -> getIdDocumento()) ?>
            </td>
        </tr>
        <?php endforeach; ?>
    </tbody>
</table>
```

Es decir, por un lado accedemos desde cada documento a su usuario asociado, y extraemos el nombre y los apellidos, y por otro accedemos al array con sus versiones asociadas, lo iteramos y extraemos los números de versión. Además se ha completado el código de las acciones mediante los enlaces correspondientes a la edición del documento y a la subida de nuevas versiones. Por su puesto estas acciones aún no existen, pero ya hemos decidido sus nombres y módulo (*gesdoc/modificar* y *gesdoc/subirVersion*) en esta etapa.

Los enlaces de las acciones se han construido utilizando una función denominada *link_to()*, la cual construye el código *HTML* del enlace con el texto que se indique en su primer argumento y la *URL* del módulo y acción adaptada al servidor donde se ejecuta la aplicación, en su segundo argumento. De esta manera cuando cambiamos la aplicación de servidor y/o ubicación, no tendremos que cambiar todos los enlaces de las acciones, ya que *symfony* construirá la *URL* correcta mediante dicha función. La función *link_to()*, al igual que *url_for()* que vimos en la unidad 3, es lo que se denominan un *helper* de *symfony*.

Aprovecharemos este momento para ilustrar como, extendiendo el modelo, podemos simplificar el código y hacerlos más legible y reutilizable. Fíjate que el acceso al nombre y apellido del usuario se realiza mediante el siguiente código:

```
<?php echo $d -> getUsuarios() -> getNombre().' '. $d -> getUsuarios() -> getApellidos()?>
```

Es decir, tenemos que usar dos veces el objeto para mostrar el nombre y los apellidos. Sería todo más sencillo si el objeto *Usuarios* ofreciese un método que devolviese el nombre y los apellidos de una vez. Pero ya sabemos que *Propel* nos brinda la posibilidad de extender los objetos a través de las clases hijas. Así pues podemos implementar un método en la clase *Usuarios* (archivo *lib/model/Usuarios.php*), que se llame, por ejemplo, *dameNombreYApellidos()*, y que realice dicha labor:

Función añadida al archivo: *lib/model/Usuarios.php*

```
public function dameNombreYApellidos()
{
```

Implementación del listado documentos

```
        return self::getNombre() . ' ' . self::getApellidos();  
    }
```

Ahora podemos sustituir en la plantilla la farragosa línea anterior por la siguiente que es mucho más legible:

```
<?php echo $d -> getUsuarios() -> dameNombreYApellidos() ?>
```

Pero podemos ir incluso más lejos. Y es que *PHP* está construido de tal manera que si el programador implementa en sus objetos un método denominado *_toString()*, que devuelva una cadena de texto, se puede enviar dicho objeto directamente a la salida estándar (la pantalla) mediante la instrucción *echo*. Por ello, si cambiamos de nombre al método *dameNombreYApellidos()* y lo llamamos *_toString()*, la línea de la plantilla donde se pinta el nombre y el apellido del usuario quedaría reducida a:

```
<?php echo $d -> getUsuarios() ?>
```

Como veremos más adelante cuando estudiemos los formularios y la generación automática de módulos *CRUD* de *symfony*, es muy importante que los objetos de *Propel* implementen el método *_toString()*. Y esta tarea la debemos realizar manualmente, ya que el *framework* no puede adivinar cómo queremos mostrar por pantalla nuestros objetos.

Por último, adaptamos la plantilla a los estilos que estamos utilizando en nuestro proyecto (*CSS's*), y nos queda el siguiente código para el fichero *indexSuccess.php* (atención!, no olvides implementar el método *_toString()* de la clase *Usuarios*):

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/indexSuccess.php*

```
<div id="sf_admin_header">  
    <h2>Listado de documentos</h2>  
    <div class="notice">Mensaje de advertencia(que modificaremos más tarde)</div>  
</div>  
<div id="sf_admin_content">  
    <div id="sf_admin_list">  
        <table>  
            <thead>  
                <tr>  
                    <th>Título</th>  
                    <th>Autor</th>  
                    <th>Versiones</th>  
                    <th>Acciones</th>  
                </tr>  
            </thead>  
            <tbody>  
                <?php foreach ($documentos as $d): ?>  
                <tr>  
                    <td><?php echo $d -> getTitulo() ?></td>  
                    <td><?php echo $d -> getUsuarios() ?></td>  
                    <td>  
                        <?php foreach ($d -> getVersioness() as $v): ?>  
                        <?php echo link_to($v -> getNumero(), ('gesdoc/verVersion?id_version=' . $v -> getIdVersion())) ?> |  
                        <?php endforeach; ?>  
                    </td>  
                    <td>  
                        <?php echo link_to('modificar', 'gesdoc/modificar?id_documento=' . $d -> getIdDocumento()) ?> |  
                        <?php echo link_to('subir versión', 'gesdoc/subirVersion?id_documento=' . $d -> getIdDocumento()) ?>  
                    </td>  
                </tr>  
            </tbody>  
        </table>  
    </div>  
</div>
```

Observa que en este código se han implementado, mediante el uso del *helper link_to()*, los *links* correspondientes para descargar los documentos asociados a las versiones al picar en el nº de versión. Se ha decidido llamar a la acción en cuestión *verVersion*, y se alojará en el propio módulo *gesdoc*.

Implementación del filtro de documentos

Implementación del filtro de documentos

Ahora, según lo especificado en el análisis que realizamos en la unidad 4, vamos a colocar en la pantalla que acabamos de construir un formulario para filtrar los documentos. Lo ideal para realizar esta labor es utilizar los filtros que hemos generado al principio del proyecto mediante la tarea *propel:build-filters*. Los filtros son una serie de clases que modelan formularios para realizar búsquedas. No obstante, para no sobrecargar de contenidos la presente unidad, hemos decidido prescindir de ellos y construir manualmente el código *HTML* correspondiente al filtro en cuestión. Más adelante, cuando tratemos los formularios y filtros de *symfony*, volveremos a revisar este punto de la aplicación.

Insertaremos cajas de texto para los campos *título*, *descripción* y *autor*, un desplegable para el campo *tipo*, y una caja de selección múltiple para las *categorías*. El formulario de búsqueda que debemos añadir a la plantilla *indexSuccess.php* quedaría como sigue:

Formulario de búsqueda añadido al archivo:
apps/frontend/modules/gesdoc/templates/indexSuccess.php

```
...
<form name="filtro" method="post" action=<?php echo url_for('gesdoc/index') ?>>
    titulo:<input type="text" id="titulo" name="documentos[titulo]" value="" />
    descripcion:<input type="text" id="descripcion" name="documentos[descripcion]" value="" />
    autor:<input type="text" id="autor" name="documentos[autor]" value="" />
    tipo: <select name="documentos[id_tipo]" id="id_tipo">
        <option value="" selected="selected" />
        <?php foreach ($tipos as $t) : ?>
            <option value=<?php echo $t -> getIdTipo()?> selected="selected">
                <?php echo $t -> getNombre() ?>
            </option>
        <?php endforeach; ?>
    </select>
    categorias: <select name="documentos[categoria_list][]" multiple="multiple" id="categoria_list">
        <option value="" selected="selected" />
        <?php foreach ($categorias as $c) : ?>
            <option value=<?php echo $c -> getIdCategoria() ?> selected="selected">
                <?php echo $c -> getNombre() ?></option>
        <?php endforeach; ?>
    </select>
    <input type="submit" value="Buscar" />
</form>
...
```

Fíjate que la lista de tipos y de categorías hay que construirlas dinámicamente mediante consulta a la base de datos, es decir, a través de objetos de *Propel*. Es decir, para que esta plantilla pueda ser interpretada correctamente por *PHP*, debemos proporcionarle las variables *\$tipos* y *\$categorias*, que serán respectivamente objetos de las clases de *Propel Tipos* y *Categorias*. Como ya debes saber, el lugar donde se definen estos objetos es en la acción correspondiente. Así pues la acción *index* (método *executeIndex()*) quedaría como sigue:

Trozo de código del archivo: apps/frontend/modules/gesdoc/actions/actions.class.php

```
public function executeIndex(sfWebRequest $request)
{
    $c = new Criteria();

    $this -> tipos = TiposPeer::doSelect($c);
    $this -> categorias = CategoriasPeer::doSelect($c);
```

Procesamiento de la petición. El objeto SfWebRequest.

```
$this -> documentos = DocumentosPeer::doSelect($c);  
}
```

Podemos utilizar el mismo criterio en los tres métodos *doSelect()* de cada clase *peer* ya que, por lo pronto, está vacío. Ahora puedes probar el resultado de la acción y verás el formulario de búsqueda con el conjunto de *tipos* y *categorías* debidamente relleno. El problema es que no queda muy bonito, le falta formato. Bastaría adaptar el código *HTML* correspondiente al formulario de búsqueda a las *CSS*'s facilitadas en la unidad 4. Fíjate en el documento *HTML* de ejemplo denominado *listadoConFiltro.html* para comprender como usar la hoja de estilo en la visualización del formulario:

Formulario de búsqueda añadido al archivo:
apps/frontend/modules/gesdoc/templates/indexSuccess.php

```
<div id="sf_admin_bar">  
    <div class="sf_admin_filter">  
        <form name="filtro" method="post" action="php echo url_for('gesdoc/index') ?">  
            <table>  
                <tbody>  
                    <tr class="sf_admin_form_row">  
                        <td>Título</td>  
                        <td><input type="text" id="titulo" name="documentos[titulo]" value="" /></td>  
                    </tr>  
                    <tr class="sf_admin_form_row">  
                        <td>Descripción</td>  
                        <td><input type="text" id="descripcion" name="documentos[descripcion]" value="" /></td>  
                    </tr>  
                    <tr class="sf_admin_form_row">  
                        <td>Autor</td>  
                        <td><input type="text" id="autor" name="documentos[autor]" value="" /></td>  
                    </tr>  
                    <tr class="sf_admin_form_row">  
                        <td>Tipo</td>  
                        <td>  
                            <select name="documentos[id_tipo]" id="id_tipo">  
                                <option value="" selected="selected" />  
                                <?php foreach ($tipos as $t) : ?>  
                                <option value="php echo $t -&gt; getIdTipo() ?">  
                                    <?php echo $t -> getNombre() ?>  
                            </option>  
                            <?php endforeach; ?>  
                            </select>  
                        </td>  
                    </tr>  
                    <tr class="sf_admin_form_row">  
                        <td>Categoría</td>  
                        <td><select name="documentos[categoria_list][]" multiple="multiple" id="categoria_list">  
                            <option value="" selected="selected" />  
                            <?php foreach ($categorias as $c) : ?>  
                            <option value="php echo $c -&gt; getIdCategoria() ?">  
                                <?php echo $c -> getNombre() ?>  
                            </option>  
                            <?php endforeach; ?>  
                        </select>  
                    </td>  
                </tbody>  
            </table>  
            <input type="submit" value="Buscar" />  
        </form>  
    </div>  
</div>
```

Este código, que define la capa *sf_admin_bar* se puede colocar en cualquier parte de la plantilla *indexSuccess.php* siempre que quede al mismo nivel que las capas *sf_admin_header* y *sf_admin_content*. Vuelve a cargar la página y verás que tiene un aspecto más agradable. No obstante aún no hemos acabado nuestro trabajo, ya que no sólo basta con pintar el formulario, también hay que procesarlo para realizar el filtrado de documentos.

La acción que dispara el envío del formulario de búsqueda es la propia acción que lo genera, esto es, la acción *index*. Se trata ahora de recoger los parámetros de la petición *HTTP* que se han pasado por *POST* y construir un criterio adecuado para que el método *doSelect()* de la clase *DocumentosPeer* devuelva únicamente los documentos que satisfacen el criterio de búsqueda.

Procesamiento de la petición. El objeto SfWebRequest.

Procesamiento de la petición. El objeto SfWebRequest.

Symfony proporciona un objeto especial para la manipulación de las peticiones HTTP: el objeto *SfWebRequest*, que sirve al programador, fundamentalmente, para recoger los parámetros que el cliente (*browser*) envía al servidor web y para comprobar la existencia de dichos parámetros. Los métodos más usados son:

hasParameter('nombre_parametro');	Devuelve true si la petición contiene un parámetro llamado nombre_parametro y false en caso contrario.
getParameter('nombre_parametro', 'valor_defecto');	Devuelve el valor del parámetro nombre_parametro de la petición si este existe o el valor indicado en valor_defecto si no existe.

Este objeto es uno de los más utilizados en la implementación de las acciones, ya que la interacción con el usuario se lleva a cabo, fundamentalmente, a través de parámetros que se envían desde el cliente mediante peticiones HTTP. Otro mecanismo fundamental para controlar la interacción con el usuario es la *sesión de usuario*. Como es de esperar, symfony también cuenta con un objeto, que estudiaremos en detalle en la próxima unidad, para la manipulación de la sesión.

Con este nuevo concepto en mente vamos a describir el procedimiento que emplearemos para procesar los valores enviados y recoger los documentos pertinentes.

En primer lugar comprobaremos que se han enviado parámetros para la búsqueda, ya que en caso contrario se desea listar todos los documentos y el criterio se dejará vacío. Si, en efecto, la petición viene con el array *documentos* revisaremos los valores de cada uno de los campos e iremos añadiendo criterios de búsqueda al objeto *Criteria* que será pasado como parámetro al método *DocumentosPeer::doSelect()*.

Aplicando este procedimiento la acción *executeIndex()* nos queda como sigue:

Código de la acción index del archivo:
apps/frontend/modules/gesdoc/actions/actions.class.php

```
public function executeIndex(sfWebRequest $request)
{
    $this -> tipos = TiposPeer::doSelect(new Criteria());
    $this -> categorias = CategoriasPeer::doSelect(new Criteria());

    $c = new Criteria();

    //Inicio del filtro
    if($request -> hasParameter('documentos'))
    {
        $documentos = $request -> getParameter('documentos');

        if($documentos['titulo'] != '')
        {
            $c -> add(DocumentosPeer::TITULO, $documentos['titulo'], Criteria::LIKE);
        }

        if($documentos['descripcion'] != '')
        {
            $c -> add(DocumentosPeer::DESCRIPCION, $documentos['descripcion'], Criteria::LIKE);
        }

        if($documentos['autor'] != '')

    }

    $c -> addJoin(DocumentosPeer::ID_USUARIO, UsuariosPeer::ID_USUARIO);
    $c1 = $c -> getNewCriterion(UsuariosPeer::NOMBRE, $documentos['autor'], Criteria::LIKE);
    $c2 = $c -> getNewCriterion(UsuariosPeer::APELLIDOS, $documentos['autor'], Criteria::LIKE);
    $c1 -> addOr($c2);
}
```

Implementación del páginado.

```
        $c -> add($c1);
    }

    if($documentos['id_tipo'] != '')
    {
        $c -> addJoin(DocumentosPeer::ID_TIPO, TiposPeer::ID_TIPO);
        $c -> add(TiposPeer::ID_TIPO, $documentos['id_tipo']);
    }

    $kuku = true;
    foreach ($documentos['categoria_list'] as $cat)
    {
        if($cat != '')
        {
            $c -> addJoin(DocumentosPeer::ID_DOCUMENTO, DocumentoCategoriaPeer::ID_DOCUMENTO);
            $c -> addJoin(DocumentoCategoriaPeer::ID_CATEGORIA, CategoriasPeer::ID_CATEGORIA);
            $c -> addAnd(CategoriasPeer::ID_CATEGORIA, $cat);
        }
    }
}//Fin del filtro

$this -> documentos = DocumentosPeer::doSelect($c);
}
```

Con esto tenemos la funcionalidad especificada en el requisito D07 satisfecha. Para completar el requisito D08 tan sólo nos falta presentar el listado con un paginado. Eso es lo que haremos en el próximo apartado.

Implementación del páginado.

Cuando el número de documentos crezca lo suficiente, será más cómodo presentar el listado subdivido en páginas de una cierta cantidad de registros, con la posibilidad de pasar de página hacia adelante y hacia atrás. *Symfony* también nos ayuda en esto gracias al objeto *sfPropelPager*.

Como su nombre indica, pertenece a la capa de abstracción de base de datos *Propel*. Lo cual es natural ya que en los listados se presentan colecciones de registros que provienen de la base de datos.

Veamos como utilizarlo. En primer lugar sustituimos la última línea de la acción *executeIndex()*

```
$this -> documentos = DocumentosPeer::doSelect($c);
```

Por el siguiente fragmento de código:

```
$pager = new sfPropelPager('Documentos', 4);
$pager->setCriteria($c);
$pager->setPage($request -> getParameter('page', 1));
$pager->init();
$this->pager = $pager;
```

Cuyo funcionamiento es el siguiente:

1. Primero se construye un objeto de la clase *sfPropelPager* (un paginador) indicando en el primer argumento de su constructor el objeto de *Propel* sobre el que deseemos realizar la paginación (en nuestro caso *Documentos*), y en el segundo argumento el número de registros que mostrará por página.
2. Después indicamos el criterio de búsqueda que se utilizará para recuperar los registros.
3. Indicamos qué número de página queremos mostrar. En el código anterior dicho número se recoge de la petición *HTTP* a través del parámetro *page*, si no existe este

Implementación del páginado.

parámetro se sustituye por el valor 1. Esto último nos indica que debe de existir una interacción con el usuario a través de la página que muestra los documentos. Veremos claramente esta interacción en la implementación de la parte de la vista (la plantilla).

4. Se inicializa el paginador, lo que significa que se ejecuta la consulta relativa al criterio utilizado.
5. Por último se hace el paginador accesible a la plantilla, ya que es allí donde lo utilizaremos.

Una vez que el paginador ha sido debidamente construido, podemos utilizar sus métodos para acceder a las entidades relacionadas con él. Los métodos más usados son los siguientes:

Método	Descripción
<code>getNbResults()</code>	Devuelve el nº de registros total que se han recuperado en la consulta.
<code>setPage(n)</code>	Define la página actual como la número <i>n</i> . Este método es el que se ha utilizado en la acción.
<code>getResults()</code>	Devuelve un array con los <i>m</i> objetos de <i>Propel</i> (en nuestro caso <i>Documentos</i>) de la página <i>n</i> , siendo <i>m</i> el número de elementos que se desean mostrar definido cuando se crea el objeto paginado.
<code>getFirstIndice()</code>	Devuelve el número del primer registro mostrado en la página
<code>getLastIndice()</code>	Devuelve el número del último registro mostrado en la página
<code>haveToPaginate()</code>	Devuelve true si hay que paginar, es decir, si el nº de registros devueltos por la consulta es superior al nº de registros por página que deseamos mostrar.
<code>getFirstPage()</code>	Devuelve el número de la primera página que ha construido el paginador. En realidad siempre devolverá el nº 1.
<code>getLastPage()</code>	Devuelve el número de la última página que ha construido el paginador.
<code>getPreviousPage()</code>	Devuelve el número de la página anterior a la que se está mostrando (es decir a la actual, resultado de utilizar el método <code>setPage()</code>)
<code>getNextPage()</code>	Devuelve el número de la página siguiente a la que se está mostrando.

Utilizando adecuadamente estos métodos del objeto *sfPropelPager* en la plantilla vamos a mostrar un listado con un pequeño nº de documentos por página (4 en nuestro ejemplo) y un menú de navegación que permitirá ir directamente a las primera y última página, avanzar o retroceder una página por vez, o ir directamente a un número de página específico.

En la plantilla *indexSuccess.php*, sustituimos la iteración sobre el array de objetos \$documentos por la iteración sobre el array de *m* objetos (4 en nuestro caso) devuelto por \$pager → `getResults()`. De esta manera mostraremos únicamente los 4 registros de la página que se haya indicado a la acción *index* mediante el parámetro *page* de la petición *HTTP*.

```
...
<?php foreach ($pager -> getResults() as $d): ?>
...
```

Implementación de operaciones sobre documentos concretos.

El menú de navegación lo colocaremos en el pie de la tabla que muestra el listado. Añadimos a la tabla de la plantilla *indexSuccess.php* el siguiente trozo de código que implementa el menú de navegación teniendo en cuenta los estilos de las CSS's que estamos utilizando:

Código añadido para implementar la navegación del paginado en el archivo *apps/frontend/modules/gesdoc/templates/indexSuccess.php*

```
...
<tfoot>
  <tr>
    <th colspan="20">
      <div class="sf_admin_pagination">
        <?php if ($pager->haveToPaginate()): ?>
        <?php echo link_to(image_tag('first.png'), 'gesdoc/index?page='.$pager->getFirstPage()) ?>
        <?php echo link_to(image_tag('previous.png'), 'gesdoc/index?page='.$pager->getPreviousPage()) ?>
        <?php $links = $pager->getLinks(); foreach ($links as $page): ?>
        <?php echo ($page == $pager->getPage()) ? $page : link_to($page, 'gesdoc/index?page='.$page) ?>
        <?php if ($page != $pager->getCurrentMaxLink()): ?> - <?php endif ?>
        <?php endforeach ?>
        <?php echo link_to(image_tag('next.png'), 'gesdoc/index?page='.$pager->getNextPage()) ?>
        <?php echo link_to(image_tag('last.png'), 'gesdoc/index?page='.$pager->getLastPage()) ?>
        <?php endif ?>
      </div>
    <?php echo $pager->getNbResults() ?> resultados (del <?php echo $pager->getFirstIndice() ?> al <?php echo $pager->getLastIndice() ?>)
  </th>
</tr>
</tfoot>
...

```

Aunque aún quede por delante mucha aplicación, esta ya comienza a tomar forma permitiéndonos listar documentos filtrados y paginados. Incluso hemos incorporado el acceso mediante *links* a ciertas operaciones (ver *versión*, *modificar* y *subir versión*) cuya implementación realizaremos en los próximos apartados.

Sin embargo, antes de continuar, queremos advertirte que el listado con filtro y paginado que acabamos de implementar presenta un grave fallo. En efecto, mientras no utilicemos el formulario de búsqueda la paginación funciona perfectamente. Pero si realizamos una búsqueda cuyo resultado contenga más registros que los mostrados en cada página, cuando accedemos a otra página a través de algunos de los enlaces del menú de navegación, aparecen de nuevo todos los registros, perdiéndose el filtro que habíamos aplicado. Pruébalo, tienes derecho a enojarte.

La razón de este mal comportamiento es que la aplicación no guarda memoria del filtro que se ha aplicado, y al ser *HTTP* un protocolo sin estado (*stateless*), cuando se realiza la próxima petición a través de cualquier *link* del páginado tras aplicar el filtro, se vuelve a procesar la acción *index* sin enviar datos en el formulario de búsqueda. Por eso se vuelve a construir un criterio vacío y se recuperan de nuevo todos los documentos.

Resolveremos este problema utilizando la *sesión del usuario en el servidor*, que es un mecanismo que se utiliza extensivamente en el desarrollo de aplicaciones *web* para almacenar en el propio servidor datos persistentes que la aplicación necesite conocer acerca del cliente mientras este último la esté utilizando. La sesión es una manera de superar la ausencia de control de estado del protocolo *HTTP*.

Como no queremos introducir más conceptos nuevos en esta unidad, resolveremos este “pequeño” fallo de la aplicación en la próxima unidad, que estudia y utiliza a fondo la sesión de usuario. Por lo pronto continuaremos implementando algunas de las operaciones básicas que se establecieron en el requisito D08 del análisis, concretamente:

- Descargar una versión del documento
- Ver metadatos del documento

Implementación de operaciones sobre documentos concretos.

Descarga de las versiones de un documento.

Implementación de operaciones sobre documentos concretos.

En el listado que acabamos de construir hemos colocado en la columna 'versiones' enlaces a cada una de las versiones de los documentos. Cuando piquemos en estos enlaces la aplicación nos enviará el archivo asociado a la versión, es decir, realizaremos una descarga del mismo.

En el momento de construir dichos enlaces ya propusimos la acción que realizaría la descarga del archivo: *verVersion*, acción que espera recibir a través de la petición *HTTP* la clave principal de la versión a descargar. Fíjate en el código *HTML* del *link* correspondiente a las versiones de los documentos. Tienen este aspecto:

```
<a href="/gestordocumental/web/frontend_dev.php/gesdoc/verVersion/id_version/4">2</a>
```

Así pues el próximo paso es crear una nueva acción en el módulo *gesdoc* de la aplicación *frontend* denominada *verVersion()*, que reciba por *GET* la clave principal de una versión y devuelva al navegador el archivo correspondiente.

Esta acción presenta una particularidad; la respuesta que el servidor *web* debe enviar al cliente no es un documento *HTML*. Actualmente este hecho no debe sorprendernos ya que en el mundo de las aplicaciones *web* cada vez es más frecuente que el servidor *web* envíe a los clientes otro tipo de respuestas, especialmente archivos *XML* que son interpretados por elementos *javascript* en la parte cliente cuando se utilizan interfaces de usuario enriquecidas.

La acción que vamos a implementar ahora funcionará de la siguiente manera: en primer lugar se recoge la clave principal de la versión que se ha solicitado, se recupera el registro de la tabla *versiones* correspondiente a través de un objeto de *Propel* y se usa la información del mismo para localizar en el sistema de archivos el fichero solicitado y enviarlo al cliente en la respuesta. Esta acción, por tanto, no tendrá ninguna plantilla asociada.

Añadimos la acción al módulo implementando el método *executeVerVersion()* según el siguiente código:

Código de la acción *verVersion* del archivo:
apps/frontend/modules/gesdoc/actions/actions.class.php

```
public function executeVerVersion(sfWebRequest $request)
{
    $this ->forward404Unless($request -> hasParameter('id_version'));

    $idVersion = $request -> getParameter('id_version');
    $oVersion = VersionesPeer::retrieveByPK($idVersion);

    $fichero = sfConfig::get('sf_upload_dir').'/usuario_'. $oVersion -> getDocumentos() -> getIdUsuario().'/'. $oVersion -> getNombreFichero();

    $ctype = $oVersion -> getDocumentos() -> getTipos() -> getTipoMime();
    $path = pathinfo($oVersion -> getNombreFichero());
    $nombreFichero = $oVersion -> getDocumentos() -> getTitulo().'_ver-'. $oVersion -> getNumero().'.'. $path['extension'];

    header("Pragma: public"); // required
    header("Expires: 0");
    header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
    header("Cache-Control: private",false); // required for certain browsers
    header("Content-Type: $ctype");

    header("Content-Disposition: attachment;
           filename=\"".basename($nombreFichero)."\");
    header("Content-Transfer-Encoding: binary");
    header("Content-Length: ".filesize($fichero));

    if(file_exists($fichero))
    {
        echo readfile($fichero);
    }
    else
    {
        echo 'No se puede descargar el '. $fichero . '. Comuniquenlo al responsable de la aplicación.' ;
    }

    return sfView::NONE;
}
```

El fichero es enviado directamente utilizando la función *header()* de *PHP*, la cual envía una cabecera *HTTP* pura en la que se indica que la respuesta consiste en un fichero adjunto con

Mostrar los metadatos de un documento.

un determinado tipo *MIME*, que es el se asoció al documento en la base de datos cuando se subió al servidor.

Para enviar el fichero mediante esta función necesitamos las siguientes variables:

Variable	Significado
\$fichero	Es la ruta absoluta del fichero que se va a descargar. Sirve para leer el contenido del fichero en cuestión y enviarlo mediante la instrucción echo <code>readfile(\$fichero)</code>
\$ctype	Es el tipo <i>MIME</i> del fichero que se va a descargar.
\$path	Es un <i>array</i> devuelto por la función <code>pathinfo()</code> que nos permitirá extraer la extensión del fichero en cuestión para utilizarla en el nombre del fichero que el cliente interpretará.
\$nombreFichero	Es el nombre que le asignaremos al contenido que estamos enviando en la respuesta, de manera que cuando llegue al cliente será este el nombre que tendrá el contenido que se ha enviado. Observa que una cosa es el contenido del fichero y otra distinta el nombre que tiene. De hecho en el servidor el archivo tiene un nombre que se originó automáticamente en el proceso de subida al servidor, y cuando lo enviamos al cliente lo cambiamos por <code>\$nombreFichero</code> , que es más descriptivo y que hemos compuesto uniendo el título que le dimos al documento en la base de datos junto con el nº de versión que se descarga.

Por último debemos indicar a *symfony* que la acción no debe ser renderizada por ninguna plantilla. Este es el significado de la última línea:

```
return sfView::NONE;
```

Ahora ya deben funcionar los enlaces de la columna versiones del listado de documentos. Compruébalo y observa como se lleva a cabo la descarga del documento sin que se produzca un cambio de la interfaz de usuario (página web) con el listado.

Mostrar los metadatos de un documento.

En el listado de documentos no hemos incluido ninguna operación (enlace) con la intención de mostrar los metadatos de cada documento. Una posibilidad inmediata sería incluir en la columna de acciones la nueva acción que precisamos. Si embargo, para no sobrecargar demasiado dicha columna propondremos una solución alternativa; se trata de convertir el título de cada uno de los documentos en un enlace que dispare la acción que muestra los metadatos de un documento. Llamaremos a esta acción `verMetadatos()`, y utilizaremos el helper `link_to()` para cambiar los elementos de la columna 'Titulo' por enlaces a dicha acción.

Cambiamos en la plantilla `indexSuccess.php` la línea:

```
<td><?php echo $d -> getTitulo() ?></td>
```

Por esta otra:

```
<td><?php echo link_to($d -> getTitulo(),'gesdoc/verMetadatos?id_documento='.$d -> getIdDocumento()) ?></td>
```

Y ahora escribimos la acción `executeVerMetadatos()`:

Acción `verMetadatos` del archivo: `apps/frontend/modules/gesdoc/actions/actions.class.php`

Mostrar los metadatos de un documento.

```
public function executeVerMetadatos(sfWebRequest $request)
{
    $this -> forward404Unless($request -> hasParameter('id_documento'));
    $id_documento = $request -> getParameter('id_documento');

    $this -> documento = DocumentosPeer::retrieveByPK($id_documento);
}
```

Y creamos el fichero *verMetadatosSuccess.php* donde implementamos la plantilla que mostrará los atributos del objeto `$this -> documento`:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/verMetadatosSuccess.php*

```
<div id="sf_admin_container">
    <h1>Metadatos del documento: <i><?php echo $documento -> getTitulo() ?></i></h1>

    <div id="sf_admin_content">
        <ul class="sf_admin_actions">
            <li><?php echo link_to('volver', 'gesdoc/index') ?></li>
        </ul>
        <div class="sf_admin_form">
            <form>
                <fieldset id="fieldset_1">
                    <h2>Metadatos</h2>
                    <div class="sf_admin_form_row">
                        <div>
                            <label>Titul...</label>
                            <?php echo $documento -> getTitulo() ?>
                        </div>
                    </div>
                    <div class="sf_admin_form_row">
                        <div>
                            <label>Descripción:</label>
                            <?php echo $documento -> getDescripcion() ?>
                        </div>
                    </div>
                    <div class="sf_admin_form_row">
                        <div>
                            <label>Tipo:</label>
                            <?php echo $documento -> getTipos() -> getNombre() ?>
                        </div>
                    </div>
                    <div class="sf_admin_form_row">
                        <div>
                            <label>Público:</label>
                            <?php echo ($documento -> getPublico() == 1)? 'Si' : 'No' ?>
                        </div>
                    </div>
                    <div class="sf_admin_form_row">
                        <div>
                            <label>Autor:</label>
                            <?php echo $documento -> getUsuarios() ?>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
        <div id="sf_admin_footer">
            </div>
    </div>
```

En cuya confección se ha tenido en cuenta la aplicación de los estilos de las CSS's que estamos utilizando.

Ya puedes probar la nueva funcionalidad. Observa que si realizas una búsqueda y, a continuación, miras los metadatos de algún documento, cuando usas el enlace 'volver' para mostrar de nuevo el listado de documentos, se pierde la búsqueda que realizaste. Se trata del mismo problema que ya hemos comentado con los enlaces del paginado. Lo

Conclusión

solucionaremos en la próxima unidad.

Conclusión

Hemos comenzado el desarrollo de nuestro gestor documental implementando el listado de documentos disponibles. Para ello hemos partido de una serie de datos de ejemplo introducidos en la base de datos “a pelo”, ya que nuestra aplicación aún no permite la subida de archivos. También se ha construido un formulario para realizar búsquedas de documentos que será muy útil cuando el número de estos sea muy alto. Además hemos incorporados la posibilidad de descargar las versiones asociadas a los documentos y una pantalla para mostrar los metadatos de los mismos.

Para realizar todo esto hemos necesitado conocer con suficiente profundidad la capa de abstracción de base de datos que proporciona *Propel*. Por eso la primera parte de la unidad se ha dedicado a un estudio teórico, aunque centrado en lo operativo, sobre el *ORM Propel*. Aún queda bastante por hacer pero la aplicación ya tiene “cuerpo” y “presencia” y satisface casi al completo los requisitos D07 y D08 del análisis planteado en la unidad 4. Es cierto que queda por resolver la pérdida del filtro cuando cambiamos de acción a través del paginado o cuando volvemos al listado desde la pantalla de metadatos, pero quédate tranquilo pues volveremos a este punto y lo resolvemos en la próxima unidad que trata el tema de la sesión de usuario en el servidor.

Como siempre te aconsejamos que una vez finalizado el tema lo vuelvas a estudiar con más profundidad “toqueteando” el código desarrollado y reflexionando sobre los resultados obtenidos.

Unidad 6: La sesión de usuario en el servidor. El inicio de sesión.

Terminamos la unidad anterior revelando un problema que se ha presentado cuando pedimos una nueva página del listado después de realizar una búsqueda. O cuando mostramos los metadatos de un documento y volvemos de nuevo al listado a través del enlace 'volver' de la pantalla que muestra dichos metadatos. Y es que en esos casos se vuelve a mostrar un listado con todos los documentos existentes, sin tener en cuenta el filtro que se aplicó con anterioridad.

Además, por el momento, la aplicación no "sabe" qué usuario la está utilizando por lo que no muestra los datos del mismo en la cabecera, tal y como se ha especificado en los requisitos y en los bocetos de las pantallas. Además, debido a este desconocimiento, cualquier persona podría utilizar todas las funcionalidades de la aplicación. Para evitar este grave problema de seguridad y satisfacer los requisitos U01, U02 y U06 debemos diseñar e implementar algún tipo de control de acceso.

Aunque aparentemente inconexos, estos dos problemas se resuelven con un mismo elemento clave de todas las aplicaciones web: **la sesión de usuario en el servidor**. Este será el concepto que abordaremos y exploraremos en la presente unidad, donde resolveremos el problema del filtro perdido e incorporaremos el mecanismo mediante el cual los usuarios se identifican y quedan reconocidos por la aplicación hasta que se desconectan de la misma, es decir durante toda la **sesión**.

La sesión de usuario en el servidor

Las aplicaciones web pertenecen a un tipo arquitectura de software conocida como cliente-servidor; la aplicación reside en un servidor que envía la interfaz de usuario y los datos al cliente (el navegador web) que la solicita. Cliente y servidor se encuentran físicamente en distintos lugares del mundo pero se comunican a través de la red mediante el uso de *HTTP* que es un protocolo sin estado. Esto último significa que no guarda información sobre conexiones anteriores, es decir que cada petición realizada por el cliente debe especificar todos los datos necesarios para que la aplicación que reside en el servidor sepa como debe responder. O dicho de otra manera, cada petición es independiente de las anteriores, no "sabe" nada de ellas.

Según lo que acabamos de explicar, si la aplicación necesita conocer, por ejemplo, quien la está utilizando y en qué estado se encuentra, el cliente debe encargarse de comunicárselo en cada petición que realice. En principio esta limitación puede parecer engorrosa y peligrosa. Engorrosa por que pueden ser muchos los datos que se necesiten para especificar tanto a un usuario como a un determinado estado. Pensemos en nuestra aplicación; para satisfacer los requisitos se necesita el nombre los apellidos y el perfil del usuario. Por otra parte, en el caso del filtro de búsqueda que hemos implementado se puede llegar a requerir hasta 5 parámetros (título, descripción, autor, tipo y categoría). Y peligrosa por que los datos deberían circular por la red cada vez que el cliente realiza una petición *HTTP* al servidor o este último devuelve una respuesta al cliente comprometiendo la seguridad de la aplicación.

Afortunadamente existe un mecanismo que simplifica la cantidad de datos que circulan por la red y permite preservar los datos claves que necesita la aplicación durante el tiempo que un cliente determinado la use. Estrictamente hablando, dicho tiempo se denomina **sesión** del usuario, y el mecanismo se conoce como **control de la sesión de usuario**. Pero, abusando del lenguaje, hablaremos simplemente de sesión. El contexto nos indicará en cada momento a que nos referimos, si al tiempo o al mecanismo de control.

Obviamente, en el lado del servidor las aplicaciones web pueden crear ficheros para leer y escribir en ellos. La técnica consiste en crear un fichero la primera vez que un cliente conecta con la aplicación y asociarlo a tal cliente. Este fichero es utilizado por la aplicación

Unidad 6: La sesión de usuario en el servidor. El inicio de sesión.

para almacenar los datos que deben persistir durante el tiempo que el cliente la utiliza. Por motivos de seguridad el fichero es eliminado por el servidor un tiempo después de que el cliente haya dejado de realizar peticiones. El nombre del fichero es una larga cadena alfanumérica que se genera aleatoriamente y es usado como identificador de la sesión. Cuando el cliente realiza una nueva petición debe enviar de alguna manera este identificador al servidor, de esta manera sabrá qué fichero almacena los datos persistentes relativos a la sesión que el cliente inició. Aunque el identificativo de la sesión se puede enviar desde el cliente al servidor como un dato de la petición *GET*, la manera más segura de hacerlo es a través de una *cookie*, de forma que la primera vez que el cliente pide utilizar la aplicación, esta crea el fichero de sesión y envía al cliente una *cookie* con la cadena que lo identifica. A partir de ese momento el cliente enviará dicha *cookie* al servidor durante las sucesivas peticiones, y la aplicación sabrá donde guarda los datos que necesita para ese cliente en concreto. Esta es la forma en que *symfony* trata por defecto las sesiones y aunque el *framework* puede ser configurado para pasar el identificativo de la sesión por la *URL* (petición *GET*) no es recomendable.

En *PHP* el acceso a las variables almacenadas en la sesión se realiza mediante el *array* asociativo *\$_SESSION* que es una variable predefinida de *PHP*. Dicho *array* se construye y está disponible durante toda la sesión siempre que se indique mediante la instrucción de *PHP* *start_session()*. Si necesitamos almacenar una variable durante la sesión basta que la incluyamos como miembro de *\$_SESSION*. Por ejemplo, si necesitamos almacenar el nombre y los apellidos del usuario podemos hacer los siguientes:

```
<?php  
  
$_SESSION['nombre'] = 'Alberto';  
$_SESSION['apellidos'] = 'González García'
```

Sin embargo, por cuestiones de organización, sería más correcto hacer lo siguiente:

```
<?php  
  
$_SESSION['usuario']['nombre'] = 'Alberto';  
$_SESSION['usuario']['apellidos'] = 'González García'
```

De esta manera podremos seguir añadiendo datos referidos al usuario y recuperarlos de una vez:

```
<?php  
  
$usuario = $_SESSION['usuario'];  
  
$nombre = $usuario['nombre'];  
$apellidos = $usuario['apellidos']
```

Lo que pretendemos decir con esto es que, ya que *PHP* representa la sesión con un *array* asociativo, utilicemos la potencia estructural que estos ofrecen para organizar adecuadamente los datos de la sesión.

Cuando trabajamos en un proyecto *symfony*, el propio *framework* se hace cargo de iniciar la sesión, por lo que no tenemos que preocuparnos de ello. Por otro lado, aunque podemos utilizar el *array* predefinido *\$_SESSION*, *symfony* ofrece una manera más elegante y adecuada de manipular los datos de la sesión. Como no podía ser de otra manera en un entorno de programación orientado a objetos, *symfony* utiliza un objeto para realizar operaciones con la sesión. Tal objeto es una instancia de la clase definida para cada

Unidad 6: La sesión de usuario en el servidor. El inicio de sesión.

aplicación en el archivo `apps/nombre_aplicación/lib/myUser.class.php`. Si abres el fichero `apps/frontend/lib/myUser.class.php` comprobarás que la clase se denomina `myUser` y deriva de la clase `sfBasicSecurityUser`. Como podrás suponer, podemos cambiar dicha clase para adaptarla a nuestras necesidades, aunque para la aplicación que estamos desarrollando en el curso nos basta con la que `symfony` nos ofrece por defecto. A partir de ahora nos referiremos a este objeto como `sfUser`.

Desde las acciones, se puede acceder a dicho objeto utilizando el método `getUser()` de la acción,

Desde una acción:

```
<?php  
  
//Porción de código dentro de una acción  
...  
$usuario = $this -> getUser();  
...
```

y desde las plantillas utilizando la variable `$sf_user`.

```
<?php  
//Porción de código dentro de una plantilla.  
...  
$usuario = $sf_user;  
...
```

Una vez que disponemos del objeto de sesión podemos definir nuevos atributos (variables de sesión) y recuperarlos mediante los métodos `setAttribute()` y `getAttribute()` respectivamente. También es muy útil el método `hasAttribute()` para comprobar la existencia de una variable de sesión. A continuación mostramos la manera de utilizarlos en ejemplos de código dentro de una acción:

```
<?php  
...  
// Definir una variable de sesión  
$this -> getUser() -> setAttribute('nombre', 'Alberto');  
  
// Recuperar la variable de sesión 'nombre',  
$nombre = $this -> getUser() -> getAttribute('nombre');  
  
// Definir un atributo que es un array  
$usuario['nombre'] = 'Alberto';  
$usuario['apellidos'] = 'González García';  
  
$this -> setAttribute('usuario', $usuario);  
  
// Recuperar el atributo 'usuario'  
$usuario = $this -> getUser() -> getAttribute('usuario');  
  
//Comprobar si existe el atributo 'usuario'  
if($this -> getUser() -> hasAttribute('usuario'))  
{  
    //hacer algo
```

De vuelta con el filtro perdido.

```
}
```

En el código anterior `$this` se refiere al objeto `sfActions` donde se esté trabajando en el momento. Observa la similitud que existe entre el acceso a los datos de la sesión mediante el objeto `sfUser` y el acceso a la petición del cliente mediante el objeto `sfWebRequest`, pero a la vez ten en cuenta que representan dos conceptos muy distintos, aunque ambos sirven para que la aplicación manipule datos relativos al cliente.

Como veremos más tarde en esta misma unidad, las aplicaciones *web* construidas con *symfony* manejan la **autenticación** y la **autorización** de sus usuarios mediante otros métodos adicionales que ofrece el objeto `sfUser`. Pero por lo pronto quedémonos con lo dicho hasta el momento y resolvamos el problema del filtro perdido.

De vuelta con el filtro perdido.

Si has entendido todo lo que llevamos dicho en esta unidad ya habrás intuido como resolver el dichoso problema del filtro perdido que dejamos pendiente en la unidad anterior. La solución consiste en diseñar un mecanismo que permita guardar el estado del filtro entre peticiones utilizando la sesión de usuario:

1. Comprobamos si existe el parámetro *documentos* en la petición, lo cual significa que el usuario rellenó alguno o todos los elementos del formulario de búsqueda. Recuerda que hemos organizado los parámetros del formulario como un *array* asociativo cuyas claves son los nombres de los elementos de búsqueda.
2. Si existe dicho parámetro creamos una variable de sesión, que también denominamos *documentos*, y copiamos el valor de aquel en esta. Así hemos almacenado en la sesión el estado del filtro.
3. Recorremos todos los elementos de la variable de sesión que acabamos de crear (recuerda que es un *array* asociativo) y vamos construyendo progresivamente el objeto *Criteria* que posteriormente utilizaremos para recuperar los registros. Además almacenamos los valores en variables accesibles por la plantilla (usando `$this`) para mostrar en el propio formulario los valores que se solicitaron en la petición anterior, de manera que el usuario sepa que los registros recuperados corresponden a los valores que muestra el formulario de búsqueda.

Este procedimiento modifica el código de la acción *index* de la siguiente manera:

Código de la acción del fichero: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
public function executeIndex(sfWebRequest $request)
{
    $this -> tipos = TiposPeer::doSelect(new Criteria());
    $this -> categorias = CategoriasPeer::doSelect(new Criteria());

    $c = new Criteria();

    if($request -> hasParameter('documentos'))
    {
        $this -> getUser() -> setAttribute('documentos', $request -> getParameter('documentos'));
    }
    //Inicio del filtro
    $this -> valoresFiltro = array();
    $this -> valoresFiltro['titulo']      = '';
    $this -> valoresFiltro['descripcion'] = '';
    $this -> valoresFiltro['autor']       = '';
```

De vuelta con el filtro perdido.

```
$this -> valoresFiltro['id_tipo'] = '';
$this -> valoresFiltro['categorias'] = array();

if($this -> getUser() -> hasAttribute('documentos'))
{
    $documentos = $this -> getUser() -> getAttribute('documentos');

    if($documentos['titulo'] != '')
    {
        $c -> add(DocumentosPeer::TITULO, $documentos['titulo'], Criteria::LIKE);

        $this -> valoresFiltro['titulo'] = $documentos['titulo'];
    }

    if($documentos['descripcion'] != '')
    {
        $c -> add(DocumentosPeer::DESCRIPCION, $documentos['descripcion'], Criteria::LIKE);

        $this -> valoresFiltro['descripcion'] = $documentos['descripcion'];
    }

    if($documentos['autor'] != '')
    {
        $c -> addJoin(DocumentosPeer::ID_USUARIO, UsuariosPeer::ID_USUARIO);
        $c1 = $c -> getNewCriterion(UsuariosPeer::NOMBRE, $documentos['autor'], Criteria::LIKE);
        $c2 = $c -> getNewCriterion(UsuariosPeer::APELIDOS, $documentos['autor'], Criteria::LIKE);
        $c1 -> addOr($c2);
        $c -> add($c1);

        $this -> valoresFiltro['autor'] = $documentos['autor'];
    }

    if($documentos['id_tipo'] != '')
    {
        $c -> addJoin(DocumentosPeer::ID_TIPO, TiposPeer::ID_TIPO);
        $c -> add(TiposPeer::ID_TIPO, $documentos['id_tipo']);

        $this -> valoresFiltro['id_tipo'] = $documentos['id_tipo'];
    }

    if(isset($documentos['categoria_list']))
    {
        foreach ($documentos['categoria_list'] as $cat)
        {
            if($cat != '')
            {
                $c -> addJoin(DocumentosPeer::ID_DOCUMENTO, DocumentoCategoriaPeer::ID_DOCUMENTO);
                $c -> addJoin(DocumentoCategoriaPeer::ID_CATEGORIA, CategoriasPeer::ID_CATEGORIA);
                $c -> addAnd(CategoriasPeer::ID_CATEGORIA, $cat);

                $this -> valoresFiltro['categorias'][] = $cat;
            }
        }
    }
}

//Fin del filtro
$pager = new sfPropelPager('Documentos', 4);
$pager->setCriteria($c);
$pager->setPage($request -> getParameter('page', 1));
$pager->init();
$this->pager = $pager;
}
```

También hay que modificar la plantilla correspondiente (*indexSuccess.php*) para que el formulario de búsqueda muestre los valores que se introdujeron en la petición anterior:

Código de la plantilla *apps/frontend/modules/gesdoc/templates/indexSuccess.php*

```
<div id="sf_admin_header">
    <h2>Listado de documentos</h2>
    <div class="notice">Mensaje de advertencia</div>
</div>

<div id="sf_admin_bar">
    <div class="sf_admin_filter">
        <form name="filtro" method="post" action="<?php echo url_for('gesdoc/index') ?>" >
```

De vuelta con el filtro perdido.

```
<table>
  <tbody>
    <tr class="sf_admin_form_row">
      <td>Título</td>
      <td><input type="text" id="titulo" name="documentos[titulo]" value="php echo $valoresFiltro['titulo'] ?&gt;" /&gt;&lt;/td&gt;
    &lt;/tr&gt;
    &lt;tr class="sf_admin_form_row"&gt;
      &lt;td&gt;Descripción&lt;/td&gt;
      &lt;td&gt;&lt;input type="text" id="descripcion" name="documentos[descripcion]" value="<?php echo $valoresFiltro['descripcion'] ?&gt;" /&gt;&lt;/td&gt;
    &lt;/tr&gt;
    &lt;tr class="sf_admin_form_row"&gt;
      &lt;td&gt;Autor&lt;/td&gt;
      &lt;td&gt;&lt;input type="text" id="autor" name="documentos[autor]" value="<?php echo $valoresFiltro['autor'] ?&gt;" /&gt;&lt;/td&gt;
    &lt;/tr&gt;
    &lt;tr class="sf_admin_form_row"&gt;
      &lt;td&gt;Tipo&lt;/td&gt;
      &lt;td&gt;
        &lt;select name="documentos[id_tipo]" id="id_tipo"&gt;
          &lt;option value=""&gt;
            &lt;?php foreach ($tipos as $t) : ?&gt;
            &lt;option value="<?php echo $t -&gt; getIdTipo() ?&gt;" &gt;?php if ($valoresFiltro['id_tipo'] == $t -&gt; getIdTipo()) echo 'selected="selected"' ?&gt;
            &lt;?php echo $t -&gt; getNombre() ?&gt;
          &lt;/option&gt;
        &lt;/select&gt;
      &lt;/td&gt;
    &lt;/tr&gt;
    &lt;tr class="sf_admin_form_row"&gt;
      &lt;td&gt;Categorías&lt;/td&gt;
      &lt;td&gt;&lt;select name="documentos[categoria_list[]]" multiple="multiple" id="categoria_list"&gt;
        &lt;option value=""&gt;
          &lt;?php foreach ($categorias as $c) : ?&gt;
          &lt;option value="<?php echo $c -&gt; getIdValue() as $cv : ?&gt;" &gt;?php $arrayVF = $valoresFiltro -&gt; getRawValue() ?&gt;
          &lt;option value="<?php echo $c -&gt; getIdCategoria() ?&gt;" &gt;?php if (in_array($cv, $arrayVF['categorias'])) echo 'selected="selected"' ?&gt;
          &lt;?php echo $c -&gt; getNombre() ?&gt;
        &lt;/option&gt;
      &lt;/select&gt;
    &lt;/td&gt;
  &lt;/tr&gt;
  &lt;tr&gt;
    &lt;td&gt;&lt;/td&gt;
    &lt;td&gt;&lt;input type="submit" value="Buscar" /&gt;
  &lt;/td&gt;
  &lt;/tr&gt;
&lt;/tbody&gt;
&lt;/table&gt;</pre

```
</div>
</div>
```



```
<div id="sf_admin_content">
 <div id="sf_admin_list">
 <table>
 <thead>
 <tr>
 <th>Título</th>
 <th>Autor</th>
 <th>Versiones</th>
 <th>Acciones</th>
 </tr>
 </thead>
 <tbody>
 <?php foreach ($pager -> getResults() as $d) : ?>
 <tr>
 <td><?php echo link_to($d -> getTitulo(), 'gesdoc/verMetadatos?id_documento='.$d -> getIdDocumento(), array('class' => 'example5')) ?></td>
 <td><?php echo $d -> getUsuarios() ?></td>
 <td>
 <?php foreach ($d -> getNumeros() as $v) : ?>
 <?php echo link_to($v -> getNumero(), 'gesdoc/verVersion?id_version='.$v -> getIdVersion()) ?> |
 <?php endforeach; ?>
 </td>
 <td>
 <?php echo link_to('modificar', 'gesdoc/modificar?id_documento='.$d -> getIdDocumento()) ?> |
 <?php echo link_to('subir versión', 'gesdoc/subirVersion?id_documento='.$d -> getIdDocumento()) ?>
 </td>
 </tr>
 <?php endforeach; ?>
 </tbody>
 <tfoot>
 <tr>
 <th colspan="20">
 <div class="sf_admin_pagination">
 <?php if ($pager -> haveToPaginate()): ?>
 <?php echo link_to(image_tag('first.png'), 'gesdoc/index?page='.$pager->getFirstPage()) ?>
 <?php echo link_to(image_tag('previous.png'), 'gesdoc/index?page='.$pager->getPreviousPage()) ?>
 <?php $links = $pager->getLinks();
 foreach ($links as $page): ?>
 <?php if ($page -> isCurrent()): ?>
 <?php echo link_to($page, 'gesdoc/index?page='.$page) ?>
 <?php endif; ?>
 <?php endforeach; ?>
 <?php echo link_to(image_tag('next.png'), 'gesdoc/index?page='.$pager->getNextPage()) ?>
 <?php echo link_to(image_tag('last.png'), 'gesdoc/index?page='.$pager->getLastPage()) ?>
 </div>
 <?php endif; ?>
 <?php echo $pager->getNbResults() ?> resultados (del <?php echo $pager->getFirstIndice() ?> al <?php echo $pager->getLastIndice() ?>)
 </th>
 </tr>
 </tfoot>
 </table>
 </div>
</div>
```


```

Et voila!, el problema del filtro perdido quedó resuelto utilizando la sesión de usuario para almacenarlo. Observa que si el usuario cambia los valores del filtro en una próxima petición, también se cambiará el valor de la variable de sesión que lo representa. De hecho la variable de sesión es un reflejo del último cambio realizado por el usuario en el filtro de búsqueda. Ahora podemos utilizar los *links* del paginado sin que se pierda el resultado de la búsqueda.

Nota

En el código anterior se ha utilizado el método `getRawValue()` sobre los objetos `$categorias` y `$valoresFiltros`, los cuales son variables que provienen de la acción. Según lo que hemos dicho hasta el momento, esperamos que estos objetos sean `arrays` (así ocurría en la acción). Sin embargo esto no es verdad debido a que tenemos activado el modo `escaping_strategy` en el fichero `settings.yml` de nuestra aplicación. Lo cual ofrece más seguridad pero también da lugar a más complejidad en el tratamiento de los `arrays` que se desean pasar de la acción a la vista, ya que estos son transformados en objetos `sfOutputEscaper` y para obtener el array original hay que

utilizar el método `getRawValue()` sobre ellos.

Aplicaciones seguras. Autentificación y autorización.

Ahora que conocemos el funcionamiento de la sesión de usuario y como es manejada por `symfony` a través del objeto `sfUser`, vamos a estudiar como utilizarla para dotar a nuestra aplicación de un control de acceso que garantice su seguridad resolviendo los requisitos exigidos en el análisis. Pero antes estudiaremos los conceptos de **autentificación** y **autorización** que ofrecen los fundamentos sobre los que descansa el control de acceso de las aplicaciones *web*.

Autentificación y Autorización

La autentificación es el proceso mediante el cual la aplicación comprueba si el usuario que pretende utilizarla es realmente quien dice ser. Es un proceso de identificación. Para ello la aplicación solicita al usuario ciertos parámetros que lo identifiquen y, mediante algún tipo de comprobación, decide si lo considera identificado en el sistema o no.

La autorización es un proceso mediante el cual la aplicación decide qué funcionalidades puede utilizar el usuario que la maneja y qué información le puede presentar. La aplicación toma tal decisión basándose en la identificación del usuario, esto es, decidirá qué recursos puede ofrecerle una vez que ha admitido la autentificación del usuario. Es, por tanto, un segundo nivel de seguridad en el control de acceso.

Algunas aplicaciones seguras ofrecen todos sus recursos a cualquier usuario autenticado, en cuyo caso la autorización se confunde con la autentificación, pero en la mayoría de las aplicaciones no es así. De hecho nuestro gestor documental exige en sus requisitos este doble nivel de seguridad, ya que dependiendo del perfil que tenga asociado el usuario podrá utilizar más o menos recursos de la aplicación.

En la mayor parte de aplicaciones los parámetros que se requieren para autenticarse son dos: el nombre de usuario (`username`) y su clave o contraseña (`password`) asociada, que constituyen un par que sólo debe ser conocido por el usuario en cuestión para evitar suplantaciones de identidad. También pueden diseñarse mecanismos de autentificación que soliciten otros parámetros distintos, incluso se podría implementar algún tipo de control biométrico, como puede ser la lectura de la huella dactilar, que complementase o sustituyese al que acabamos de describir. No obstante la mayor parte de las aplicaciones *web* basan su control de identidad en el par de parámetros nombre de usuario y clave y, por tanto, será este el que utilizaremos en nuestro gestor documental.

Las aplicaciones *web* utilizan una base de datos o un directorio para almacenar los datos que permiten comprobar tanto la identidad del usuario como sus derechos de acceso. Una vez realizada la comprobación guardan dichos datos en la sesión de usuario en el servidor, de manera que la aplicación puede saber en cada solicitud quien la está utilizando y qué funcionalidades y datos puede ofrecer. Cuando el usuario solicite el final de la sesión o pase un determinado tiempo sin actividad, la aplicación destruirá la sesión y, cuando el usuario realice una nueva petición de un recurso, la aplicación volverá a pedir a este sus parámetros de autentificación (`login`, `password`), volviendo a crear una nueva sesión.

Seguridad en la acción

En las aplicaciones construidas con `symfony` podemos controlar este doble nivel de seguridad a nivel de cada acción mediante el uso de la sesión y los **archivos de configuración para la seguridad**.

Aplicaciones seguras. Autentificación y autorización.

Los archivos de configuración de seguridad, como es de esperar, se deben colocar en los directorios *config* de la aplicación y de los módulos, y se denominan *security.yml*. El nivel de seguridad de cada acción queda definido por la combinación de lo que se especifique en los ficheros *security.yml* de la aplicación y del módulo, primando lo que dicte este último en caso de conflicto. Por lo general, en el archivo de seguridad de la aplicación se define la seguridad por defecto de cada aplicación y en los de los módulos se complementa o cambian dichos parámetros para cada acción.

Autentificación

El parámetro de configuración *is_secure*, que puede ser *true* o *false*, indica al *framework* que para ejecutar la acción el usuario debe estar autenticado. Vamos a comprobarlo. Abre el fichero de seguridad *apps/frontend/config/security.yml* y define el parámetro *is_secure* como *true*:

Contenido del archivo de seguridad de la aplicación: *apps/frontend/config/security.yml*

```
default:  
    is_secure: true
```

Con esto estamos indicando que, mientras no se indique lo contrario en los archivos de configuración de los módulos, todas las acciones requieren que el usuario esté autenticado. Si intentas ejecutar ahora la aplicación verás que aparece una pantalla indicando que la página no es pública y que se requiere estar autenticado para poder acceder.

Nota

Recuerda, si usas el controlador de producción el cambio será efectivo cuando borres la caché con la instrucción *symfony cc*.

Con esta configuración, antes de ejecutar la acción, el *framework* comprueba si el usuario está autenticado. La comprobación se lleva a cabo consultando al objeto *myUser*, ya que *symfony* almacena en la sesión los datos relativos a la seguridad de la acción.

Este objeto proporciona los método *isAuthenticated()* y *setAuthenticated()* para manipular la autenticación. De esta manera, desde una acción cualquiera, podemos autenticar al usuario mediante la siguiente instrucción:

```
<?php  
...  
$this -> getUser() -> setAuthenticated(true);  
...  
//O comprobar si está autenticado mediante esta otra:  
if($this -> getUser() -> isAuthenticated()  
{  
    // haz algo  
}
```

Autorización

El segundo nivel de seguridad, la autorización, al igual que la autenticación, es controlada por el *framework* mediante los ficheros de seguridad (de aplicación y de módulos) y el objeto

Aplicaciones seguras. Autentificación y autorización.

sfUser. Para ello se utilizan las **credenciales** con las que se puede representar el modelo de seguridad (grupos, permisos, perfiles, etcétera).

Las credenciales no son más que valores que podemos asignar a la sesión de usuario mediante el método *addCredential()* del objeto *sfBasicSecurityUser*:

```
<?php  
...  
$this -> getUser() -> addCredential('administrador');  
...
```

También se pueden añadir varias credenciales de una vez mediante el método *addCredentials()*:

```
<?php  
...  
$this -> getUser() -> addCredentials('lectura', 'escritura');  
...
```

También se pueden eliminar una credencial con *removeCredential()*:

```
<?php  
...  
$this -> getUser() -> removeCredential('lectura');  
...
```

O todas de una vez con *clearCredentials()*:

```
<?php  
...  
$this -> getUser() -> clearCredentials();  
...
```

Por último con *hasCredential()* podemos comprobar si el usuario posee ciertas credenciales.

```
<?php  
...  
// Comprueba si tiene la credencial lectura  
if($this -> getUser() -> hasCredential('lectura'))  
{  
    //haz algo  
}  
  
// Comprueba si tiene la credencial lectura Y la credencial escritura  
if($this -> getUser() -> hasCredential(array('lectura','escritura'))  
{  
    //haz algo  
}  
  
// Comprueba si tiene la credencial lectura O la credencial escritura  
if($this -> getUser() -> hasCredential(array('lectura','escritura'), false))  
{  
    //haz algo
```

```
}
```

En los ficheros `security.yml` podemos indicar las credenciales que debe tener el usuario para poder ejecutar las acciones. Como ocurría con la autentificación, lo normal es definir las credenciales por defecto en el fichero de seguridad de la aplicación y complementar o modificar dichas credenciales para cada acción en el fichero de seguridad de los módulos. **Antes de ejecutar una acción, symfony comprobará si el usuario dispone de las credenciales que, en los ficheros de configuración, se han especificado para tal acción.**

Para `symfony` las credenciales no son más que valores que contrasta entre la configuración y la sesión de usuario para permitir o no la ejecución de las acciones. Es decir, las credenciales sólo tienen significado dentro del modelo de seguridad que se haya definido en el análisis de la aplicación.

Las entradas referentes a las credenciales en los ficheros `security.yml` admiten combinaciones lógicas entre ellas, con lo que existe una gran flexibilidad para implementar el modelo de seguridad de la aplicación. Así por ejemplo, si un usuario debe poseer las credenciales '`lectura`' **Y** '`escritura`' para ejecutar la acción `index` de un módulo determinado, se especificaría en el fichero `security.yml` de dicho módulo de la siguiente manera:

```
...
index:
    credentials: [lectura, escritura]
...
```

Si la condición fuese '`lectura`' **O** '`escritura`':

```
...
index:
    credentials: [[lectura, escritura]]
...
```

Es decir, se utilizan corchetes simples para la condición **AND** y dobles para la condición **OR**.

Bueno, con todo esto ya tenemos suficiente carga teórica como para emprender la implementación de la seguridad de la aplicación `frontend` de nuestro gestor documental. En primer lugar diseñaremos unas reglas sencillas de credenciales que satisfagan el modelo de seguridad especificado en los requisitos de la unidad 4. Después construiremos un nuevo módulo que se encargará de comprobar la identidad del usuario y de asignarle una sesión de usuario con los parámetros de seguridad que le correspondan en función de su perfil asociado.

El modelo de seguridad

Los requisitos de la aplicación que estamos construyendo especifican lo siguiente respecto de la seguridad:

El modelo de seguridad

| | |
|------|--|
| U.01 | La aplicación contemplará 4 tipos de usuarios: <ul style="list-style-type: none">• invitado, que podrá realizar búsquedas y descargas de documentos públicos.• lector, que podrá realizar búsquedas y descargas de todos los documentos• autor, que además podrá subir documentos• administrador, que además podrá administrar todos los aspectos de la aplicación. |
| U.02 | La aplicación presentará una parte pública (perfil invitado) en la que cualquier persona podrá realizar búsquedas y descargas de documentos públicos. Para todas las demás acciones el usuario debe estar registrado. |
| C.01 | Los usuarios registrados podrán enviar comentarios a los documentos. |
| C.02 | Los usuarios registrados podrán ver los comentarios de los documentos. |
| P.01 | Los usuarios registrados podrán votar sólo una vez cada documento consultado |

Teniendo esto en cuenta definiremos las siguientes credenciales:

- *lectura*, para buscar y descargar todo tipo de documentos (privados y públicos), para enviar y leer comentarios y para votar los documentos,
- *escritura*, para subir archivos al servidor,
- *administracion*, para administrar todos los aspectos de la aplicación y crear usuarios

Y las asignaremos a los perfiles de la siguiente manera:

| Perfil | Credenciales asociadas |
|---------------|---|
| Invitado | No se le asocian credenciales |
| Lector | <i>lectura</i> |
| Autor | <i>lectura, escritura</i> |
| Administrador | <i>lectura, escritura, administración</i> |

Por otro lado todas las acciones serán seguras (requieren autentificación), salvo la acción *index* para permitir al usuario invitado buscar documentos públicos y descargarlos.

Una vez definido el modelo de seguridad lo implementamos en los ficheros de configuración. Comenzamos definiendo a la aplicación *frontend* segura por defecto. Es decir, el fichero *apps/frontend/config/security.yml* quedaría así:

Contenido del archivo de seguridad de la aplicación: apps/frontend/config/security.yml

```
default:  
    is_secure: true
```

A continuación creamos el directorio de configuración del módulo *gesdoc* de la aplicación *frontend* (*apps/frontend/modules/gesdoc/config*), ya que cuando se genera un módulo este directorio no se crea por defecto:

```
# mkdir apps/frontend/modules/gesdoc/config
```

El modelo de seguridad

Y creamos en dicho directorio el correspondiente fichero `security.yml` con el siguiente contenido:

Contenido del fichero: apps/frontend/modules/gesdoc/config/security.yml

```
index:  
    is_secure: false  
  
verMetadatos:  
    is_secure: false  
  
verVersion:  
    is_secure: false  
  
modificar:  
    credentials: [escritura]  
  
subirVersion:  
    credentials: [escritura]
```

Hemos definido la acción `index` como no segura para que los usuarios invitados puedan buscar y ver documentos. Sin embargo a dichos usuarios no se les debe mostrar los documentos privados. Esta distinción ya no la puede hacer automáticamente el mecanismo de seguridad de `symfony`. Por tanto somos nosotros quienes debemos modificar la acción `index` para tener en cuenta este detalle. Basta con incluir en el criterio la condición de que se recuperen únicamente documentos públicos en el caso de que el usuario no esté autenticado. El final de la acción `index` quedaría:

Comprobación de la autenticación en la acción index para mostrar o no documentos privados

```
<?php  
...  
// Si el usuario no está autenticado (es invitado)  
// muestra sólo los documentos públicos.  
if(!$this -> getUser() -> isAuthenticated())  
{  
    $c -> add(DocumentosPeer::PUBLICO, 1);  
}  
$pager = new sfPropelPager('Documentos', 4);  
$pager->setCriteria($c);  
$pager->setPage($request -> getParameter('page', 1));  
$pager->init();  
$this->pager = $pager;
```

En negrita se indica el código añadido.

Si vuelves a ejecutar la acción `index` comprobarás que aparecen menos documentos en el listado y que todos son públicos.

Nota

Recuerda borrar la caché si estás utilizando el controlador de producción.

El modelo de seguridad

Además si intentas modificar un documento o enviar una nueva versión, aparece una pantalla indicando que no puedes acceder a ella puesto que no estás autenticado. Este último hecho nos hace pensar que, ya que el usuario no autenticado (invitado) no puede ejecutar estas acciones, la aplicación no debería mostrarles dichos accesos. Pues nada, se los vamos a quitar. Se trata de comprobar en la plantilla *indexSuccess.php* si el usuario no está autenticado y, en ese caso, no mostrar los enlaces *modificar* y *subir versión*. El código siguiente muestra como incluir dicho control a la plantillas.

```
...
<?php if($sf_user -> isAuthenticated()) : ?>
<th>Acciones</th>
<?php endif; ?>

...
<?php if($sf_user -> isAuthenticated()) : ?>
<td>
<?php echo link_to('modificar', 'gesdoc/modificar?id_documento='.$d -> getIdDocumento()) ?> |
<?php echo link_to('subir versión', 'gesdoc/subirVersion?id_documento='.$d -> getIdDocumento()) ?>
</td>
<?php endif; ?>
```

Prueba ahora. Mucho mejor ¿no?. Si quieres volver a ver la pantalla como usuario autenticado, para hacer pruebas, puedes añadir al comienzo de la acción *index* la siguiente linea:

```
<?php
$this -> getUser -> setAuthenticated(true);
```

Ahora podrás comprobar que si intentas acceder a la acción *modificar* y *subir versión* a través de los correspondientes enlaces, la aplicación impide la ejecución y muestra un mensaje en el que se dice que se requiere autorización. En efecto, aunque el usuario esté autenticado, no le hemos asignado las credenciales exigidas en el fichero de configuración para ejecutar dichas acciones.

Cuando termines de hacer pruebas elimina la linea anterior, si no quieres que la aplicación presente un grave problema de seguridad. De todas formas aunque elimines tal linea si vuelves a ejecutar la acción *index* verás que el usuario sigue autenticado. ¿Por qué ocurre esto?. La razón es que, aunque hayas eliminado la linea, la sesión de usuario sigue manteniendo al usuario como autenticado durante el tiempo que esta dure. Así que si quieres volver a definir al usuario como no autenticado puedes hacer una de estas tres cosas:

- Esperar sin utilizar la aplicación el tiempo que la configuración de tu PHP (*php.ini*) tenga reservado para la duración de la sesión. Esta solución no es práctica.
- Destruir la sesión de usuario cerrando completamente el navegador.
- Mediante programación, volver a definir al usuario como no autenticado mediante la siguiente instrucción: `$this -> getUser() -> setAuthenticated(false)`, y volver a ejecutar la acción.

Una vez que hemos blindado las acciones del módulo *gesdoc* mediante la asignación de credenciales y exigencias de autentificación a las acciones a través de los ficheros de configuración *security.yml*, debemos construir un procedimiento que construya la sesión de usuario asignándole los atributos de seguridad que le correspondan al usuario en función de su perfil, es decir: si está o no autenticado, y en caso positivo qué credenciales le corresponden.

Registro de usuario o inicio de sesión

Los usuarios registrados en el sistema (lectores, autores y administradores), si desean utilizar la aplicación como tales deberán autentificarse en el sistema mediante sus nombres de usuarios y contraseñas. Para ello el sistema debe contar con un procedimiento mediante el que recoja tales datos, los compruebe y en caso de éxito asigne al usuario la condición de autenticado y las credenciales que le corresponda según el perfil. Este procedimiento se conoce como registro de usuario o inicio de sesión, y es el que vamos a construir en este apartado.

Implementaremos el inicio de sesión en un nuevo módulo de la aplicación que denominaremos *inises*. Más adelante “pasaremos” este módulo de la aplicación *frontend* a un *plugin* con el fin de que pueda ser compartido por la aplicación *backend* del proyecto (aún por construir).

La acción *signIn* del módulo será la encargada de controlar toda la lógica del inicio de sesión cuyo flujo exponemos a continuación:

1. Si en la petición *HTTP* no se ha enviado nada, se mostrará un formulario para que el usuario introduzca su nombre de usuario y contraseña. Tal formulario enviará los datos a la propia acción *signIn*.
2. Si en la petición *HTTP* vienen los datos '*username*' y '*password*', entonces se comprobará si existe un usuario asociado a dicho par de parámetros. Si no existe, la acción vuelve a mostrar el mismo formulario de identificación junto con un mensaje que indica que los datos introducidos no corresponden a ningún usuario.
3. Si existe un usuario asociado al par '*username*', '*password*', entonces:
 - Se define en la sesión de usuario el atributo '*id_usuario*' cuyo valor será la clave primaria del usuario en la tabla '*usuarios*'. De esta manera, en sucesivas peticiones, la aplicación podrá acceder a los datos correspondientes al usuario en cuestión (que están almacenados en la base de datos).
 - Se declara en la sesión al usuario como autenticado. Así, en sucesivas peticiones, la aplicación sabrá que el usuario está debidamente registrado en su base de datos.
 - Se añaden a la sesión las credenciales que le correspondan al usuario en función del perfil que tenga asociado. Por tanto, en sucesivas peticiones, la aplicación sabrá qué funcionalidades puede usar el usuario.
 - Se hace una redirección a la acción '*gesdoc/index*', que presentará más o menos recursos según el grado de autorización del usuario, es decir, según el perfil que tenga asociado.

Por otro lado, la acción *signOut* será la encargada de realizar la desconexión del usuario. Consistirá en destruir la sesión y realizar una redirección al listado de documentos.

Comenzamos por crear el módulo *inises*:

```
# symfony generate:module frontend inises
```

Para implementar el formulario de identificación del usuario, utilizaremos el *framework* de formularios de *symfony*, ya que nos facilita la vida a la hora de realizar la validación de los datos. Aunque aún no hemos tratado los formularios de *symfony* a fondo, nos basta con saber lo que se explicó acerca de ellos en la unidad 5. Comenzamos por crear el directorio *lib* del módulo *inises* donde ubicaremos el fichero con la descripción del formulario:

Registro de usuario o inicio de sesión

```
# mkdir apps/frontend/modules/inises/lib
```

Y en su interior creamos el fichero *LoginForm.php* con la definición del formulario de identificación:

Contenido del fichero *apps/frontend/modules/inises/lib/LoginForm.php*

```
<?php

class LoginForm extends sfForm
{
    public function configure()
    {
        $this -> setWidgets(array(
            'username'      => new sfWidgetFormInput(),
            'password'      => new sfWidgetFormInputPassword(),
        ));

        $this->widgetSchema->setNameFormat('datos[%s]');

        $this -> setValidators(array(
            'username'      => new sfValidatorString(array('required' => true), array('required' => 'Campo requerido')),
            'password'      => new sfValidatorString(array('required' => true), array('required' => 'Campo requerido'))
        ));
    }
}
```

En pocas palabras, el formulario anterior define dos cajas de texto, una para introducir el nombre de usuario y la otra para la clave. Ambas cajas de texto serán requeridas en el proceso de validación del formulario. Por último los parámetros se pasarán en la petición *HTTP* de la siguiente forma: *datos[username]* y *datos[password]*, de manera que serán interpretados por *PHP* como un *array*, mejorando la organización de los datos.

Para definir los parámetros de seguridad debemos crear el directorio *config* del módulo:

```
# mkdir apps/frontend/modules/inises/config
```

y añadirle el fichero *security.yml* con el siguiente código:

Contenido del fichero: *apps/frontend/modules/inises/config/security.yml*

```
signIn:
    is_secure: false
```

Ahora creamos la acción *signIn*, es decir añadimos el método público *executeSignIn()* a la clase *inisesActions* definida en el fichero *actions.class.php* del módulo:

Contenido del fichero: *apps/frontend/modules/inises/actions/actions.class.php*

```
<?php

class inisesActions extends sfActions
{
    public function executeSignIn(sfWebRequest $request)
    {
        $this -> form = new LoginForm();
        if ($request->isMethod('post'))
        {
```

Registro de usuario o inicio de sesión

```
$this->form->bind($request->getParameter('datos'));
if ($this->form->isValid())
{
    $datos = $request -> getParameter('datos');
    $usuario = $this -> compruebaUsuario($datos);
    if($usuario instanceof Usuarios) // Existe el usuario con los datos dados
    {
        $this -> getUser() -> setAuthenticated(true);
        $this -> getUser() -> setAttribute('id_usuario', $usuario -> getIdUsuario());
        $this -> asociaCredenciales($usuario);
        $this -> redirect('gesdoc/index');
    }
    else
    {
        $this -> mensaje = 'Usuario no autorizado';
    }
}
}
```

Esta acción implementa el flujo explicado al principio del apartado. La acción comienza por declarar como atributo de la clase (uso de `$this`) un objeto de la clase `LoginForm`, es decir un formulario. Si la acción no ha sido llamada a través de una petición `POST` (si no se han enviado datos desde el cliente a través del formulario de identificación), la acción termina y da paso a su plantilla correspondiente, `signInSuccess.php`:

Contenido del fichero: `apps/frontend/modules/inises/templates/signInSuccess.php`

```
<form name="loginForm" action="php echo url_for('inises/signIn') ?" method="post">
<?php echo $form -> renderGlobalErrors() ?>
<?php echo $form -> renderHiddenFields() ?>
<?php if (isset($mensaje)) : ?>
<?php echo $mensaje ?>
<?php endif; ?>
<table>
    <tr>
        <th><label for="username">Usuario:</label></th>
        <td><?php echo $form['username'] -> renderError() ?><?php echo $form['username'] -> render() ?></td>
    </tr>
    <tr>
        <th><label for="password">Clave:</label></th>
        <td><?php echo $form['password'] -> renderError() ?><?php echo $form['password'] -> render() ?></td>
    </tr>
</table>
<input type="submit" value="ingresar" /> | <?php echo link_to('volver al listado','gesdoc/index') ?>
```

Es decir, se envía al cliente el formulario de identificación. Observa el uso del objeto `$form` (pasado a la plantilla por la acción `signIn`): para cada caja de texto se pinta la propia caja (`$form['elemento'] ->render()`) y, si lo hubiera, el error que arroja la validación del formulario (`$form['elemento'] ->renderError()`). Obviamente, la primera vez que se envía el formulario, como aún no ha sido validado, la función `renderError()` no devuelve nada. Una vez que se valide el formulario tampoco devolverá nada si la validación es correcta. Pero si dejamos alguno de los campos sin llenar, la validación arrojará un error y al volver a pintar el formulario la función `renderError()` devolverá la cadena 'Campo requerido' que se definió en la declaración de los validadores del formulario.

La validación del formulario se solicita en la acción `signIn` cuando se ha comprobado que la petición es del tipo `POST`. Entonces se "enlazan" (método `bind()`) los datos de la petición con el objeto formulario y se realiza la validación (`$this -> form -> isValid()`). La comprobación de la identidad del usuario se realiza una vez que la validación del formulario es correcta. Entonces se accede a la base de datos para comprobar si existe un usuario con el nombre de usuario y contraseña enviado. Para ello nos apoyamos en la función `compruebaUsuario()` que debemos añadir a la clase `inisesActions`:

Código añadido al archivo: `apps/frontend/modules/inises/actions/actions.class.php`

Registro de usuario o inicio de sesión

```
<?php

protected function compruebaUsuario($datos)
{
    $c = new Criteria();

    $c -> add(UsuariosPeer::USERNAME, $datos['username']);
    $c -> add(UsuariosPeer::PASSWORD, md5($datos['password']));

    $usuario = UsuariosPeer::doSelectOne($c);

    return $usuario;
}
```

y que devuelve el objeto `$usuario` si este existe. En caso contrario devuelve `null`. Si el usuario existe construimos la sesión añadiendo el atributo `id_usuario` con la clave principal del usuario en la tabla `usuarios` y definiéndolo como autentificado. Además le asociamos las credenciales correspondiente en función del perfil. Esto último lo realiza la función `asociaCredenciales()`, que también debemos añadir al código de la clase `inisesActions`:

Código añadido al archivo: `apps/frontend/modules/inises/actions/actions.class.php`

```
<?php

protected function asociaCredenciales($usuario)
{
    $perfil = $usuario -> getPerfil();

    switch ($perfil)
    {
        case 'lector':
            $this -> getUser() -> setAttribute('perfil', 'lector');
            $this -> getUser() -> addCredential('lectura');
            break;
        case 'autor':
            $this -> getUser() -> setAttribute('perfil', 'autor');
            $this -> getUser() -> addCredentials(array('lectura', 'escritura'));
            break;
        case 'administrador':
            $this -> getUser() -> setAttribute('perfil', 'administrador');
            $this -> getUser() -> addCredentials(array('lectura', 'escritura', 'administracion'));
            break;
    }
}
```

Una vez generada la sesión se redirige a la acción `'gesdoc/index'`. Por último, si no existe un usuario con el nombre de usuario y clave enviado, se vuelve a pintar el formulario de identificación con el mensaje `'Usuario no autorizado'`.

Ya sólo nos queda añadir la acción `signOut()` para la desconexión del usuario:

Código de la acción `signOut` del archivo: `apps/frontend/modules/inises/actions/actions.class.php`

```
<?php
...
public function executeSignOut(sfRequest $request)
{
    session_destroy();
    $this -> redirect('gesdoc/index');
}
```

La cual destruye la sesión y redirige de nuevo al listado de documentos.

Registro de usuario o inicio de sesión

Finalmente el código de la clase *inisesActions* queda de la siguiente forma:

Código del archivo: *apps/frontend/modules/inises/actions/actions.class.php*

```
<?php

/**
 * inises actions.
 *
 * @package    gestordocumental
 * @subpackage inises
 * @author     Your name here
 * @version    SVN: $Id: actions.class.php 12479 2008-10-31 10:54:40Z fabien $
 */
class inisesActions extends sfActions
{
    /**
     * Executes index action
     *
     * @param sfWebRequest $request A request object
     */
    public function executeSignIn(sfWebRequest $request)
    {
        $this -> form = new LoginForm();
        if ($request->isMethod('post'))
        {
            $this->form->bind($request->getParameter('datos'));
            if ($this->form->isValid())
            {
                $datos = $request -> getParameter('datos');
                $usuario = $this -> compruebaUsuario($datos);
                if($usuario instanceof Usuarios) // Existe el usuario con los datos dados
                {
                    $this -> getUser() -> setAuthenticated(true);
                    $this -> getUser() -> setAttribute('id_usuario', $usuario -> getIdUsuario());
                    $this -> asociaCredenciales($usuario);
                    $this -> redirect('gesdoc/index');
                }
                else
                {
                    $this -> mensaje = 'Usuario no autorizado';
                }
            }
        }
    }

    public function executeSignOut(sfWebRequest $request)
    {
        session_destroy();
        $this -> redirect('gesdoc/index');
    }

    protected function compruebaUsuario($datos)
    {
        $c = new Criteria();

        $c -> add(UsuariosPeer::USERNAME, $datos['username']);
        $c -> add(UsuariosPeer::PASSWORD, md5($datos['password']));

        $usuario = UsuariosPeer::doSelectOne($c);

        return $usuario;
    }

    protected function asociaCredenciales($usuario)
    {
        $perfil = $usuario -> getPerfil();

        switch ($perfil)
        {
            case 'lector':
                $this -> getUser() -> addCredential('lectura');
                break;
            case 'autor':
                $this -> getUser() -> addCredentials(array('lectura', 'escritura'));
                break;
            case 'administrador':
                $this -> getUser() -> addCredentials(array('lectura', 'escritura', 'administracion'));
                break;
        }
    }
}
```

Conclusiones.

Ahora puedes comprobar el funcionamiento del módulo de inicio de sesión realizando la siguiente petición desde el navegador:

`http://localhost/gestordocumental/web/frontend_dev.php/inises/signIn`

Pruébalo dejando algún campo en blanco, insertando parámetros de identificación falsos y verdaderos. Prueba también la desconexión mediante la siguiente petición:

`http://localhost/gestordocumental/web/frontend_dev.php/inises/signOut`

Queda pues construido el proceso de inicio de sesión y desconexión del usuario. Ahora nos queda integrarlo en la interfaz de usuario de la aplicación, ya que no es muy elegante ni cómodo tener que realizar peticiones directamente desde la barra de direcciones del navegador. Haremos esto último en la siguiente unidad donde introduciremos algunos conceptos importantes relacionados con la vista que nos permitirán incorporar a la aplicación un menú de navegación, un mensaje de bienvenida con los datos del usuario que está registrado en la aplicación y un enlace para conectarnos como usuario registrado cuando trabajamos en modo invitado o un enlace para desconectarnos si trabajamos en modo registrado.

Conclusiones.

En esta unidad hemos trabajado el concepto de sesión de usuario en el servidor gracias al cual la aplicación puede “conocer” quien la está utilizando y puede almacenar variables de estado entre sucesivas peticiones. Hemos visto que es un mecanismo gracias al cual se puede superar la restricción de ausencia de estado impuesta por el protocolo *HTTP*, base de la comunicación entre cliente y servidor en las aplicaciones *web*. Debe quedar claro que este mecanismo no dota al protocolo *HTTP* de un control de estado; entre una petición *HTTP* y otra no existe ninguna relación. Para que la aplicación pueda realizar un seguimiento de quien la está utilizando, cada petición debe enviar un parámetro que identifique al cliente. Este parámetro es la *cookie* de sesión y es utilizado por la aplicación *web* para identificar las sesiones que tiene abierta en el servidor donde se aloja.

Mediante el mecanismo de sesión hemos resuelto dos problemas: la pérdida del filtro de búsqueda entre peticiones y la autenticación y autorización del usuario en la aplicación, conceptos que también han sido estudiado en la unidad junto con la estrategia propia de *symfony* para su tratamiento.

Unidad 7: Profundizando en la arquitectura MVC de Symfony

Aún quedan bastantes requisitos por cumplir pero la aplicación va tomando cuerpo. Para llegar hasta aquí hemos estudiado gran parte de los conceptos básicos de *symfony* y algunos, como la sesión de usuario, que son propios de cualquier aplicación web, aunque, eso sí, desde la óptica de *symfony*.

El objetivo que perseguimos en esta unidad es mirar más de cerca algunos de los aspectos que ya hemos usado a lo largo del curso e introducir algunos conceptos de “grano fino” muy poderosos para hacer un uso eficaz del *framework*. Comenzaremos por repasar los principios de la arquitectura *MVC*. Posteriormente veremos, sin entrar en los detalles del núcleo de *symfony*, el funcionamiento básico de la parte controladora, y por último estudiaremos la manera en que *symfony* compone las vistas a través de los conceptos de *layout*, *template*, *slot*, *partials* y componentes.

Los conocimientos adquiridos nos permitirán incorporar a nuestra aplicación las siguientes características:

- Un menú de navegación.
- Un botón para entrar como usuario registrado en el caso de que estemos utilizando la aplicación como invitado, o para desconectarnos de la aplicación si ya hemos iniciado una sesión como usuario registrado en ella.
- Un espacio para indicar los datos del usuario que está utilizando la aplicación: nombre, apellidos y perfil.
- Una presentación distinta según el perfil que tenga asociado el usuario que la utiliza.
- Un sencillo servicio *web* que suministra información actualizada a suscriptores *RSS* acerca de los nuevos documentos públicos.

Y sin más preámbulos ivamos al lío!

El patrón MVC en symfony

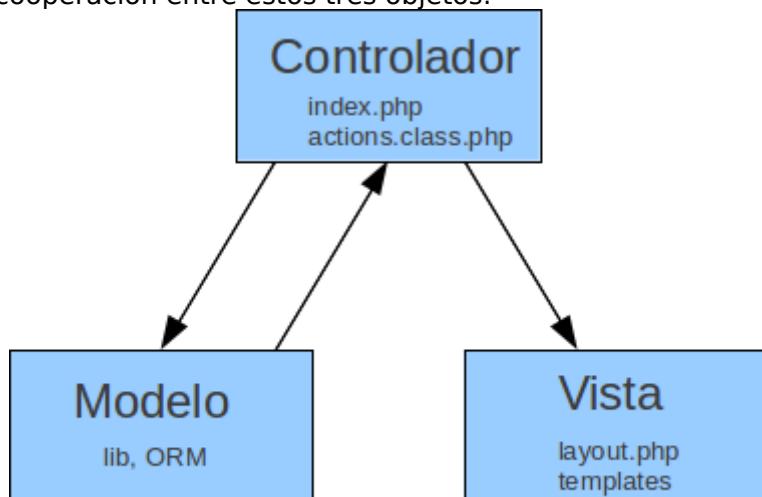
A lo largo de muchos años de programación de aplicaciones en general y de programación orientada a objetos en particular, los diseñadores y programadores se han percatado de la existencia de elementos comunes en las soluciones de los problemas que resolvían. Este hecho, especialmente en el mundo de la *POO*, ha posibilitado el estudio y definición de los denominados patrones de diseño. En pocas palabras, y para satisfacer los propósitos de este curso, los patrones de diseño son propuestas generales planteadas en término de una colaboración de objetos mediante las que se resuelven una gran cantidad de problemas que aparecen una y otra vez en el desarrollo de aplicaciones informáticas. Es decir, los patrones nos muestran ciertas **clases** de problemas y unas estrategias bien contrastadas por la experiencia para atacar su solución.

Christopher Alexander, un reconocido arquitecto inglés dice que “*cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacerlo mismo dos veces*”. Y aunque se refiere al mundo de la arquitectura, sus palabras describen a la perfección la esencia de los patrones de diseño de software.

Los patrones de diseño se encuentran bien catalogados. El famoso libro “*patrones de diseño*” de *E. Gamma, R. Helm, R. Johnson y J. Vlissides*, más conocidos como la “banda de los cuatro” (*the gang of four*), realiza una exhaustiva catalogación y descripción de los patrones de diseño. Si tuviésemos que enfrentarnos a la difícil tarea de construir un complejo sistema *software*, por ejemplo: un *framework* para el desarrollo de aplicaciones *web*, sería de mucha ayuda conocer con detalle este catálogo de patrones y utilizar aquellos

que encasen en la naturaleza de nuestro problema. Esto mismo, además de estudiar otros frameworks existentes en otros lenguajes como *ruby on rails*, es lo que han hecho los chicos de *symfony* para su desarrollo. Y, seguramente, a ello se deba gran parte del éxito de *symfony*: *No reinventar la rueda y utilizar con rigor todo aquello cuyo funcionamiento este bien demostrado*. *Observer*, *decorator*, *factory*, y *MVC* son los nombres de algunos patrones de diseño que utiliza *symfony* en su implementación. De todos ellos, el usuario de *symfony* debería conocer con cierto detalle el patrón *MVC*, ya que dota a la aplicación de su estructura “más visible”, y ayuda al programador de aplicaciones web con *symfony* a colocar cada cosa en su sitio y a construir aplicaciones altamente modulares, extensibles, portables, mantenibles y, sobre todo, “vivas” durante mucho tiempo.

El patrón *MVC* consiste en tres tipos de objetos: el modelo que es el objeto de la aplicación, la vista que es su representación y el controlador que define el modo en que la interfaz reacciona a la entrada del usuario. De una manera poco ortodoxa podemos decir que el controlador controla el flujo de la aplicación, pide al modelo aquello que el usuario solicita, y le devuelve (al usuario) una representación del modelo a través de la vista. El siguiente gráfico ilustra la cooperación entre estos tres objetos:



Esta separación en capas con responsabilidades bien definidas permite, además de mejorar la organización del código, la posibilidad de representar de diferentes formas la misma información. Por ejemplo, en nuestro gestor documental, podremos ofrecer al usuario un listado de documentos en formato *HTML* para que pueda ser visualizado por un navegador *web*, que es lo que se suele hacer en las aplicaciones *web*. Pero también podemos ofrecer el mismo listado en formato *XML* con la semántica de *RSS* para que sea procesado por un programa lector de noticias *RSS*; o en formato *XML* con una semántica definida por nosotros para ofrecer un servicio *web* a medida; o en cualquier otro formato que se nos ocurra (*JSON*, *YAML*, *CSV*, etcétera) para hacer lo que se nos ocurra. *Symfony*, al implementar el *MVC* ofrece todas estas posibilidades.

En *Symfony* cada parte del patrón *MVC* constituye un sistema de varios componentes:

Parte del patrón	Componentes
Controlador	El Controlador frontal, los filtros, las acciones y los objetos <i>request</i> , <i>response</i> y <i>session</i>
Vista	El Layout de la aplicación, las plantillas de los módulos, los <i>slots</i> , los <i>partials</i> , los componentes y los <i>helpers</i>

La parte del controlador.

Modelo	El <i>ORM</i> , los formularios, las extensiones propias que el programador realice de las clases del <i>ORM</i> y las clases y funciones propias que el programador construya para implementar la lógica de negocio.
--------	---

En esta unidad volveremos a la carga con las dos primeras.

La parte del controlador.

El controlador frontal

El punto único de entrada de una aplicación *symfony* es un archivo *PHP* ubicado en el directorio *web*, que se conoce como el controlador frontal y que tiene el siguiente aspecto:

```
<?php  
  
require_once(dirname(__FILE__).'../../config/ProjectConfiguration.class.php');  
  
$configuration = ProjectConfiguration::getApplicationConfiguration('frontend', 'prod', false);  
sfContext::createInstance($configuration)->dispatch();
```

Aunque, como ya hemos visto a lo largo del curso, al generar una aplicación con la tarea *generate:app* se crean dos controladores frontales, uno para desarrollo y el otro para la producción, podemos crear tantos controladores como deseemos. La diferencia entre uno y otro, como se aclarará en los párrafos siguientes, será la definición del entorno de ejecución.

Veamos todo lo que hacen estas tres líneas de código.

La primera línea carga la clase de configuración del proyecto y las librerías de *symfony* (el núcleo). En el archivo *config/ProjectConfiguration.class.php* se declara la ruta al núcleo de *symfony*. Si estamos utilizando una instalación centralizada de *symfony* en el servidor (por ejemplo si hemos instalado *symfony* con *PEAR*), esta ruta apuntará a dicha instalación, pero también podemos colocar todo el núcleo de *symfony* dentro del árbol de directorio de nuestro proyecto y cambiar esta ruta para que apunte allí donde hayamos colocado el núcleo. Un buen sitio puede ser un directorio llamado *symfony* que cuelgue del directorio *lib* del proyecto (*lib/symfony*). De esta manera nuestro proyecto será autosuficiente y podremos portarlo a otros servidores con *PHP* que no tengan instalado *symfony*. Al fin y al cabo podemos decir que, a nivel de ficheros, *symfony* no es más que un conjunto de librerías *PHP*.

La segunda línea crea la configuración de la aplicación. El primer argumento indica el nombre de la aplicación que deseamos lanzar, el segundo el entorno que se desea ejecutar, y el tercero la habilitación del modo de depuración.

Los entornos típicos son *prod* y *dev*. El primero se refiere al entorno de producción, y el segundo al entorno de desarrollo. Podemos crear tantos entornos y controladores frontales como deseemos, aunque por defecto sólo se proporcionan estos dos; suficientes para desarrollar todo tipo de aplicaciones con *symfony*. Te habrás fijado que en muchos de los archivos *YAML* de configuración algunos parámetros aparecen bajo la sección *dev*, otros bajos la sección *prod* y otros en la sección *all*. En función del entorno indicado en el controlador frontal se utilizan unos u otros parámetros en la ejecución del *framework*. Los que pertenecen a la sección *all* son comunes a todos los entornos a menos que el mismo parámetro se defina en otro entorno, en cuyo caso tiene validez el de este último. Este hecho permite, por ejemplo, que en el entorno de desarrollo se utilice una versión de la capa de abstracción de acceso a base de datos *PDO* diseñada para la depuración, mientras que en el de producción se utiliza la versión más optimizada de la misma capa. Puedes verlo consultando el archivo *config/databases.yml*. También podemos utilizar esta funcionalidad para definir distintas bases de datos en cada entorno. En fin, a medida que avanzamos vamos comprobando la tremenda flexibilidad de configuración que ofrece el *framework*.

Los filtros y las acciones

La tercera línea lanza toda la maquinaria de *symfony* con la configuración especificada anteriormente. La secuencia de operaciones que se disparan descrita a alto nivel es la siguiente:

- Se cargan e inicializan las clases del núcleo.
- Se carga la configuración correspondiente al entorno de ejecución indicado en el controlador frontal.
- Se decodifica a *URL* de la petición *HTTP* para determinar la acción a ejecutar y sus parámetros.
- Si la acción no existe se redirecciona a la acción asignada al *error 404*. Esta acción se define en el archivo *apps/nombre_aplicación/config/setting.yml* a través de los parámetros *error_404_module* y *error_404_action*. En caso de que no definamos explicitamente estos parámetros *symfony* utiliza una propia del *framework* por defecto.
- Se activan los filtros. Si los ficheros de configuración de seguridad (*security.yml*) lo exigen, se realiza la comprobación de la autentificación y las credenciales que hemos estudiado en el tema anterior. De manera que si el usuario, en su sesión, no dispone de la autentificación y las credenciales exigidas se realiza una redirección a la acción de “autentificación requerida” o “autorización requerida”, las cuales se definen en el fichero *apps/nombre_aplicación/config/setting.yml* mediante los parámetros *login_module* y *login_action*, para el caso de violación de autentificación o *secure_module* y *secure_action*, para el caso de violación de credenciales (autorización). Si no definimos estos parámetros, *symfony* realiza la redirección a unas acciones internas que ofrece por defecto. Sin conocer la existencia de estas acciones, ya las hemos visto funcionando en la unidad anterior cuando estudiábamos la seguridad en la acción.
- Se ejecutan los filtros (primera pasada). Más adelante hablamos de los filtros.
- Se ejecuta la acción y se produce la vista. Es decir se ejecuta el código construido por el programador, el cual constituye las peculiaridades de la aplicación, es decir, las piezas que le faltan al puzzle para completarlo.
- Se ejecutan los filtros (segunda pasada).
- Se envía la respuesta al cliente.

Este flujo constituye una parte importante del núcleo de *symfony* y conviene conocerlo para hacerse un plano de situación que nos dé una visión general del conjunto. No entraremos en las profundidades del núcleo ya que no es necesario para hacer un uso provechoso del *framework* y construir aplicaciones *web* de calidad. No obstante, al estudiante curioso y con ganas de ir más allá de la construcción de aplicaciones *web* le resultará un seductor y desafiante ejercicio estudiar los aspectos internos del núcleo.

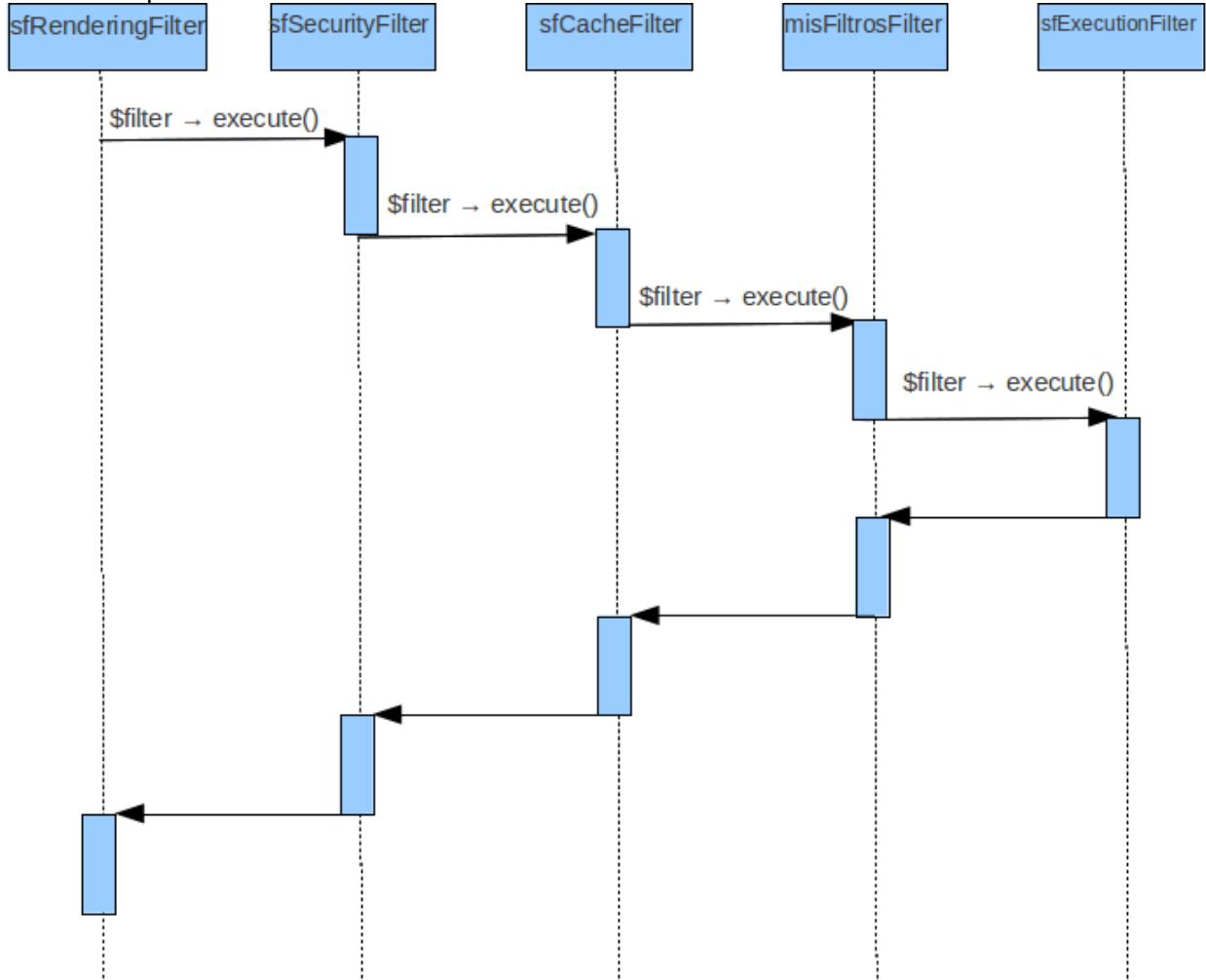
Los filtros y las acciones

Aunque ya hemos implementado unas cuantas acciones a lo largo del curso y podemos pensar que tenemos un conocimiento empírico suficiente, en esta sección mostraremos algunos detalles aún desconocidos.

En primer lugar, si volvemos al flujo de operaciones del controlador, comprobamos que el turno de ejecución de la acción está entre dos turnos de ejecución de filtros, o lo que es lo mismo, entre un pre-filtro y un post-filtro. ¿Y qué es esto de los filtros? Pues otro patrón de diseño denominado *chain of responsibility* o cadena de responsabilidad en nuestro idioma. En términos genéricos, el propósito de este patrón es dar a más de un objeto la posibilidad de responder a una petición, encadenando los objetos receptores que van pasando la petición a través de la cadena hasta que es procesada por algún objeto final. Cada uno de los objetos en la cadena realiza su propio proceso siendo la salida de uno la entrada del

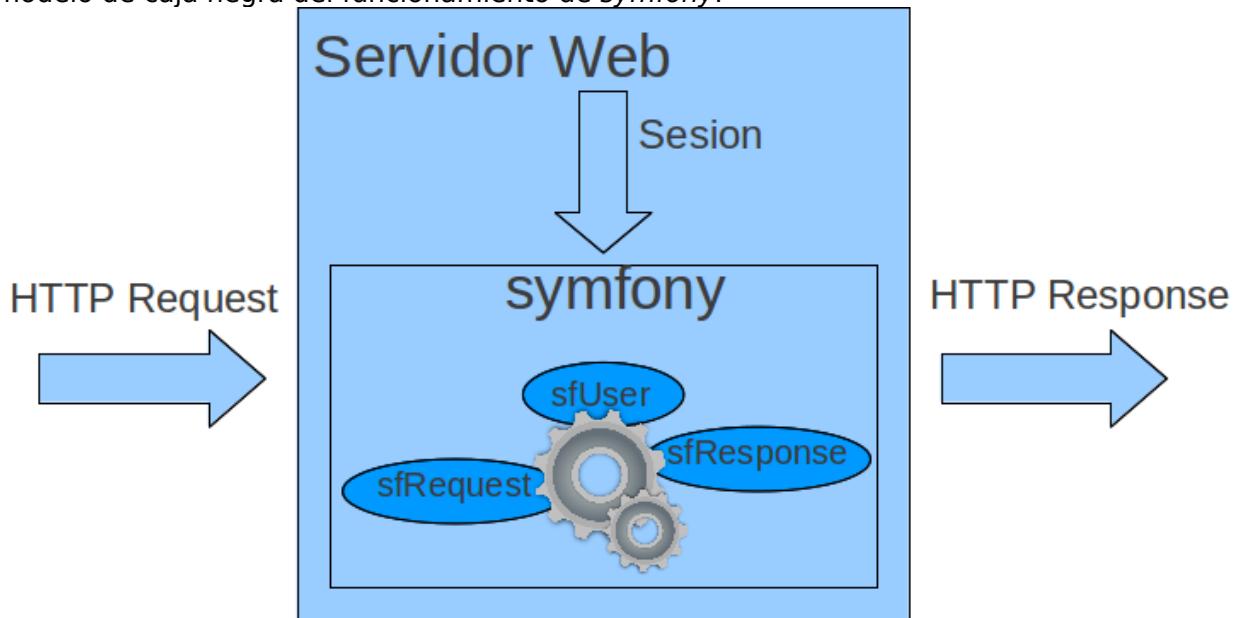
Los filtros y las acciones

siguiente. Un diagrama de secuencia describe con precisión el fundamento de este sencillo pero eficaz patrón de diseño:



Como vemos cada filtro realiza algunas operaciones durante un tiempo y pasa la ejecución al siguiente filtro que repite el procedimiento: realizar sus operaciones y pasar el testigo al siguiente filtro hasta llegar al último, que en el caso de *symfony* es el encargado de ejecutar la acción y renderizar la plantilla correspondiente. Fíjate también que una vez que el último filtro termina su actividad el control pasa al filtro anterior, recorriendose ahora la cadena en sentido contrario. Es decir, que una vez ejecutada la acción se vuelve a pasar de nuevo por los filtros (segunda pasada), por ello la ejecución de la acción forma un “emparedado” con los filtros. Durante toda la ejecución de los filtros tenemos disponibles el objeto que modela la petición *HTTP* del cliente (`sfRequest`), el que modela la respuesta *HTTP* que se enviará al cliente (`sfResponse`) y el que modela la sesión de usuario (`sfUser`). Manipulando estos objetos tanto en la acción como en los filtros podemos conseguir cualquier cosa que se nos ocurra. De manera un tanto informal podemos decir que el principal objetivo de la ejecución del *framework* es construir progresivamente un objeto respuesta a partir de los datos que se reciben en la petición, del estado representado en la sesión y, por supuesto, de la lógica de negocio que decide qué debe hacer con estas entradas. El siguiente gráfico ilustra este

modelo de caja negra del funcionamiento de *symfony*.



La secuencia de filtros se establece en el fichero de configuración de la aplicación `apps/nombre_aplicacion/config/filters.yml`. Ábrelo y échale un vistazo. Comprobarás la referencia a cuatro de los filtros que aparecen en la figura. El quinto filtro (*misFiltros*) representa, en realidad, a tantos filtros como el programador desee añadir. Normalmente no es necesario ninguno, pero a veces pueden resultar muy útiles.

Nota

Recurso: En esta URL puedes encontrar una explicación de los filtros de *symfony* realizada por los autores de *symfony*:

http://www.librosweb.es/symfony_1_2/capitulo6/filtros.html

Otro aspecto de las acciones que puede resultar muy útil son las pre-acciones y *post-acciones*. Supongamos que en algún módulo todas las acciones necesitan realizar alguna actividad común antes de pasar a su ejecución. Por ejemplo, que todas las acciones deban definir algún parámetro común o necesiten comprobar alguna condición o cualquier otra cosa que se nos ocurra. En tal caso, en lugar de repetir el mismo código al principio de cada acción, que sería la solución inmediata, lo correcto sería colocar dicho código en la *pre-acción* del módulo en cuestión. Esto significa crear una función denominada `preExecute()` en dicho módulo. Lo mismo se haría si en el caso de que el código común tuviese que ejecutarse al final de cada acción, solo que en este caso la función que debemos declarar se llama `postExecute()`:

```
<?php
class moduloActions extends sfActions
{
    public function preExecute()
    {
        //Aquí el código que será ejecutado justo antes de la ejecución de
        //cualquier acción del módulo
```

Asociación de la plantilla a la acción

```
}

public function postExecute()
{
    //Aquí el código que será ejecutado justo después de la ejecución de
    //cualquier acción del módulo
}

// Aquí las acciones
}
```

Asociación de la plantilla a la acción

Finalizaremos el estudio de las acciones aclarando como se produce la asociación de la plantilla a la acción. Hasta ahora hemos visto que a una acción denominada *miaccion* le corresponde una plantilla denominada *miaccionSuccess.php*. Symfony utiliza el valor devuelto por la acción para saber que plantilla debe utilizar para pintar los datos. Si nosotros no indicamos el valor devuelto por la acción, como de hecho ocurre en todas las acciones que hemos implementado hasta el momento, el valor devuelto por defecto es *sfView::SUCCESS*, de ahí el nombre de la plantilla utilizada. Sin embargo podemos cambiar este valor y el framework utilizará otra plantilla distinta para mostrar los datos. La siguiente tabla muestra los valores devueltos que se permiten en una acción y la plantilla asociada:

Valor devuelto en la acción	Nombre de la plantilla utilizada para renderizar los datos
<i>return sfView::SUCCESS</i>	<i>{nombre_accion}Success.php</i>
<i>return sfView::ERROR</i>	<i>{nombre_accion}Error.php</i>
<i>return sfView::ALERT</i>	<i>{nombre_accion}Alert.php</i>
<i>return sfView::INPUT</i>	<i>{nombre_accion}Input.php</i>
<i>Return 'MiResultado'</i>	<i>{nombre_accion}MiResultado.php</i>
<i>return sfView::NONE</i>	<i>No se utiliza ninguna vista.</i>
<i>return sfView::HEADER_ONLY</i>	<i>Envía al cliente únicamente las caberas HTTP</i>

Finalmente, si queremos que la acción sea dibujada por una plantilla específica que no se corresponda con el nombre de la acción, debemos utilizar el método *setTemplate()*, el cual podemos combinar con los anteriores valores de retorno.

Así pues el siguientes código al final de una acción:

```
<?php
//Código de una acción
...
$this -> setTemplate('otraPlantilla');
```

Produciría la renderización con la plantilla *otraPlantillaSuccess.php*, mientras que el siguiente código:

```
<?php
//Código de una acción
...
```

Implementación de un filtro para la selección de CSS en función del perfil del usuario.

```
$this -> setTemplate('otraPlantilla');
return sfView::ERROR;
```

Produciría la renderización con la plantilla *otraPlantillaError.php*.

En conclusión, podemos utilizar cualquier plantilla que deseemos para renderizar la acción. Eso sí, la plantilla debe estar preparada para pintar los parámetros que se hayan definido en la acción. La flexibilidad que *symfony* nos ofrece a la hora de construir nuestras aplicaciones sigue poniéndose de manifiesto a medida que avanzamos en el curso.

Implementación de un filtro para la selección de CSS en función del perfil del usuario.

Para ilustrar el uso de los filtros de *symfony*, vamos a incorporar a nuestro gestor documental una nueva funcionalidad que no contemplamos en el análisis de la aplicación. Se trata de utilizar distintas CSS's en función del perfil que tenga asociado el usuario. Esta nueva funcionalidad resultará muy atractiva y resultaría aun más útil si un mismo usuario pudiese tener asociado más de un perfil, ya que por el aspecto gráfico que muestra la aplicación el usuario sabría el perfil con el que se encuentra trabajando.

En primer lugar crearemos una CSS para cada perfil. Recuerda que hemos dividido los estilos en tres archivos CSS. Para los propósitos de este ejemplo únicamente cambiaremos el archivo *default.css*. Creamos cuatro copias de dicho archivo y las denominamos *default_invitado.css*, *default_lector.css*, *default_autor.css* y *default_administrador.css*:

```
# cp web/css/default.css web/css/default_invitado.css
# cp web/css/default.css web/css/default_lector.css
# cp web/css/default.css web/css/default_autor.css
# cp web/css/default.css web/css/default_administrador.css
# rm web/css/default.css
```

Ahora cambiamos los estilos definidos en las anteriores CSS's para particularizarlos al perfil. Como se trata de un ejemplo pedagógico únicamente cambiaremos el color del fondo del elemento *body*, asignando los siguientes colores a cada perfil:

Perfil	Color
Invitado	#1F8CB5
Lector	#E3A114
Autor	#B4F2A2
Administrador	#E890AD

Se trata de modificar el atributo *background-color* en la línea 117 de los ficheros *default_{nombre_perfil}.css*.

Ahora creamos el filtro como una clase denominada *FiltroCSS* y la ubicamos en el directorio *lib* de la aplicación:

Contenido del archivo: /apps/frontend/lib/FiltroCSS.class.php

```
<?php
class FiltroCSS extends sfFilter
```

Implementación de un filtro para la selección de CSS en función del perfil del usuario.

```
{  
    public function execute($filterChain)  
    {  
        if($this -> isFirstCall())  
        {  
            $user = $this->getContext()->getUser();  
            $perfil = ($user -> hasAttribute('perfil'))? $user -> getAttribute('perfil') : 'invitado';  
  
            $respuesta = $this -> getContext() -> getResponse();  
            $respuesta -> addStylesheet('default_'.$perfil);  
        }  
  
        //Ejecutar el próximo filtro  
        $filterChain->execute();  
    }  
}
```

En este filtro se utiliza la función `isFirstCall()` para garantizar que únicamente se ejecute una vez en el caso de que se haya realizado una redirección desde otra acción. Además, todos los filtros deben terminar con una llamada al siguiente filtro, lo cual se hace en la línea:

```
<?php  
  
$filterChain -> execute();
```

El filtro detecta el tipo de perfil que tiene el usuario consultando la sesión, y en función del resultado obtenido añade a la respuesta la hoja de estilos correspondiente.

Ya casi lo tenemos. Ahora debemos indicar a `symfony` que incluya este filtro en su cadena de filtros. Para ello modificamos el archivo `apps/frontend/config/filters.yml` de la siguiente manera:

Contenido del archivo: apps/frontend/config/filters.yml

```
rendering: ~  
security: ~  
  
# insert your own filters here  
css:  
    class: FiltroCSS  
  
cache: ~  
execution: ~
```

El texto en negrita muestra el código añadido. Ya únicamente nos queda hacer una cosa, eliminar del fichero de configuración de la vista la hoja de estilos `default.css`, ya que con los cambios realizados ha dejado de utilizarse.

Contenido del archivo: apps/frontend/config/view.yml

```
default:  
    http_metas:  
        content-type: text/html  
  
metas:  
    title:      Gestor Documental  
    description: Un gestor documental construido con symfony para un curso de Mentor  
    keywords:   symfony, gestor_documental, mentor
```

```
language:      es
robots:       index, follow

stylesheets:   [ admin.css, menu.css]

javascripts:  []

has_layout:   true
layout:       layout
```

Ya está. Ahora puedes comprobar el funcionamiento registrándose con distintos usuario que tengan asociados distintos perfiles y verás como cambia el color del fondo de la aplicación.

La parte de la vista

Desde el principio del curso hemos trabajado los conceptos de *layout* de la aplicación y plantillas (o *templates*) de los módulos, introduciendo nuevos aspectos a medida que los necesitábamos. En este apartado, igual que hemos hecho con la parte del controlador, trataremos más de cerca la parte de la vista.

En las aplicaciones *web*, la mayor parte de las respuestas que el servidor envía al cliente, contienen como datos una representación *HTML* del recurso solicitado, ya que es el lenguaje de marcado que entienden los navegadores *web*. Sin embargo esto no tiene por qué ser siempre así. En ocasiones el servidor *web* puede enviar un fichero de cualquier tipo. Es lo que hace nuestro gestor documental cuando se le solicita una versión de un documento. En otras ocasiones se pueden enviar otro tipo de representaciones, siendo el *XML* uno de los lenguajes de marcados más utilizados en las aplicaciones *web* gracias a su capacidad para el intercambio de datos entre aplicaciones. Es decir, es un formato muy adecuado para ser procesado por las máquinas facilitando la interoperabilidad entre las mismas. Por esa razón es el lenguaje más utilizado para la implementación y consumo de servicios *web*. *JSON* es otro de los formatos de intercambio de datos de fuerte presencia en las aplicaciones y servicios *web*, especialmente cuando la información que se transmite tiene que ver con estructuras de datos y objetos software que deben ser ejecutados en el cliente. A medida que la *web* semántica extienda su popularidad, posiblemente en muy poco tiempo, el formato *RDF* entrará de lleno en la escena como otro sistema de representación de recursos. El *PDF* también goza de buena fama cuando de imprimir documentos se trata. En definitiva, no solo de *HTML* vive la *web*, y *symfony*, como *full stack framework* para el desarrollo de aplicaciones *web*, ofrece la posibilidad de generar vistas más allá del *HTML*.

Este apartado lo hemos dividido en dos partes diferenciadas; la primera trata de la generación de vistas orientadas a la presentación en navegadores *web*, es decir, representaciones *HTML* de los recursos. Y la segunda de la generación de vistas en representaciones no *HTML* (todas las demás). Queremos dejar claro que, a pesar de esta división metodológica, un mismo recurso puede ser representado en cualquier tipo de formato que podamos imaginar, siendo los más usuales el *HTML* (para mostrar en los navegadores), el *PDF* (para imprimir), el *XML* (para casi cualquier cosa, servicios *web* como ejemplo ilustrativo) y el *JSON* (para enviar objetos software al cliente).

La vistas *HTML*

El proceso de decoración. Los layouts.

La generación de vistas en *symfony* se realiza según lo establecido por otro patrón de diseño denominado *decorator*. Este patrón, de nombre bastante descriptivo, responde a la necesidad de añadir dinámicamente funcionalidad a un objeto. En *symfony* dicho objeto sería la plantilla con la que se renderiza una determinada acción de algún módulo, y la

Uso de javascripts y CSS's. Los ficheros de configuración view.yml

funcionalidad añadida dinámicamente sería el resto del documento *HTML*, definido en alguno de los ficheros alojados en el directorio *apps/nombre_aplicacion/templates*, es decir en alguno de los *layouts* de la aplicación.

Recordemos el gráfico que utilizamos en la unidad 2 para explicar el concepto de generación de la vista como combinación de una plantilla y un *layout*, pues ilustra bastante bien el concepto de decoración.



Hasta el momento únicamente hemos hablado de un solo fichero donde se define el *layout* de la aplicación, ya que este se utiliza por defecto para decorar las plantillas y, ha sido suficiente para cubrir los objetivos de nuestra aplicación. Sin embargo podemos cambiar este comportamiento en las acciones e indicar otras plantillas para **decorarlas**. Para ello utilizamos el método *setLayout()* en la acción en cuestión :

```
<?php  
// Código dentro de una acción  
...  
$this -> setLayout('otroLayout');  
...
```

Obviamente, para que el trozo de código anterior tuviese efecto, tenemos que definir en el directorio reservado para los *layouts* de la aplicación (*apps/nombre_aplicacion/templates*), el fichero *otroLayout.php* con la definición del mismo.

Uso de javascripts y CSS's. Los ficheros de configuración view.yml

Los documentos *HTML* están estructurados en dos partes principales: la cabecera entre las etiquetas *<head></head>* y el cuerpo entre las etiquetas *<body></body>*. En la cabecera se coloca la meta-information que describe el documento en sí, se pueden incluir enlaces a recursos *javascripts* que serán utilizados en el navegador cliente, y enlaces a los ficheros *CSS's* que se utilizan en el cuerpo para dotar a los elementos visibles de un determinado aspecto gráfico.

Si queremos incluir ficheros *CSS's* y/o *javascript* a los documentos *HTML* generados por nuestra aplicación debemos indicarlo explícitamente en el *layout* mediante las funciones *include_stylesheets()* y *include_javascripts()*. Aclaramos: estas funciones indican al framework que el *layout* en cuestión “desea” utilizar *CSS's* o *Javascript*, no especifica ningún archivos *CSS* o *Javascript* en concreto. Esto último se puede hacer de las distintas formas que explicaremos a continuación.

1. En el archivo *view.yml* de la aplicación (*apps/nombre_aplicacion/config/view.yml*). Si queremos que unas *CSS's* y/o unos *Javascripts*, sean incluidos en todos los documentos generados por la aplicación, o dicho de otra manera, que estén disponibles para todas

Uso de javascripts y CSS's. Los ficheros de configuración view.yml

las acciones de todos los módulos, podemos utilizar las secciones *stylesheets* y *javascript* del archivo *view.yml* de la aplicación para incluirlos.

2. En los archivos *view.yml* de los módulos (*apps/nombre_aplicacion/modules/nombre_modulo/config/view.yml*). Si lo que queremos es que únicamente los documentos que resulten de la ejecución de las acciones de algún módulo incluyan ciertas CSS's y/o Javascripts, entonces las secciones *styleheets* y *javascript* de los ficheros *view.yml* de los módulos son los lugares donde podemos especificarlos.
3. En las plantillas de los módulos a través de las funciones *use_stylesheet()* y *use_javascript()*. Si lo hacemos de esta forma, las CSS'S y/o Javascripts, serán incluidos únicamente en los documentos *HTML* generados a partir de las acciones que utilicen la plantilla en cuestión.
4. Utilizando directamente el objeto *sfResponse*. Como ya hemos dicho en otro momento, el objeto *sfResponse* modela la respuesta *HTTP* que se envía al cliente al final del proceso. Desde las acciones podemos acceder directamente a dicho objeto mediante el método *getResponse()*:

```
<?php  
  
// Trozo de código en una acción  
  
...  
  
$respuesta = $this -> getResponse();  
  
$respuesta -> addStylesheet('mi_hoja_de_estilo.css');  
$respuesta -> addJavascript('mi_javascript.js');  
...
```

Como puedes imaginar, la hojas de estilo referenciada en el código anterior debe estar ubicada en el directorio *web/css*, y el fichero *javascript* en el directorio *web/js*.

Desde los filtro debemos acceder a través del contexto general de la aplicación:

```
<?php  
  
// Trozo de código en un filtro  
  
...  
  
$respuesta = $this -> getContext() -> getResponse();  
$respuesta -> addStylesheet('mi_hoja_de_estilo.css');  
...
```

Tal y como hemos hecho en el filtro implementado en un apartado anterior.

Como puedes ver la flexibilidad de *symfony* sigue en aumento. La forma en que añadas las CSS's y/o javascripts dependerá de la situación en concreto. Si utilizas únicamente el archivo *view.yml* de la aplicación nunca fallarás, pero puede que estés sobrecargando innecesariamente la respuesta y, por tanto, desaprovechando el ancho de banda.

El fichero de configuración *view.yml* de la aplicación, además de para especificar las CSS's y javascripts comunes a toda la aplicación, se utiliza para incluir la meta-information que va en la sección *head* del fichero *HTML* y los parámetros de la respuesta *HTTP* como el *content-type*:

Asociación de la vista a la acción.

Contenido del archivo: /apps/nombre_aplicacion/config/view.yml

```
default:
    http_metas:
        content-type: text/html

    metas:
        title:      symfony project
        description: symfony project
        keywords:   symfony, project
        language:   en
        robots:     index, follow

    stylesheets:  [main.css]
    javascripts: []
    has_layout:   true
    layout:       layout
```

Por último, el parámetro `has_layout` indica al *framework* si debe decorar las acciones (`true`) o no (`false`).

Asociación de la vista a la acción.

El mecanismo de asociación entre la acción y las vista ha sido explicado en el apartado 2.3 de esta misma unidad, correspondiente a la parte controladora. Hemos incluido este apartado con el fin de mostrar que dicho mecanismo es algo que también pertenece a la parte de la vista. Podemos decir que dicho mecanismo ofrece el punto de comunicación entre el controlador y la vista. Obviamente no vamos a repetir la explicación y remitimos al estudiante al apartado 2.3 de esta misma unidad.

Los helpers

Ya hemos utilizado algunos *helpers* a lo largo del curso. Ahora los definiremos con precisión, presentaremos los más usuales y explicaremos como puedes producir tus propios *helpers*.

Los *helpers* no son más que funciones de *PHP* que devuelven una cadena con código para el cliente, normalmente código *HTML* o *javascript*. Se pueden utilizar tanto en las plantillas de los módulos como en los *layouts* de la aplicación. Estas criaturas se agrupan en librerías según su propósito. Si echamos un vistazo al directorio *helper* del núcleo de *symfony* vemos los siguientes ficheros que representan estas agrupaciones:

- *AssetHelper.php*
- *CacheHelper.php*
- *DateHelper.php*
- *DebugHelper.php*
- *EscapingHelper.php*
- *HelperHelper.php*
- *I18NHelper.php*
- *JavascriptBaseHelper.php*
- *NumberHelper.php*

Asociación de la vista a la acción.

- *TextHelper.php*
- *UrlHelper.php*

Como ves los nombres de los ficheros que contienen helpers terminan con el sufijo *Helper*. La regla general es que si deseamos utilizar algún *helper*, debemos incluir en la plantilla el nombre del fichero (sin el sufijo *Helper*) que lo contiene mediante la función *use_helper()*. Por ejemplo si queremos usar en una plantilla el *helper* *format_date()* que está en el archivo *DateHelper*, colocaríamos al principio de la plantilla el siguiente código:

```
<?php use_helper('Date') ?>
...
<?php echo format_date(date(), 'd', 'es') ?>
...
```

Es decir, el nombre del fichero sin el sufijo.

Sin embargo este no es el caso de los ficheros *HelperHelper.php*, *TagHelper.php*, *UrlHelper.php* y *AssetHelper.php*, que se incluyen automáticamente en el *framework* ya que son necesarios para el mecanismo de plantillas.

Dos de los *helpers* más utilizados, pertenecientes al fichero *UrlHelper.php* son *link_to()* y *url_for()*, que sirven para generar enlaces (*links*) *HTML* y rutas válidas para el servidor donde se ejecuta la aplicación a partir de los nombres del módulo y de la acción en combinación con los parámetros de la *query string*. Por ejemplo:

```
<?php echo url_for('mimodulo/miaccion?param1=valor1&param2=valor2') ?>
```

Daría lugar una vez interpretado por *PHP* a algo así:

`http://miservidor/miruta/web/index.php/mimodulo/miaccion/param1/valor1/param2/valor2`

En realidad, la salida exacta depende de cómo se hayan definido las rutas en el sistema de enrutamiento de *symfony*. Es la primera vez que hablamos en el curso del sistema de enrutamiento. Aunque su uso explícito no es estrictamente necesario para desarrollar una aplicación *web* con *symfony*, si recurrimos a él remataremos la aplicación con un conjunto de *URL's* que, además de limpias, ocultan los nombres de los parámetros que se pasan por *GET* al servidor, ocultando por tanto detalles de implementación interna de la aplicación. Esto, como podrás suponer, redonda en una mejora considerable de la seguridad. Hablaremos del sistema de enrutamiento en la última unidad del curso donde se tratarán algunos temas avanzados como la internacionalización y el enrutamiento.

Otro *helper* muy útil es *image_tag()*, el cual arroja el código *HTML* para la inclusión de una imagen:

```
<?php echo image_tag('miimagen.png', 'alt=imagen size=200x100') ?>
```

Daría lugar a algo así:

```

```

Obviamente la imagen debe estar ubicada en el directorio *web/images*.

Por último vamos a explicar como puedes crear tus propios *helpers*. Creas un fichero con el nombre tu *helper* seguido del sufijo *Helper* en el directorio *lib* que sea más adecuado para tus propósitos (normalmente será el de la aplicación). En su interior defines funciones que

Partials y componentes.

deben devolver una cadena con un trozo de código *HTML*, *javascript*, *XML* o lo que sea. A continuación lo incluyes en la plantilla que donde necesites alguna de estas funciones. Para ello utilizas la función *use_helper()*, igual que si se tratase de un *helper* de “serie”. Y punto y final.

Partials y componentes.

¿Recuerdas el principio *DRY: Don't Repeat Yourself*, del que hemos hablado varias veces a lo largo del curso? Los *partials* y componentes son recursos mediante los que podemos modularizar y reutilizar los elementos de la vista. Una forma natural de llegar a ellos es a través de la sana costumbre de refactorizar continuamente el código que se escribe. Cuando veamos partes de la vista que se repiten en muchos lugares, seguramente llevar tales partes a un *partial* o a un componente nos resultará de gran ayuda.

Un *partial* es una plantilla que puede ser incluida en cualquier otra plantilla o en un *layout*. Un ejemplo vale más que mil palabras y eso es lo que vamos a hacer para explicar este concepto. Vamos a introducir en todas las páginas de la aplicación un enlace para que se registre el usuario si aún no lo ha hecho, y para que se desconecte de la aplicación cuando le apetezca si ya se identificó.

Un *partial*, como plantilla que es, puede ubicarse en el directorio *templates* de cualquier módulo. Hemos decidido ubicarlo en el módulo *inises*, puesto que también deseamos utilizar este enlace de registro en la aplicación de *backend* y, como ya hemos dicho en otro momento, este módulo será convertido más adelante en un *plugin* para que pueda ser compartido por ambas aplicaciones. El único requisito que una plantilla debe reunir para que sea un *partial* es que el nombre del fichero que la define comience con el carácter “_”. De esa manera podrá ser incluida como parte de otra plantilla cualquiera, y no estará asociada a ninguna acción particular. Por último para incluir un *partial* en otra plantilla o en un *layout* se utiliza la función *include_partial()*.

Vamos a verlo en la práctica. Crea el fichero *_signInOut.php* en el directorio *templates* del módulo *inises* con el siguiente código:

Contenido del archivo: */apps/frontend/modules/inises/templates/_signInOut.php*

```
<?php if($sf_user -> isAuthenticated()) :?>
<?php echo link_to('desconectar', 'inises/signOut') ?>
<?php else : ?>
<?php echo link_to('registro', 'inises/signIn') ?>
<?php endif; ?>
```

Este código comprueba si el usuario está autenticado (observa el uso de la variable *\$sf_user*, recuerda que es la manera de acceder desde una plantilla al objeto que representa la sesión de usuario), en cuyo caso crea un enlace para realizar la desconexión, y en caso contrario crea un enlace para realizar el proceso de autentificación. Fíjate, en primer lugar en el uso del helper *link_to()* para crear los enlaces y, en segundo lugar en que ambas acciones ya la hemos implementado y probado a través de la barra de direcciones del navegador en el tema 6. Ahora vamos a incorporarlas a la aplicación gracias a este *partial*. Dicha incorporación la realizaremos en el *layout* de la aplicación utilizando, como hemos dicho anteriormente, la función *include_partial()*.

Contenido del archivo: *apps/frontend/templates/layout.php*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <?php include_http_metas() ?>
```

Partials y componentes.

```
<?php include_metas() ?>
<?php include_title() ?>
<?php include_stylesheets() ?>
<?php include_javascripts() ?>
<link rel="shortcut icon" href="/favicon.ico" />

</head>
<body>
    <div id="contenedor_general">
        <div id="cabecera">
            <div id="logo"></div>
        </div>

        <div id="wrapper">
            <div id="perfil"><?php include_partial('inises/signInOut') ?></div>

            <div id="menuprincipal"></div>

            <div id="admin_container"><?php echo $sf_content ?></div>
        </div>

        <div class="PiePagina">
            <ul>
                <li><a href="#" title="Aviso legal" target="">Aviso legal</a>|</li>
                <li><a href="http://www.w3.org/WAI/" title="Accesibilidad" target="">Accesibilidad</a>|</li>
                <li>
                    <a href="http://www.w3.org/WAI/" title="Logo de la WAI" target="">
                        <?php echo image_tag('valid-xhtml10.png', array('alt' => 'Accesibilidad web', 'width' => '50'))?>
                    </a>
                </li>
            </ul>
            <p>
                <a href="#" title="© Juan David Rodríguez" target="#">© Juan David Rodríguez</a>
                <br/>
                <a href="#" title="Mentor Soft" target="_blank">Mentor Soft</a>
            </p>
            <p>
                Información general : <a href="mailto:#" title="Contacte con el webmaster">webmaster at gmail dot com</a>
                <br/>
            </p>
        </div>
    </div>
</body>
</html>
```

Se ha resaltado en negrita el código añadido para incluir el *partial* que acabamos de construir. Como puedes ver, la función *include_partial()* espera como primer argumento una cadena que indica el nombre del *partial* (sin el carácter “_”) y el módulo donde reside. Ahora puedes probar el funcionamiento del invento. Si no estás aún registrado, cuando pinchas en registrar te aparece la ventana de registro y ya puedes registrarte. En función del perfil que tenga el usuario le aparecerán más o menos documentos y más o menos acciones como consecuencia de las credenciales asociadas.

A los *partials* también se le pueden pasar parámetros cuando se los incluye con la función *include_partial()*. Para ello se puede usar como segundo argumento de esta función un *array* asociativo en el que las claves son los nombres de las variables que usará el *partial*.

El componente es aún más poderoso que el *partial*. En pocas palabras: es un *partial* con una acción asociada de manera que los datos que pinta son el resultado de un proceso más complejo que suele ser el resultado de consultar al modelo. Dicha acción en el lenguaje de *symfony* se denomina componente y se implementa en un archivo llamado *components.class.php* ubicado en el directorio *actions* del módulo. El *partial* asociado, por su parte, se ubica en el directorio *templates* del mismo módulo y su nombre debe comenzar por el carácter “_”. El funcionamiento del componente, por lo tanto, sigue las directivas del patrón *MVC*, solo que aplicado a una porción de la vista. Para incluir el componente en una plantilla o en un *layout* se utiliza la función *include_component()*, cuyo primer argumento debe ser el nombre del módulo donde reside el componente, el segundo es el nombre del componente, y el tercero (opcional) un *array* asociativo con los parámetros que pueda necesitar el componente.

Para ilustrar este concepto, desarrollaremos un componente mediante el que integraremos en todas las pantallas de nuestra aplicación un mensaje de bienvenida con el nombre y el perfil asociado del usuario que la está utilizando, obviamente siempre que esté registrado en la aplicación, y si no lo está indicaremos con el mismo componente su condición de invitado.

Por la misma razón que en el caso del *partial* de registro, construiremos el componente en el módulo *inises*. Comenzamos por crear el fichero que alojará al componente donde

Partials y componentes.

escribiremos el siguiente código:

Contenido del archivo: *apps/frontend/modules/inises/actions/components.class.php*

```
<?php

class inisesComponents extends sfComponents
{
    public function executeMostrarPerfil()
    {
        $user = $this -> getUser();

        if($user -> isAuthenticated())
        {
            $id_usuario = $user -> getAttribute('id_usuario');

            $usuario = UsuariosPeer::retrieveByPK($id_usuario);

            $this -> nombre = $usuario -> getNombre();
            $this -> perfil = $usuario -> getPerfil();
        }
        else
        {
            $this -> nombre = 'usuario';
            $this -> perfil = 'invitado';
        }
    }
}
```

Observa que la estructura es idéntica a la de un fichero de acciones; incluso las funciones que actúan como componentes llevan el prefijo “execute”. Sin embargo existe una sutil diferencia respecto de las acciones, y es que al componente no se le pasa como argumento la petición *HTTP* (el objeto *sfWebRequest*). No obstante, si lo necesitamos, podemos acceder a la misma mediante el método *getRequest()*. De la misma manera podemos acceder a la respuesta mediante *getResponse()* y, como se muestra en el código anterior, también a la sesión de usuario mediante *getUser()*.

Ahora tenemos que implementar la plantilla (partial) asociada al componente, la cual debe denominarse igual que el componente pero cambiando el prefijo *execute* por el carácter “_”. Escribimos en ella el siguiente código:

Contenido del archivo: *apps/frontend/modules/inises/templates/_mostrarPerfil.php*

Bienvenido <?php echo \$nombre ?> | <?php echo \$perfil ?> |

Y para finalizar lo incluimos en el *layout* de la aplicación, justo antes del enlace para el registro que hemos desarrollado más atrás:

Porción del archivo: *apps/frontend/templates/layout.php*

```
<div id="perfil"><?php echo include_component('inises', 'mostrarPerfil') ?><?php include_partial('inises/signInOut') ?></div>
```

Si nuestro componente necesitase algún parámetro podemos utilizar el tercer argumento de la función *include_component()*, que es un array asociativo cuyas claves se convertirán en miembros públicos del componente en cuestión y que, por consiguiente, podrán ser accedidos mediante el identificador *\$this*.

Las vistas no HTML

Remataremos el estudio de la parte de la vista *HTML* añadiendo a la aplicación un menú cuyos enlaces dependerán de las credenciales asociadas al perfil, de la manera que se especifica en esta tabla:

Credencial	Menú
Cualquiera	Enlace a la ayuda
Escritura	Se le añade un enlace para crear un nuevo documento.
Administración	Se le añade un enlace para enlazar con la aplicación de administración.

Lo haremos a través del *partial* siguiente, cuya explicación se deja como ejercicio al alumno:

Contenido del archivo: *apps/frontend/templates/_menu.php*

```
|<?php if($sf_user -> hasCredential('escritura')) : ?>
<?php echo link_to('Nuevo Documento', 'gesdoc/nuevo') ?> |
<?php endif; ?>
<?php echo link_to('Ayuda', 'gesdoc/ayuda') ?> |
<?php if($sf_user -> hasCredential('administracion')) : ?>
<a href="#">Administración</a> |
<?php endif; ?>
<hr/>
```

Que se incorpora al *layout* de la aplicación en la capa reservada para el menú:

Porción del archivo: *apps/frontend/templates/layout.php*

```
<div id="menuprincipal">
  <?php include_partial('global/menu') ?>
</div>
```

Este *partial*, al no ser específico de ningún módulo, lo hemos colocado directamente en la carpeta *template* de la aplicación. Los *partials* que no pertenecen a ningún módulo se refieren como si perteneciesen a un módulo denominado *global*.

Las vistas no HTML

Como ya hemos advertido, no solo de *HTML* vive la *web*, especialmente cuando entran en la escena los servicios *web*. Hemos de tener en cuenta que *HTTP* es un protocolo mediante el que se pueden transmitir cualquier tipo de archivo, aunque los documentos *HTML* sean los más populares debido a que son los que pueden ser interpretados y pintados por los navegadores *web*. Por ello se pueden desarrollar, como de hecho se hace cada vez con más frecuencia, aplicaciones *web* que utilicen otro tipo de representación, fundamentalmente *XML*, de los recursos que sirven. A este tipo de aplicaciones se les conoce como servicios *web*. Por lo general podemos decir que las aplicaciones *web* típicas están concebidas para ser consumidas directamente por humanos y usan intensivamente el *HTML* para la representación de los recursos, mientras que los servicios *web* buscan la interoperabilidad entre máquinas y usan, preferentemente, como formato de intercambio el *XML*. Este criterio ha llevado a algunos autores a proponer los términos *web humana* y *web programable* para designar a los dos espacios *web* que surgen cuando se toma como criterio el tipo de consumidor de los recursos: el hombre o la máquina.

Symfony es un *framework* completo para la construcción de todo tipo de aplicaciones *web*, incluido los servicios *web*, por tanto ofrece un soporte nativo para distintos tipos de

Canales de noticias RSS.

representaciones, de manera que podemos utilizar el mismo controlador y modelo para arrojar vistas con distintos formatos como *txt*, *js*, *css*, *json*, *xml*, *rdf* o *atom*. En este apartado veremos como cambiar el comportamiento de la vista para proporcionar estos tipos de ficheros. Como venimos haciendo a lo largo del curso, lo haremos añadiendo a la aplicación que estamos desarrollando una nueva funcionalidad: La incorporación de un canal de noticias *RSS* para mostrar las últimas versiones de los documentos públicos subidos al repositorio.

Canales de noticias RSS.

No vamos a entrar en una descripción detallada de lo que son los canales de noticias *RSS* y la redifusión de contenidos. Contamos con que el estudiante ya sabe algo acerca del tema, y si no sabe nada suponemos que tiene la suficiente curiosidad y capacidad de autoaprendizaje para adquirir un mínimo de conocimientos que le permita seguir el desarrollo de este apartado. No obstante como introducción presentamos el resumen del artículo de la *wikipedia*.

“RSS es una familia de formatos de fuentes web codificados en XML. Se utiliza para suministrar a suscriptores de información actualizada frecuentemente. El formato permite distribuir contenido sin necesidad de un navegador, utilizando un software diseñado para leer estos contenidos RSS (agregador). A pesar de eso, es posible utilizar el mismo navegador para ver los contenidos RSS. Las últimas versiones de los principales navegadores permiten leer los RSS sin necesidad de software adicional. RSS es parte de la familia de los formatos XML desarrollado específicamente para todo tipo de sitios que se actualicen con frecuencia y por medio del cual se puede compartir la información y usarla en otros sitios web o programas. A esto se le conoce como redifusión web o sindicación web (una traducción incorrecta, pero de uso muy común).” Fuente: <http://es.wikipedia.org/wiki/RSS>

Fundamentalmente, para la implementación de nuestro canal *RSS* lo que hay que saber es que tenemos que elaborar un recurso representado en *XML* según la especificación *RSS* en el que cada *item* será una versión de un documento público almacenado en nuestro gestor documental.

Especificando el formato de la petición.

Como hemos visto hasta el momento, por defecto las acciones se renderizan con plantillas con estructura *HTML* que son decoradas con un *layout* determinado. El nombre de la plantilla sigue el patrón: *{nombre_accion}{valor_devuelto}.php*,

por ejemplo: *indexSuccess.php*.

Sin embargo, y esto lo decimos ahora por primera vez, esto es una simplificación del nombre completo que es: *{nombre_accion}{valor_devuelto}.html.php*,

de manera que el ejemplo anterior sería: *indexSuccess.html.php*. Lo que ocurre es que *symfony*, si la plantilla no lleva información del formato, supone que se corresponde con *HTML*.

Así pues, como puedes imaginar, si el formato de salida no fuese *HTML*, el patrón que sigue el nombre de la plantilla es: *{nombre_accion}{valor_devuelto}.{formato}.php*,

por ejemplo, *indexSuccess.xml.php* representa una plantilla *XML*, *indexSuccess.json.php* es una plantilla *JSON*, ...

Ahora bien, ¿cómo sabe *symfony* qué formato debe aplicar y, por tanto con qué formato de plantilla debe renderizar la acción? En este apartado descubriremos una parte de la verdad, dejaremos la otra parte para el momento en que hablaremos del mecanismo de enrutamiento ya que tiene que ver con este. Se trata de especificar explícitamente en la acción el tipo de petición mediante el método *setRequestFormat()* del objeto *sfRequest*:

Canales de noticias RSS.

```
<?php  
  
// trozo de código de una acción  
...  
$request -> setRequestFormat('xml');  
...
```

Si la acción que vamos a implementar va a ser renderizada siempre con un mismo formato basta con que lo indiquemos según acabamos de ver. Pero también puede ocurrir que deseemos representar la misma acción con distintos formatos, para lo cual tendremos que construir tantas plantillas como formatos vayamos a utilizar. La selección de un formato u otro se hará en función de lo que indique algún parámetro de la petición. Para este último caso lo más correcto es utilizar el mecanismo de enrutamiento, ya que proporciona una manera directa de asociar distintas representaciones según la forma que presente la *URL*. Pero esto lo veremos más adelante.

Para los propósitos que perseguimos será suficiente con implementar una acción que recoja una lista ordenada por fecha, desde la más actual a la más antigua, con todas las versiones de los documentos públicos. Y construir una plantilla XML asociada a dicha acción que cumpla el estándard de redifusión RSS.

Añadimos la acción *rss* al módulo *gesdoc*:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php  
  
public function executeRSS(sfWebRequest $request)  
{  
    $request -> setRequestFormat('xml');  
  
    $c = new Criteria();  
    $c -> add(DocumentosPeer::PUBLICO, 1);  
    $c -> addJoin(DocumentosPeer::ID_DOCUMENTO, VersionesPeer::ID_DOCUMENTO);  
    $c -> addDescendingOrderByColumn(VersionesPeer::FECHA_SUBIDA);  
  
    $this -> versiones = VersionesPeer::doSelect($c);  
}
```

Fíjate en la línea resaltada en negrita. Como ya hemos explicado dicha línea indica a *symfony* que la acción debe ser renderizada con una plantilla en formato XML. Creamos el archivo *rssSuccess.xml.php* con el siguiente contenido:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/rssSuccess.xml.php*

```
<?php echo "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" ?>  
<rss version="2.0">  
  <channel>  
    <title>Canal RSS del Gestor Documental del Curso de Symfony de Mentor.</title>  
    <link>http://www.mentor.net/esc</link>  
    <description>Este canal RSS ofrece información actualizada sobre los documentos públicos que se van incorporando al gestor documental del curso de symfony de mentor</description>  
    <image>  
      <url>http://localhost/curso_symfony/gestordocumental-1.4/web/Images/logo.png</url>  
      <link>http://localhost/curso_symfony/gestordocumental-1.4/web/frontend_dev.php/gesdoc</link>  
    </image>  
    <?php foreach ($versiones as $v): ?>  
    <item>  
      <title><?php echo $v -> getDocumento() -> getTitulo() ?>. Version: <?php echo $v -> getNumero() ?></title>  
      <link><?php echo "http://localhost/curso_symfony/gestordocumental-1.4/web/frontend_dev.php/gesdoc/rss?version=" . $v -> getIdVersion() ?></link>  
      <description><?php echo $v -> getDocumento() -> getDescripcion() ?>. <?php echo $v -> getDescripcion() ?>. Enviado por: <?php echo $v -> getDocumentos() -> getUsuarios() -> getNombre() ?></description>  
      <pubDate><?php echo $v -> getFechaSubida() ?></pubDate>  
    </item>  
    <?php endforeach; ?>  
  </channel>  
</rss>
```

Y ya podemos probar nuestro canal RSS ejecutando la acción que acabamos de construir introduciendo en la barra de direcciones del navegador la *URL*:

http://localhost/gestordocumental/web/frontend_dev.php/gesdoc/rss

Conclusión

Casi todos los navegadores modernos incluyen un lector de *RSS*, de manera que lo más probable es que al solicitar el canal *RSS* mediante la *URL* anterior obtengas el resultado bien presentado. Si el navegador que utilizas no tiene soporte *RSS*, entonces obtendrás el documento *XML* sin procesar. De todas formas, lo habitual para leer los canales *RSS* es utilizar un programa cliente denominado agregador *RSS*.

Ya que hemos implementado esta funcionalidad, deberíamos mostrarla al usuario en la propia aplicación. Lo que haremos es añadir al menú un enlace al canal *RSS* recién implementado. Utilizaremos además la típica imagen con la que se anuncian en casi todos los sitios *web* la existencia de un canal *RSS*. Se trata de añadir al *partial inises/_menu*, dicho enlace:

Contenido del archivo: *apps/frontend/modules/inises/templates/_menu.php*

```
|<?php if($sf_user -> hasCredential('escritura')) : ?>
<?php echo link_to('Nuevo Documento','gesdoc/nuevoDocumento') ?> |
<?php endif; ?>
<?php echo link_to('Ayuda','gesdoc/ayuda') ?> |
<?php if($sf_user -> hasCredential('administracion')) : ?>
<a href="#">Administración</a> |
<?php endif; ?>
<?php echo link_to(image_tag('rss.gif'),'gesdoc/rss') ?>
<hr/>>
```

Y ahora ya queda expuesto con claridad el canal *RSS*. Fíjate que detrás de toda la terminología y teoría que hay detrás de las *RSS*, al final se trata simplemente de construir un archivo *XML* con unas determinadas etiquetas y devolverlo como respuesta al cliente.

Conclusión

En esta unidad hemos profundizado bastante en la parte controladora y en la vista de *symfony*. Y ello nos ha permitido agregar a la aplicación una serie de funcionalidades que la enriquecen considerablemente.

En la parte controladora hemos mostrado como los filtros nos permiten un control preciso a la hora de manipular la respuesta. Pero la cantidad de cosas que con ellos podemos hacer no está limitada más que por la imaginación del programador. También las *pre-acciones* y *post-acciones* nos permiten organizar el código de la aplicación de una manera muy eficiente.

En la parte de la vista hemos introducido los conceptos de *partial*, componente, y *helpers*, hemos ampliado el concepto de *layout* y la posibilidad de generar vistas en otros formatos distintos del *HTML*. Después de esta unidad debería quedar claro que las posibilidades del *framework* para construir todo tipo de aplicaciones *web* no tienen más límite que el conocimiento que el programador adquiera del mismo y de la imaginación a la hora de combinar los conceptos que hemos estudiado.

Para ilustrar mediante la práctica los conceptos estudiados, se han realizado las siguientes implementaciones: un *partial* que muestra un enlace para que el usuario se registre o se desconecte, un componente que muestra quien está utilizando la aplicación, un menú y un servicio *web RSS* para la difusión de las últimas versiones que van añadiéndose al sitio *web*.

Después de esta unidad el estudiante tiene a su disposición un surtido conjunto de herramientas para desarrollar sus aplicaciones *web*. No dudes en probar todas aquellas ideas y cuestiones que se te hayan podido plantear después de seguir esta unidad. Comprobarás que las posibilidades de todo lo aprendido van más allá de los ejemplos que hemos construido para ilustrar los conceptos.

Unidad 8: El framework de formularios de Symfony

En esta unidad estudiaremos uno de los componentes más potentes y útiles de *symfony*; el sistema de formularios, el cual constituye por sí solo un *framework MVC* independiente pero integrado en el conjunto global de *symfony*. Esto significa que podemos utilizar este componente en otros proyectos *PHP* que no sean *symfony*.

En las primeras versiones de *symfony* los formularios se construían mediante *helpers*, de forma que existía un *helper* para renderizar cada elemento de control propio de un formulario *HTML*. A partir de la versión 1.1, *symfony* presenta un sistema de formularios completamente rediseñado y que constituye una colaboración de clases con dos responsabilidades fundamentales: construir el código *HTML* de los diferentes elementos de control que lo componen, incluyendo las *CSS's* y el código *javascript* que pudieran utilizar, y realizar la validación de los datos enviados por el cliente al servidor a través del formulario. Es decir, dibujan y validan.

En esta unidad haremos uso de los formularios de *symfony* para finalizar la implementación de las funcionalidades que aún faltan en la aplicación *frontend* de nuestro gestor documental, a saber:

- Creación de nuevos documentos.
- Modificación de los metadatos de los documentos.
- Subida de las versiones de los documentos numeradas automáticamente.

Con este objetivo bien definido nos disponemos a seguir rodeando, cada vez más de cerca, las posibilidades de *symfony*.

Formularios HTML

Antes de comenzar el estudio de los formularios de *symfony*, creemos necesario presentar algunos conceptos relativos a la producción y procesamiento de formularios *HTML* en las aplicaciones *web*.

Componentes de la interfaz gráfica

Los documentos *HTML*, a través de los formularios, pueden incorporar elementos de control que el usuario utiliza para insertar datos que son enviados posteriormente al servidor para su procesamiento. En la siguiente tabla mostramos el nombre de dichos elementos de control y un resumen de sus funcionalidades:

Nombre	Descripción
--------	-------------

Unidad 8: El framework de formularios de Symfony

<i>input</i>	Es el elemento más utilizado ya que puede tomar varias formas en función de su atributo <i>type</i> , el cual puede ser: <ul style="list-style-type: none"> • <i>text</i>: un texto en una línea, • <i>password</i>: un texto enmascarado, • <i>checkbox</i>: una casilla de selección que puede o no estar marcada. Si hay varias que comparten el mismo nombre pueden marcarse varias a la vez, • <i>radio</i>: una casilla de selección que puede o no estar marcada. Si hay varias que comparten el mismo nombre solo una puede marcarse a la vez, • <i>submit</i>: un botón para enviar el formulario al servidor, • <i>reset</i>: un botón para limpiar el formulario, • <i>file</i>: un seleccionador de ficheros • <i>hidden</i>: un campo oculto a la interfaz HTML, • <i>image</i>: un botón con una imagen asociada, • <i>button</i>: un botón para disparar una acción en el cliente.
<i>button</i>	Es como el <i>button</i> creado con el elemento <i>input</i> pero ofrece al programador más posibilidades para su renderizado. Puede ser de tres tipos: <ul style="list-style-type: none"> • <i>submit</i>, para enviar un formulario al servidor, • <i>reset</i>, para limpiar el formulario y • <i>button</i>, para disparar acciones en el cliente.
<i>select</i>	Es un menú mediante el cual el usuario puede elegir una o varias opciones. Este elemento siempre lleva asociado uno o más elementos <i>options</i> .
<i>options</i>	Cada una de las opciones de un <i>select</i> .
<i>textarea</i>	Es una caja de texto que puede ocupar más de una linea, es decir, una caja de texto multilinea.
<i>form</i>	Es el elemento que actúa como contenedor de los anteriores elementos. El atributo <i>action</i> especifican el <i>script</i> en el servidor que procesará los datos introducidos por el usuario, y el atributo <i>method</i> el método <i>HTTP</i> que se usará para su envío: <i>GET</i> o <i>POST</i> .

Así pues un formulario *HTML* está constituido por un elemento *form* que encierra a varios elementos de control. Alguno de estos elementos debe ser un botón de envío (*submit*) que, al ser pulsado por el usuario provoca, según se haya especificado, una petición *GET* o *POST* al recurso del servidor indicado en el atributo *action* del formulario, con los datos introducidos por el usuario.

Combinando adecuadamente los elementos de control *HTML* se puede modelar prácticamente cualquier componente típico de una interfaz gráfica de usuario, aunque el resultado es bastante pobre si lo comparamos con las interfaces gráficas de las típicas aplicaciones de escritorio. Sin embargo, si en combinación con el *HTML* utilizamos hojas de estilo para mejorar el aspecto gráfico de los elementos de control y código *javascript* para aportar interactividad y efectos visuales, podremos crear cualquier tipo de componente de control que se nos ocurra: barras de progresos, *sliders*, navegación por pestañas, menús

Procesamiento de los datos introducidos en el formulario.

avanzados, navegación tipo acordeón, incluso podemos inventarnos nuevos componentes. Cuando se utilizan CSS's en combinación con código javascript para mejorar la interfaz de las aplicaciones web se dice, de una manera muy descriptiva, que se ha **enriquecido** el HTML. Muchos de los productores de los sitios web más famosos en la actualidad, como pueden ser *facebook* o *google*, incorporan, cada vez más, interfaces Enriquecidas a sus aplicaciones. Como veremos, una de las ventajas de utilizar los formularios de *symfony* es que podemos crear y encapsular en objetos reutilizables componentes de control combinando elementos HTML, CSS's y javascript.

Procesamiento de los datos introducidos en el formulario.

La función de los formularios HTML es recoger datos del usuario cliente y enviarlos al servidor para hacer algo con ellos (almacenarlos en una base de datos, realizar un complejo cálculo para confeccionar una carta astral o cualquier otra cosa).

El navegador web envía los datos realizando una petición HTTP al servidor, la cual, según la especificación del protocolo HTTP, puede ser de varios tipos en función de lo que el cliente tenga intención de hacer con los datos enviados sobre el servidor. Las operaciones HTTP más conocidas que un servidor web pone a disposición de los clientes a través de su *interfaz uniforme* son *GET*, *POST*, *PUT* y *DELETE*, la cuales pueden ser comparadas semánticamente con las operaciones de una base de datos:

Método HTTP	Operación en base de datos	Descripción
GET	Retrieve	Recupera un recurso/registro. No hay modificación del mismo
POST	Update	Modifica un recurso/registro.
PUT	Create	Crear un recurso/registro
DELETE	Delete	Elimina un recurso/registro

Los navegadores web actuales tan sólo soportan las dos primeras operaciones (*GET* y *POST*) para realizar peticiones al servidor y enviarle datos. Ahora bien, ¿cuál de las dos debemos utilizar en nuestros formularios HTML? La respuesta ya la hemos insinuado unas líneas más atrás al comparar los métodos HTTP con las operaciones de una base de datos: si en el proceso de servidor que recibe los datos se va a llevar a cabo algún tipo de modificación (ya sea en bases de datos, ficheros o cualquier otro tipo de recurso) debemos utilizar *POST* que se correspondería con una operación de actualización, pero si dicho proceso utiliza los datos enviados tan solo para recuperar algún recurso, **sin que haya ningún tipo de modificación en los datos que gestiona la aplicación**, entonces debemos utilizar *GET*.

Además de los matices semánticos de ambas operaciones, existe una diferencia bien visible: En el caso de una petición del tipo *GET*, los datos se envían como parámetros que forman parte de la URL, tal y como mostramos en este ejemplo:

`http://www.elservidor.es/recurso?param1=valor1¶m2=valor2`

mientras que en las peticiones *POST* los datos viajan encapsulados en la sección de datos de la petición HTTP. Además mediante *POST* se pueden enviar ficheros al servidor indicando en la petición que el tipo de contenido (*content-type*) de los datos enviados es *multipart/form-data*. Esta indicación se realiza a través del parámetro *enctype* del elemento *form*.

Es muy importante que comprendamos que las peticiones HTTP se realizan desde un cliente sobre el cual, obviamente, el servidor no tiene ningún tipo de control directo, de manera que el cliente puede enviar al servidor los parámetros que quiera, saltándose lo prescrito por el formulario. Expliquemos esto con más detalle. La secuencia normal que seguiría una aplicación web para pedir datos al cliente y procesarlos sería:

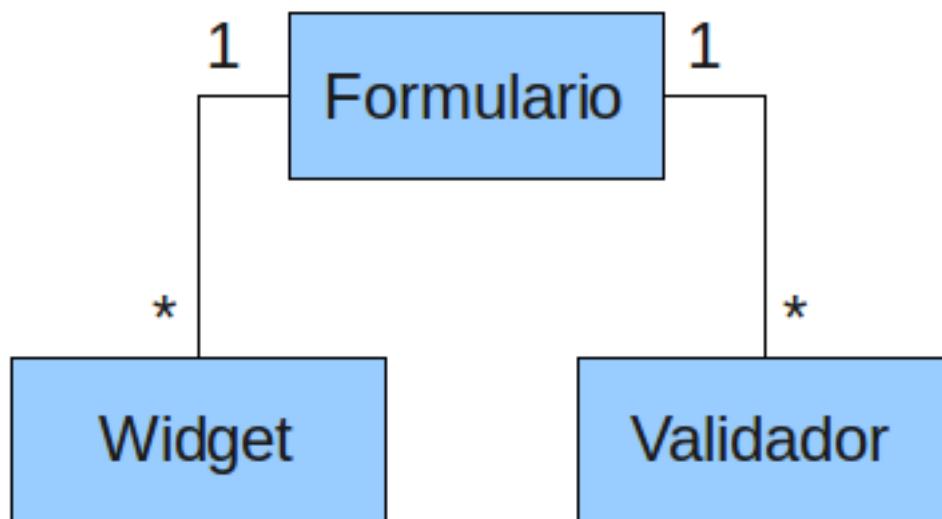
Estructura de los formularios de Symfony

1. El servidor envía un formulario *HTML* al cliente con los elementos de control que el usuario utilizará para introducir los datos.
2. El navegador interpreta el documento *HTML* y presenta el formulario al usuario
3. El usuario introduce los datos y envía el formulario relleno, es decir, pica en el botón *submit* y el navegador construye una petición *HTTP* al proceso de servidor indicado en el parámetro *action* del formulario y con los datos que el usuario ha introducido.
4. El servidor recibe la petición con los datos esperados y los procesa.

Sin embargo no hay nada que impida saltarse el paso 3 y construir en el cliente, sin utilizar el navegador, una petición *HTTP* al proceso de servidor con cualquier tipo de datos. Esto significa que el servidor **nunca** debe fiarse de los datos que traen las peticiones, ya que han podido ser manipuladas en el cliente y no tienen por qué obedecer a lo que el programador espera, dando lugar a posibles brechas de seguridad en la aplicación. Conclusión: **Los procesos de servidor deben validar TODOS los datos que le llegan antes de realizar ninguna operación con ellos**. Observa que el hecho de introducir validadores en el cliente utilizando código *javascript* no vale de nada, ya que como acabamos de indicar, el usuario puede “puentejar” completamente el uso del navegador, y por tanto del formulario enviado, para construir y lanzar la petición con los datos que deseé. Este hecho junto con el desconocimiento, la pereza o la escasa destreza del programador para blindar sus aplicaciones con los adecuados validadores del lado del servidor, dan lugar a una de las vulnerabilidades más comunes de las aplicaciones *web*. Los validadores de *symfony*, que forman parte del *framework* de formularios, ofrecen una elegante y obligatoria solución a este problema.

Estructura de los formularios de Symfony

Los formularios de *symfony* están compuestos por dos tipos de objetos: los *widgets* y los validadores. Los primeros sirven para realizar la presentación del elemento de control en el documento *HTML*, y por tanto en el navegador, y los segundos para realizar la validación de los datos que llegan en las peticiones *HTTP* al servidor. El siguiente diagrama de clases *UML* representa la estructura de un formulario *symfony*.



El programador utiliza los métodos del formulario tanto para la presentación del mismo como para su validación. Internamente el formulario se encarga de realizar la coordinación entre los *widgets* y los validadores.

Los widgets

Para el estudio de los formularios describiremos la estructura y el funcionamiento de *widgets* y validadores de manera independiente, fuera del formulario. De hecho podemos utilizar en nuestras aplicaciones unos y otros directamente, sin necesidad de incorporarlos en un formulario. Posteriormente veremos como se definen los formularios asociándoles *widgets* y validadores y cómo se utilizan en las aplicaciones web construidas con *symfony*.

Los widgets

Los *widgets* de *symfony* son objetos que derivan de la clase *sfWidgetForm*. Esta clase define una interfaz común mediante la que se pueden realizar las operaciones necesarias para la definición y renderizado del *widget*.

La siguiente tabla muestra algunos de los *widget* más utilizados. Todos ellos derivan de la clase base *sfWidgetForm*.

Nombre del widget	Funcionalidad
<i>sfWidgetFormInput</i>	Es una caja de texto de una sola línea. Representa un elemento <i>HTML input</i> del tipo <i>text</i>
<i>sfWidgetFormInputCheckBox</i>	Es una casilla para marcar/desmarcar. Representa un elemento <i>HTML input</i> del tipo <i>checkbox</i>
<i>sfWidgetFormInputHidden</i>	Es un elemento que contiene un dato oculto al navegador. Representa un elemento <i>HTML input</i> del tipo <i>hidden</i>
<i>sfWidgetFormInputPassword</i>	Es una caja de texto para introducir datos enmascarados. Representa un elemento <i>HTML input</i> del tipo <i>password</i>
<i>sfWidgetFormInputFile</i>	Es una caja de texto con un botón que un navegador para buscar e incorporar un fichero que será enviado en la petición. Representa un elemento <i>HTML input</i> del tipo <i>file</i>
<i>sfWidgetFormTextArea</i>	Es una caja de texto multilínea. Representa un elemento <i>HTML textarea</i>
<i>sfWidgetFormChoice</i>	En realidad este <i>widget</i> está compuesto por los cuatro <i>widgets</i> mostrados en el diagrama anterior y, en función de como lo configuremos, delegará el renderizado al que le corresponda. Es un <i>widget</i> que se utiliza para realizar operaciones de selección, a través de menús desplegables de selección simple o múltiple, conjuntos de <i>radiobuttons</i> o conjuntos de cajas de selección. Por lo tanto puede representar distintos tipos de elementos de selección <i>HTML</i> .
<i>sfWidgetFormDate</i>	Es un <i>widget</i> mediante el que se pueden introducir fechas en distintos formatos de localización según como se haya configurado. Representa tres elementos <i>HTML</i> de tipo <i>select</i> , uno para cada elemento de la fecha: día, mes y año.

Todos los tipos de *widget* tienen dos propiedades importantes; las opciones (*options*) y los atributos (*attributes*). Las primeras se utilizan para configurarlo y la segunda representan los atributos *HTML* que se asociarán al elemento de control *HTML* que será dibujado por el *widget*. Por otro lado, el método esencial de cualquier *widget* se denomina *render()* y sirve para arrojar el código *HTML* que le corresponda en virtud de su tipo y de su definición (opciones + atributos).

Los widgets

¿Cómo se utiliza un *widget* en la práctica? Para explicarlo vamos a suponer que deseamos mostrar en alguna de las vistas de nuestro proyecto una caja de texto simple y una entrada para fechas, para lo cual utilizamos los *widget* *sfWidgetFormInput*, y *sfWidgetFormDate*.

Nota

para realizar el ejemplo vamos a crear una acción y una vista asociada en el módulo *gesdoc* de la aplicación *frontend* que más tarde eliminaremos puesto que no forma parte de la aplicación.

Comenzamos declarando en la acción los objetos *sfWidgetFormInput* y *sfWidgetFormDate*:

Trozo del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
public function executePruebaWidget(sfWebRequest $request)
{
    $this -> wInput = new sfWidgetFormInput();
    $this -> wInput -> setOptions(array('default' => 'prueba'));
    $this -> wInput -> setAttributes(array('class' => 'mi clase', 'onblur' => "alert('hola')"));

    $this -> wDate = new sfWidgetFormDate(array('format' => '%day% - %month% - %year%'), array('class' => 'fecha'));
}
...
```

En este ejemplo hemos definido dos *widgets* de dos formas equivalentes. En el caso del *sfWidgetFormInput*, primero lo declaramos y después le asociamos las opciones y los atributos mediante los métodos *setOptions()* y *setAttributes()*. En el caso del *sfWidgetFormDate* realizamos la declaración y la configuración en un solo paso facilitando como argumentos del constructor del objeto dos *arrays*, el primero con las opciones y el segundo con los atributos. Repetimos, ambas formas son equivalentes. Además hay que advertir que cada *widget* tiene sus propias opciones. La mejor manera de conocer cuales son las opciones que acepta cada *widget* es consultando directamente el código fuente donde se definen. Dicho código lo puedes encontrar en el directorio *widget* del núcleo de *symfony*.

Ahora podemos utilizar estos objetos en la plantilla correspondiente para dibujar tantas cajas de texto y entradas de fechas como queramos. Para hacer esto utilizamos el método *render()*, el cual admite tres argumentos: el nombre del *widget*, el valor por defecto, y los atributos *HTML* que deseemos añadir a los que ya se han añadido cuando configuramos el *widget*. Únicamente el primero de los argumento es obligatorio y se utiliza para asignar el atributo *name* de los elementos *HTML* correspondientes. Crea la plantilla *pruebaWidgetSuccess.php* con el siguiente contenido:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/pruebaWidgetSuccess.php*

```
<div id="sf_admin_header">
    <h2>Prueba de widgets</h2>
</div>

<div id="sf_admin_content">

    <?php echo $wInput -> render('nombre', ESC_RAW) ?>
    <br/>
    <br/>
    <?php echo $wInput -> render('apellido', 'rodriguez', array('class' => 'otra'), ESC_RAW) ?>

```

Los validadores

```
<br/>
<br/>
<?php echo $wDate -> render('fecha1',ESC_RAW) ?>
<br/>
<br/>
<?php echo $wDate -> render('fecha2',ESC_RAW) ?>

</div>
```

Es importante que observes el efecto de lo que hemos hecho en el código *HTML* que llega al navegador *web*. Fíjate en el valor del atributo *name* de los elementos *HTML*. En el caso de la caja de texto coincide con el primer argumento del método *render()*, pero en el caso de las entradas de fecha no coincide, si no que forma parte del nombre. Esto no puede ser de otra manera ya que la entrada de fechas consta de tres cajas de texto, y cada una debe tener un nombre único para ser identificada cuando los datos lleguen al servidor. Así que el *widget* asigna como nombre para la caja de los días *fecha1[day]*, para la caja de los meses *fecha1[month]* y para la de los años *fecha1/year*.

Nota

Es un buen momento para que juegues con este ejemplo cambiando opciones y parámetros y probando nuevos *widgets*. En esta *URL* encontrarás abundante documentación acerca de los *widgets* de *symfony*, de su configuración y renderizado: http://librosweb.es/symfony_formularios/capitulo12.html

Los validadores

Los validadores de *symfony* son objetos que derivan de la clase *sfValidatorBase*. Esta clase define una interfaz común mediante la que se pueden realizar las operaciones necesarias para la validación de los *widgets*. Además de realizar la validación de los datos que llegan al servidor, estos objetos también se encargan de limpiarlos según se les haya indicado en su configuración.

La siguiente tabla muestra algunos de los validadores más utilizados. Todos ellos derivan de la clase base *sfValidatorBase*.

Nombre del validador	Funcionalidad
<i>sfValidatorString</i>	Comprueba que el dato enviado es una cadena de caracteres
<i>sfValidatorEmail</i>	Comprueba que el dato enviado es una cadena de caracteres que se corresponde con una dirección de correo electrónico
<i>sfValidatorInteger</i>	Comprueba que el dato enviado es un valor entero
<i>sfValidatorNumber</i>	Comprueba que el dato enviado es un valor numérico (<i>float</i>)
<i>sfValidatorDate</i>	Comprueba que el dato enviado se corresponde con una fecha
<i>sfValidatorFile</i>	Comprueba que el dato enviado es un fichero válido. Además, como veremos más adelante, en caso de que el fichero cumpla lo que la configuración del validador exige, devuelve un objeto del tipo <i>sfValidatedFile</i> que encapsula al fichero y proporciona una gestión sencilla del mismo.

Los validadores

Todos los tipos de validadores tienes dos propiedades importantes; las opciones (*options*) y los mensajes (*messages*). Las opciones se utilizan para especificar los requisitos del validador, es decir para definir cómo deben ser los datos válidos, y los mensajes sirven para indicar el mensaje que se mostrará si el dato no cumple lo exigido.

El método principal de un validador se denomina *clean()* y lleva a cabo las siguientes acciones:

1. Limpia el valor de espacios en blancos antes y después del dato si la opción *trim* se ha especificado.
2. Chequea si el dato está vacío.
3. Comprueba si el dato cumple lo que requiere la configuración del validador.

En todos los casos el validador devuelve el valor limpio del dato en caso de que sea válido, y si no lo es lanza una excepción con el mensaje que se haya indicado en la configuración.

Ahora vamos a mostrar el funcionamiento de los validadores en la práctica. Para ello creamos una nueva acción de prueba que denominaremos *executePruebaValidadores* y su plantilla asociada *pruebaValidadoresSuccess.php*:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
public function executePruebaValidadores(sfWebRequest $request)
{
    $strValidator = new sfValidatorString();

    $strValidator -> setOptions(array('required' => true, 'max_length' => '6', 'trim' => true));
    $strValidator -> setMessages(array('required' => 'el dato es requerido', 'max_length' => '%value% es demasiado larga, el máximo es 6'));

    $dato = " hola";
    $this -> dato_limpio = $strValidator -> clean($dato);
}
```

Contenido del archivo:
apps/frontend/modules/gesdoc/templates/pruebaValidadoresSuccess.php

```
<div id="sf_admin_header">
    <h2>Prueba de validadores</h2>
</div>

<div id="sf_admin_content">
    dato limpio:<?php echo $dato_limpio ?>
</div>
```

Como puedes ver primero hemos definido un validador y después lo hemos configurado indicando que el valor es requerido (no puede estar vacío), que debe tener una longitud máxima de 6 caracteres y que hay que limpiarlo de espacios. También se han definido los mensajes de error que la excepción debe mostrar cuando la validación no sea satisfactoria. Para ello hemos utilizado los métodos *setOptions()* y *setMessages()* pero, de la misma forma que ocurre con los *widgets*, podríamos haber realizado la declaración del validador y su configuración al mismo tiempo:

```
<?php
$strValidator = new sfValidatorString(
    array('required' => true, 'max_length' => '6', 'trim' => true),
    array('required' => 'el dato es requerido', 'max_length' => '%value% es demasiado larga, el máximo es 6')
);
```

Después hemos definido una variable *valor* que contiene una cadena y la validamos usando el método *clean()* del validador. En la plantilla simplemente mostramos el valor devuelto por

Los formularios

el validador. Si ejecutas la acción verás que se muestra la cadena “hola” sin espacios al principio. Prueba ahora a cambiar el valor de la variable dato por una cadena vacía y después por una cadena que tenga más de 6 caracteres. Verás como se lanza una excepción con el mensaje que hemos definido para cada caso.

Llegados a este punto debemos de aclarar que aunque hemos mostrado como utilizar los validadores independientemente, lo normal es utilizarlos a través de los formularios que estudiaremos en el siguiente apartado. En tal caso no es necesario utilizar el método `clean()` ya que el formulario se encarga de gestionar el validador de manera transparente.

Nota

Es un buen momento para que juegues con este ejemplo cambiando opciones y parámetros y probando nuevos validadores. En esta *url* encontrarás abundante documentación acerca de los validadores de *symfony* de su configuración y funcionamiento:

http://librosweb.es/symfony_formularios/capitulo13.html

Los formularios

Y por fin los formularios. Como ya hemos indicado anteriormente un formulario de *symfony* se compone de *widets* y validadores. Una vez declarado y definido, el formulario es un objeto que utilizamos para tres funciones distintas:

1. Presentarlos en la vista como un formulario *HTML*.
2. Presentar los errores, si los hubiera, en cada uno de los campos que no hayan pasado la validación.
3. Validar en el servidor los datos enviados desde el cliente a través de una petición *HTTP*.

Nota

Desde la versión 1.3 los formularios proporcionan un mecanismo de seguridad para evitar un tipo de ataque denominado *cross-site request forgeries (csrf)* y que es “*un tipo de exploit malicioso de un sitio web* en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía”* (fuente: wikipedia). Este ataque se evita haciendo que el servidor, cuando envía un formulario al cliente, pase un campo oculto con un *token* de seguridad generado para ese formulario y usuario en concreto. Cuando el cliente envía el formulario el servidor comprueba si se ha devuelto ese mismo *token*, de esa manera se asegura de que la respuesta se ha realizado desde el mismo formulario que se envió originalmente. Este tipo de vulnerabilidad es muy esquiva y sutil y no vamos a estudiarla en profundidad. Simplemente debemos saber que *symfony* ofrece por defecto un mecanismo para combatirla que se puede desactivar eliminando el parámetro *csrf_secret* del archivo de configuración *apps/nombre_aplicacion/config/setting.yml*. En realidad, utilizar esta funcionalidad en nuestros formularios, como veremos, es prácticamente transparente en la programación, por tanto recomendamos su uso si quieres mejorar la seguridad de tu aplicación.

Los formularios

Ilustraremos el funcionamiento de los formularios con un ejemplo sencillo que nos permitirá comprender el flujo de acciones que tiene lugar en cualquier operación en la que una aplicación *web* solicita un conjunto de datos al cliente para realizar algún tipo de proceso. Las operaciones, a alto nivel, que tienen lugar en un lado y otro son las descritas en este esquema:

1. El servidor envía el formulario *HTML* al cliente.
2. El usuario introduce los datos en el formulario que le muestra el navegador y lo envía de nuevo al servidor.
3. El servidor valida los datos
4. Si los datos son válidos se procesan y se devuelve al cliente algún mensaje indicando que la operación se ha realizado con éxito.
5. Si los datos no son válidos el servidor envía de nuevo el formulario al cliente con los datos que este ya envió y con los mensajes de error que indican qué datos han provocado el rechazo y por qué.

Supongamos que deseamos pedir al usuario su nombre y su correo electrónico para hacer cualquier cosa con estos datos (almacenarlos en una base de datos, por ejemplo). Comenzamos por definir el formulario que satisface lo requerido, el cual se declara como una clase que deriva de la clase base *sfForm*. Siguiendo la estructura de directorios de *symfony*, el lugar para declarar el formulario es alguno de los directorios *lib*; el del proyecto, el de la aplicación o el del módulo, depende de cual sea el alcance que deseemos darle. En este ejemplo lo vamos a definir en el directorio *lib* de la aplicación:

Contenido del archivo: *apps/frontend/lib/FormularioEjemplo.class.php*

```
<?php

class FormularioEjemplo extends sfForm
{
    public function configure()
    {
        $this -> setWidgets(array(
            'nombre' => new sfWidgetFormInput(),
            'email'  => new sfWidgetFormInput(),
        ));

        $this -> widgetSchema -> setNameFormat('contacto[%s]');

        $this -> setValidators(array(
            'nombre' => new sfValidatorString(
                array('required' => true, 'max_length' => 40),
                array('required' => 'Este campo no se puede dejar en blanco',
                      'max_length' => 'el nombre es demasiado largo, 40 caracteres máximo'),
                'email' => new sfValidatorEmail(
                    array('required' => true),
                    array('required' => 'Este campo no se puede dejar en blanco'))
            ));
    }
}
```

Como puedes observar en el código anterior, para definir un formulario basta con declarar un método llamado *configure()*, y definir el conjunto de *widgets* y de validadores que compondrán el formulario. Tanto un conjunto como otro se corresponden con un *array* asociativo en el que la clave será el nombre del campo en el documento *HTML*, en nuestro caso *nombre* y *email*. Tanto el *array* de *widgets* como el de validadores deben tener los mismos valores de las claves, ya que estas representan los nombres de los campos del formulario, y de esta manera se podrá realizar la correspondencia entre el *widget* y el validador.

Un detalle importante a la hora de organizar los datos y, como veremos más tarde imprescindible para llevar a cabo la validación de los datos, es declarar un formato de nombres. Esto se hace en el código anterior en la línea siguiente:

Los formularios

Contenido del archivo: *apps/frontend/lib/FormularioEjemplo.class.php*

```
<?php  
...  
$this -> widgetSchema -> setNameFormat('contacto[%s]');  
...
```

El efecto de esta línea es que los nombres de cada uno de los controles que componen el formulario en el documento *HTML* enviado al cliente, seguirán el siguiente formato: *contacto[nombre_campo]*. Esta característica permite que, una vez devueltos al servidor, los datos que provienen de un mismo formulario son los elementos de un *array* asociativo cuyas claves son los nombres de los campos. De esta manera es mucho más sencillo manipularlos. Por ejemplo se pueden recorrer rápidamente haciendo uso de un *foreach*.

Ahora creamos una acción en el módulo *gesdoc* de la aplicación *frontend* que llamaremos *executePruebaFormulario*:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php  
...  
public function executePruebaFormulario(sfWebRequest $request)  
{  
    $this -> formulario = new FormularioEjemplo();  
}
```

Y lo dibujamos en la plantilla correspondiente:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/pruebaFormularioSuccess.php*

```
<div id="sf_admin_header">  
    <h2>Prueba de formulario</h2>  
</div>  
  
<div id="sf_admin_content">  
    <form name="form" action="<?php echo url_for('gesdoc/pruebaFormulario') ?>" method="post">  
        <?php echo $formulario -> renderHiddenFields() ?>  
        <?php echo $formulario -> renderGlobalErrors() ?>  
        <?php echo $formulario ?>  
        <input type="submit" />  
    </form>  
</div>
```

Ahora puedes probar la acción que acabamos de escribir. Verás que el formulario se pinta pero no queda demasiado bien. Esto es así por que lo hemos lanzado de un “tirón” y el objeto se pinta, obviamente, sin tener en cuenta la estructura de nuestra *CSS's* (sería demasiado listo si lo hiciera). Sin embargo esta forma sencilla de pintar el formulario nos puede venir bastante bien en la primera etapa del desarrollo de una aplicación, cuando lo que deseamos es probar la funcionalidad sin tener en cuenta el diseño.

A continuación vamos a manipular el objeto formulario en la vista para acceder a las distintas partes que lo componen: la etiqueta, el control, el mensaje de error, los mensajes globales y los campos ocultos. De esta manera podremos encajarlo con cualquier estructura *HTML* que se necesite para utilizar una *CSS* determinada.

La plantilla anterior quedaría de la siguiente manera:

Los formularios

Contenido del archivo:
apps/frontend/modules/gesdoc/templates/pruebaFormularioSuccess.php

```
<div id="sf_admin_container">
    <h1>Formulario</h1>

    <div id="sf_admin_header">
        <div class="notice">Mensaje de advertencia</div>
    </div>

    <div id="sf_admin_content">
        <div class="sf_admin_form">
            <form name="form" action="<?php echo url_for('gesdoc/pruebaFormulario') ?>" method="post">
                <?php echo $formulario -> renderHiddenFields() ?>
                <?php echo $formulario -> renderGlobalErrors() ?>
                <fieldset id="fieldset_1">
                    <h2>Contacto</h2>

                    <div class="sf_admin_form_row">
                        <?php echo $formulario['nombre'] -> renderError() ?>
                        <div>
                            <?php echo $formulario['nombre'] -> renderLabel() ?>
                            <?php echo $formulario['nombre']->render() ?>
                        </div>
                    </div>

                    <div class="sf_admin_form_row">
                        <?php echo $formulario['email'] -> renderError() ?>
                        <div>
                            <?php echo $formulario['email'] -> renderLabel() ?>
                            <?php echo $formulario['email']->render() ?>
                        </div>
                    </div>

                    </fieldset>
                    <input type="submit" />
                </form>
            </div>
        <div id="sf_admin_footer">
        </div>
    </div>
```

Aunque esta nueva plantilla es algo más compleja que la anterior, fíjate que la información dinámica es la misma y se obtiene con los métodos `renderLabel()`, `renderError()` y `render()` de cada uno de los campos que componen el formulario. Lo demás es pura cobertura de diseño `HTML` necesaria para una correcta visualización con las `CSS's` que utilizamos. Los nombres de los método son autodescriptivos y no vamos a explicarlos para evitar redundancia. Si diremos que es muy importante, si utilizamos la protección contra ataques `CSRF`, utilizar el método `renderHiddenFields()`, ya que es el encargado de arrojar el campo oculto con el token `csrf`.

Ya tenemos el formulario en el cliente. Ahora toca recibir los datos y procesarlos siempre que estos sean válidos según los especificado en la declaración del formulario. Vamos a utilizar la misma acción con la que hemos enviado el formulario para recibir sus datos. Fíjate en el atributo `action` del formulario para comprobarlo. Esto no tiene por que ser así, es la estrategia que aquí seguiremos y con la que vamos a construir un esquema general mediante el que se pueden tratar casi todos los casos de solicitud y envío de datos a través de formularios en aplicaciones `web` construidas con `symfony`.

Modificamos, por tanto, la acción `executePruebaFormulario()` de manera que sirva tanto para el envío del formulario como para la recepción y procesamiento de los datos devueltos por el cliente. El código siguiente muestra dichos cambios:

Trozo de código del archivo: apps/frontend/modules/gesdoc/actions/actions.class.php

```
<?php
...

```

Creación y modificación de documentos.

```
public function executePruebaFormulario(sfWebRequest $request)
{
    $this -> formulario = new FormularioEjemplo();

    if($request -> isMethod('post')) // La acción ha sido invocada por el envío de un formulario
    {
        $datos = $request -> getParameter('contacto'); // $datos es un array con los datos de la petición
        $this -> formulario -> bind($datos); // asociamos los datos de la petición al formulario
        if($this -> formulario -> isValid()) // Y comprobamos la validez de los datos
        {
            // Aquí procesamos los datos. Por lo pronto solo los mostramos
            // en una plantilla construida para ello

            $this -> nombre = $this -> formulario -> getValue('nombre');
            $this -> email = $this -> formulario -> getValue('email');
            $this -> setTemplate('muestraDatos');
        }
    }
}
```

En negrita se han resaltado los cambios necesarios para implementar la lógica que hemos propuesto anteriormente. En primer lugar se comprueba si la petición es del tipo *POST*, si no lo fuera el resultado de la acción es equivalente a la original, es decir se pasa el formulario (vacío) a la plantilla y se envía el resultado al cliente. Pero si el tipo de petición es *POST* significa que la acción ha sido invocada por el envío del formulario. Entonces, usando el método *bind()* del formulario, se realiza la asociación entre el formulario recién definido y los datos enviados desde el cliente que se encuentran encapsulados en un *array* asociativo. Accedemos a estos datos, envueltos en la petición *HTTP*, a través del objeto *\$request*. Una vez realizada la asociación le pedimos al formulario que valide los datos asociados según los requisitos especificados en los validadores del formulario. Para lo cual usamos el método *validate()*. Si la validación tiene éxito se procesan los datos. En nuestro ejemplo simplemente los pasamos a una plantilla para que se muestren en el cliente. Si los datos no son válidos entonces se vuelve a pintar el formulario, pero esta vez los métodos *renderError()* de los campos que han fallado en la validación arrojan el mensaje que indica la causa del error. Simple y elegante.

A continuación mostramos el contenido de la plantilla *muestraDatosSuccess.php* necesaria para finalizar nuestro ejemplo.

Código de la plantilla: *apps/frontend/modules/gesdoc/templates/muestraDatosSuccess.php*

```
<div id="sf_sf_admin_container">
    <h1>Datos</h1>

    <div id="sf_sf_admin_content">
        Hola <?php echo $nombre ?>, este es tu e-mail: <?php echo $email ?>
    </div>
</div>
```

Pues bien, lo estudiado hasta ahora es suficiente para desarrollar cómodamente las funcionalidades que nos faltan para completar la aplicación *frontend* de nuestro gestor documental. Observa en los siguientes apartados cómo la estructura esencial del envío del formulario, la validación y el proceso de los datos es exactamente la misma a la que acabamos de explicar y desarrollar en este apartado. A pesar de que la cantidad de código será obviamente mayor, debido a las complicaciones propias del proceso de datos.

Creación y modificación de documentos.

Los requisitos del gestor documental enunciaban que los usuarios con perfil autor podrían crear nuevos documentos caracterizados por los siguientes metadatos:

- título

El formulario DocumentosForm

- descripción
- autor del documento
- ¿es público?
- Categorías

Además cada documento podría tener asociado muchos ficheros que representan las distintas versiones del mismo. Las versiones de cada documento tendrían una numeración consecutiva que sería asignada automáticamente mediante un proceso descrito en la unidad 4. Por otro lado, los autores podrían editar los metadatos pero las versiones no podrían ser eliminadas ni, por tanto, los documentos. En este punto es importante que vuelvas a repasar la unidad 4 y que la tengas dispuesta para ser consultada durante el desarrollo del resto de esta unidad.

Lo primero que haremos es definir los formularios que necesitamos; uno para introducir los datos de los documentos y otro para las versiones.

El formulario DocumentosForm

El usuario autor, cuando vaya a crear un nuevo documento tendrá que introducir los metadatos y, opcionalmente, el archivo que se corresponde con la primera versión del mismo. Cuando estos datos lleguen al servidor y sean validados favorablemente, el proceso asociado a la creación del nuevo documento insertará un registro en la tabla *documentos* y, si se ha subido un fichero, otro en la tabla *versiones* relacionado con el primero. Además el fichero físico subido al servidor se alojará en un directorio asociado al autor que lo ha enviado y se le asignará automáticamente un nombre según el proceso que ya hemos mencionado anteriormente.

El siguiente formulario, que denominaremos *FormularioDocumentos* y que ubicaremos en el directorio *lib* de la aplicación *frontend*, contiene los campos necesarios para que el usuario comunique a la aplicación la creación de un documento nuevo.

Contenido del archivo: *apps/frontend/lib/FormularioDocumentos.class.php*

```
<?php

class FormularioDocumentos extends sfForm
{
    public function configure()
    {
        $this->setWidgets(array(
            'id_documento'          => new sfWidgetFormInputHidden(),
            'titulo'                => new sfWidgetFormInputText(),
            'descripcion'           => new sfWidgetFormInputText(),
            'publico'               => new sfWidgetFormChoice(array('choices' => array('no','si'), 'expanded' => false, 'multiple' => false)),
            'id_usuario'             => new sfWidgetFormInputHidden(),
            'id_tipo'                => new sfWidgetFormPropelChoice(array('model' => 'Tipos', 'add_empty' => false)),
            'documento_categoria_list' => new sfWidgetFormPropelChoice(array('multiple' => true, 'model' => 'Categorias')),
            'fichero'                => new sfWidgetFormInputFile()
        ));

        $this->setValidators(array(
            'id_documento'          => new sfValidatorPropelChoice(array('model' => 'Documentos', 'column' => 'id_documento', 'required' => false)),
            'titulo'                => new sfValidatorString(array('max_length' => 255)),
            'descripcion'           => new sfValidatorString(array('max_length' => 255, 'required' => false)),
            'publico'               => new sfValidatorInteger(array('min' => 0, 'max' => 1)),
            'id_usuario'             => new sfValidatorPropelChoice(array('model' => 'Usuarios', 'column' => 'id_usuario')),
            'id_tipo'                => new sfValidatorPropelChoice(array('model' => 'Tipos', 'column' => 'id_tipo')),
            'documento_categoria_list' => new sfValidatorPropelChoice(array('multiple' => true, 'model' => 'Categorias', 'required' => false)),
            'fichero'                => new sfValidatorFile(array('required' => false))
        ));

        $this->widgetSchema->setNameFormat('documentos[%s]');
    }
}
```

Para los campos título y descripción hemos utilizado cajas de texto sencillas (*sfWidgetFormInputText*). Para los campo *id_documento* y *id_usuario* hemos optado por campos ocultos (*sfWidgetFormInputHidden*), ya que sus valores no los decide el autor cuando crea el documento. En efecto, el *id_documento* es la clave primaria del documento que se está creando y será el sistema gestor de base de datos quien lo decida cuando sea almacenado en la tabla correspondiente. Por otro lado, el *id_usuario* debe coincidir con el del autor que está creando el nuevo documento. Recuerda que este último dato (*id_usuario*) lo tenemos disponible en la sesión.

El formulario DocumentosForm

El *id_tipo* debe corresponderse con alguno de los registros de la tabla *tipos*. Por ello utilizamos un *widget* muy útil con el que se pueden crear controles de selección con valores que provienen de una base de datos: el *sfWidgetFormPropelChoice*. Vamos a describir el funcionamiento de este *widget*: En la declaración de sus opciones se especifica el parámetro *model*, que hace alusión a la clase del modelo de donde se pretende seleccionar los datos, en este caso de la clase *Tipos*. Además especificamos mediante el parámetro *add_empty* que deseamos incorporar un elemento vacío al menú de selección. Otro parámetro muy útil, aunque aquí no lo utilizamos, es *criteria*, mediante el cual podemos asociar un criterio de selección para elegir un subconjunto de todos los registros de la tabla en cuestión. Con estos datos, el *sfWidgetFormPropelChoice* construirá un menú de selección en el que los elementos *HTML option* tendrán asociado como valores (atributo *value*) los *id's* correspondientes a la clave principal de la tabla y mostrará al usuario el valor devuelto por el método *_toString()* de cada uno de los objetos que han poblado el control. Por ello, para utilizar este *widget* con un objeto de *Propel* es indispensable definir el método *_toString()* de dicho objeto. En nuestro caso debemos definir el método *_toString()* del objeto *Tipos*:

Contenido del archivo: *lib/model/Tipos.php*

```
<?php

class Tipos extends BaseTipos
{
    public function __toString()
    {
        return self::getNombre();
    }
}
```

Para la selección de las categorías a las que pertenece un documento hacemos lo mismo que con los tipos, usamos el *sfWidgetFormPropelChoice* aplicado a la clase *Categorias*. Además, para que podamos seleccionar varias categorías especificamos en sus opciones el parámetro *multiple* como *true*. Por supuesto debemos definir el método *_toString()* del objeto *Categorias*:

Contenido del archivo: *lib/model/Categorias.php*

```
<?php

class Categorias extends BaseCategorias
{
    public function __toString()
    {
        return self::getNombre();
    }
}
```

Para que el usuario decida si el documento es público o no utilizamos el *widget* de selección *sfWidgetFormChoice* que es similar al anterior. La única diferencia es que los elementos de selección no se obtienen de la base de datos, sino directamente de una *array* que se pasa como opción del *widget* que, en nuestro caso, contiene los valores no y si con índices 0 y 1 respectivamente.

Por último, para que el usuario envíe, si lo desea, un archivo que corresponda a la primera versión del documento, hemos utilizado el *widget sfWidgetFormInputFile*, que se presentará en el navegador un control *HTML input* de tipo *file* para la selección y subida de ficheros

El formulario DocumentosForm

locales.

Segunda parte: los validadores. Recuerda lo que dijimos sobre la desconfianza obligatoria que toda aplicación *web* debe mostrar ante los datos que les llegan en las peticiones. Como puedes observar en la definición del formulario, hemos asociado validadores a todos los campos.

Los campos que han sido poblados con valores de la base de datos son validados con el objeto *sfValidatorPropelChoice*, que se configura pasándole como opción el nombre del objeto de *Propel* que servirá para obtener los registros válidos y el nombre de la columna que se utilizará como valor de validación. Esto ocurre con los campos *id_documento*, *id_usuario* *id_tipo* y *documento_categoria_list*. Observa que ni el campo *id_documento* ni el campo *documento_categoria_list* son requeridos.

Los campos *titulo* y *descripcion* se validan como cadenas de no más de 255 caracteres, siendo el primero obligatorio y el segundo opcional. Hacemos esto mediante un *sfValidatorString*.

El campo *publico* debe ser uno o cero. Eso es lo que exigimos con el validador *sfValidatorInteger*.

Por último el fichero subido al servidor se valida con el objeto *sfValidatorFile*. Como único parámetro especificamos que no es requerido. Sin embargo podemos configurarlo de manera que establezcamos el tamaño máximo y otros aspectos relacionados con el fichero a validar. La mejor forma de estudiar todas las posibilidades de validación tanto de este validador como del resto, así como de los *widgets* es consultando directamente la *API* de *symfony*.

Es importante saber que una vez validados los datos en el servidor mediante el método *isValid()* del formulario, se puede acceder a los valores limpios usando el método *getValue()* del formulario pasándole como argumento el campo cuyo valor limpio deseamos obtener:

```
<?php  
...  
// más atrás el formulario ha sido validado  
$valor_limpio = $formulario -> getValue('nombre_campo');  
...
```

En el desarrollo de las funcionalidades que aún quedan por implementar haremos un uso extensivo del método *getValue()* del formulario, así que conviene comprenderlo bien.

El valor que devuelve el método anterior cuando se aplica a un campo del tipo *sfWidgetFormInputFile*, es un objeto denominado *sfValidatedFile*, el cual representa al fichero subido y facilita una serie de métodos muy útiles para la gestión del mismo. La tabla siguiente muestra los métodos del objeto *sfValidatedFile* que utilizaremos más adelante. Para una referencia completa remitimos de nuevo a la *API* de *symfony*.

Métodos del objeto sfValidatedFile	Descripción
<i>generateFileName()</i>	Genera un nombre aleatorio para el fichero actual
<i>save(\$ruta)</i>	Guarda el fichero en la ruta especificada
<i>isSaved()</i>	Devuelve <i>true</i> si el fichero ya ha sido guardado y <i>false</i> en caso contrario

También dispone de otros métodos para conocer otros aspectos del fichero como el tamaño, la extensión, el nombre original, y otros más. Ya sabes, consulta la *API* para conocerlos y utiliza el que resuelva tu problema concreto.

Implementación de la creación de nuevos documentos

Por último observa que hemos utilizado como formato de nombres para los campos del formulario el patrón *documentos[%s]*.

Implementación de la creación de nuevos documentos

Ya disponemos del formulario para crear nuevos documentos. Ahora hay que construir la lógica para presentarlo al usuario a través de su navegador y para procesar los datos que este devuelva. No es más que volver a repetir el flujo de operaciones del ejemplo desarrollado en el apartado 2.3 adaptándolo a las necesidades marcadas por los requisitos de nuestra aplicación. La acción que se encargará de controlar dicho flujo la denominaremos *executeNuevo()*, y el código correspondiente es el que sigue:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
public function executeNuevo($request)
{
    $this -> formulario = new FormularioDocumentos();
    $this -> formulario -> setDefault('id_usuario', $this -> getUser() -> getAttribute('id_usuario'));

    if($request -> isMethod('post'))
    {
        $this -> formulario -> bind($request -> getParameter('documentos'), $request -> getFiles('documentos'));
        if($this -> formulario -> isValid())
        {

            $this -> procesaNuevoDocumento($this -> formulario);
            $this -> getUser() -> setFlash('mensaje', 'el documento ha sido creado');
            $this -> redirect('gesdoc/index');
        }
    }
}
```

Y este es el código de la plantilla correspondiente *nuevoSuccess.php*:

Contenido del archivo:
apps/frontend/modules/gesdoc/templates/nuevoDocumentoSuccess.php

```
<div id="sf_admin_container">
    <h1>Formulario</h1>

    <div id="sf_admin_header">
    </div>

    <div id="sf_admin_content">
        <ul class="sf_admin_actions">
            <li class="sf_admin_action_list"><a href="php echo url_for('gesdoc/index') ?&gt;"&gt;volver al listado&lt;/a&gt; &lt;/li&gt;
        &lt;/ul&gt;
        &lt;div class="sf_admin_form"&gt;
            &lt;form name="form" action="<?php echo url_for('gesdoc/nuevo') ?&gt;" method="post" enctype="multipart/form-data"&gt;
                &lt;?php echo $formulario -&gt; renderHiddenFields() ?&gt;
                &lt;?php echo $formulario -&gt; renderGlobalErrors() ?&gt;
                &lt;fieldset id="fieldset_1"&gt;
                    &lt;h2&gt;Nuevo Documento&lt;/h2&gt;

                    &lt;div class="sf_admin_form_row"&gt;
                        &lt;?php echo $formulario['titulo'] -&gt; renderError() ?&gt;
                        &lt;div&gt;
                            &lt;?php echo $formulario['titulo'] -&gt; renderLabel() ?&gt;
                            &lt;?php echo $formulario['titulo']-&gt;render() ?&gt;
</pre  


```
 </div>
 </div>

 <div class="sf_admin_form_row">
 <?php echo $formulario['descripcion'] -> renderError() ?>
 <div>
 <?php echo $formulario['descripcion'] -> renderLabel() ?>
 <?php echo $formulario['descripcion']->render() ?>

```


```

Implementación de la creación de nuevos documentos

```
</div>
<div class="sf_admin_form_row">
    <?php echo $formulario['publico'] -> renderError() ?>
    <div>
        <?php echo $formulario['publico'] -> renderLabel() ?>
        <?php echo $formulario['publico']->render() ?>
    </div>
</div>
<div class="sf_admin_form_row">
    <?php echo $formulario['id_tipo'] -> renderError() ?>
    <div>
        <?php echo $formulario['id_tipo'] -> renderLabel() ?>
        <?php echo $formulario['id_tipo']->render() ?>
    </div>
</div>
<div class="sf_admin_form_row">
    <?php echo $formulario['documento_categoria_list'] -> renderError() ?>
    <div>
        <?php echo $formulario['documento_categoria_list'] -> renderLabel() ?>
        <?php echo $formulario['documento_categoria_list']->render() ?>
    </div>
</div>
<div class="sf_admin_form_row">
    <?php echo $formulario['fichero'] -> renderError() ?>
    <div>
        <?php echo $formulario['fichero'] -> renderLabel('fichero (versión 1)') ?>
        <?php echo $formulario['fichero']->render() ?>
    </div>
</fieldset>
<input type="submit" />
</form>
</div>
<div id="sf_admin_footer">
</div>
</div>
```

Como puedes comprobar el código es esencialmente el mismo que el del ejemplo del apartado 2.3, solo que hemos utilizado el formulario recién definido para la creación y edición de documentos. Vamos a explicar los elementos novedosos. En primer lugar utilizamos el método `setDefault()` del formulario para definir el valor del campo `id_usuario`, ya que este debe coincidir con el `id` del usuario que está utilizando la aplicación y que tenemos disponible en la sesión de usuario. Este método acepta como primer argumento el nombre del campo que se desea definir y como segundo el valor por defecto que se le asignará. Recuerda que este campo se ha definido como oculto y, aunque estará disponible en el formulario `HTML`, no será visible al usuario.

Por otro lado, en la plantilla, hemos declarado el elemento `HTML form` como `multipart/form-data`, pues además de datos queremos enviar un fichero. Los ficheros que se envían en la petición `HTTP`, se asocian al formulario mediante el segundo argumento del método `bind()`:

```
<?php
$this -> formulario -> bind($request -> getParameter('documentos'), $request -> getFiles('documentos'));
```

Es decir, el primer argumento del método `bind()` representa los datos que vienen en la petición, y el segundo los archivos que llegan vía dicha petición. La razón de esto radica en el funcionamiento de la operación `POST` del protocolo `HTTP`. Esta operación `HTTP` permite enviar simultáneamente al servidor ficheros y datos pero, por decirlo de alguna forma no muy rigurosa, cada uno “van por su parte”, y en la composición de la petición `POST` hay que indicarlo explícitamente mediante el parámetro `enctype="multipart/form-data"`.

A continuación, si los datos que hemos recibido son válidos (el método `isValid()` del formulario es quien lo decide utilizando los validadores que hemos declarado en la definición del formulario), se lleva a cabo el procesamiento de los datos. Con el fin de construir un código más limpio y legible, hemos llevado dicho procesamiento a un método protegido

Implementación de la creación de nuevos documentos

(*protected*) de la misma clase *gesdocActions* denominado *procesaNuevoDocumento()*. Se ha declarado protegido ya que sólo se accederá a él desde dentro de la propia clase *gesdocActions*. El código de dicho método es el siguiente:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
protected function procesaNuevoDocumento($formulario)
{
    // Antes de nada colocamos en su sitio el fichero (si se ha subido alguno)
    if($formulario -> getValue('fichero') instanceof sfvalidatedfile)
    {
        $fichero = $formulario -> getValue('fichero');

        $tipo = TiposPeer::retrieveByPK($formulario -> getValue('id_tipo'));

        if(!in_array($fichero -> getType(), $tipo -> dameTiposMimePermitidos()))
        {
            $this -> getUser() -> setFlash('error', 'el fichero de tipo '.$fichero -> getType(). ' no se corresponde con el tipo de fichero seleccionado '.$tipo -> getTipoMime());
            $this -> redirect('gesdoc/index');
        }

        $nombreFichero = time() . $fichero -> generateFileName();
        $ruta = sfConfig::get('sf_upload_dir').'/usuario_'.$this -> getUser() -> getAttribute('id_usuario').'/'.$nombreFichero;
        $fichero -> save($ruta);

        if(!$fichero -> isSaved())
        {
            $this -> getUser() -> setFlash('mensaje', 'el fichero no ha podido guardarse');
            $this -> redirect('gesdoc/index');
        }
    }

    // Realizamos una transacción para evitar que se guarden los datos si el
    // proceso, por algún motivo, se rompe
    $con = Propel::getConnection();
    $con -> beginTransaction();

    try
    {
        // Creamos el documento
        $documento = new Documentos();
        $documento -> setTitulo($formulario -> getValue('titulo'));
        $documento -> setDescripcion($formulario -> getValue('descripcion'));
        $documento -> setPublico($formulario -> getValue('publico'));
        $documento -> setIdTipo($formulario -> getValue('id_tipo'));
        $documento -> setIdUsuario($formulario -> getValue('id_usuario'));
        $documento -> save();

        //Asociamos las categorías
        foreach ($formulario -> getValue('documento_categoria_list') as $categoria)
        {
            $documento_categoria = new DocumentoCategoria();
            $documento_categoria -> setIdDocumento($documento -> getIdDocumento());
            $documento_categoria -> setIdCategoria($categoria);
            $documento_categoria -> save();
        }

        // Creamos la versión 1 (si se ha subido un fichero)
        if($formulario -> getValue('fichero') instanceof sfvalidatedfile)
        {
            $version = new Versiones();
            $version -> setNumero(1);
            $version -> setNombreFichero($nombreFichero);
            $version -> setDescripcion('Versión 1 del documento');
            $version -> setFechaSubida(date('Y-m-d H:i:s'));
            $version -> setIdDocumento($documento -> getIdDocumento());
            $version -> save();
        }
    }

    $con ->commit();
} catch (Exception $e)
{
    $con -> rollBack();
    throw $e;
}
}
```

El flujo troncal de este procedimiento es el siguiente:

1. Si se ha subido un fichero se comprueba que el tipo *mime* del mismo coincide con uno de los tipos *mimes* que se han asociado al tipo de fichero en la tabla *tipos*. En caso afirmativo se asigna un nombre **único** al fichero que consiste en una concatenación del tiempo actual medido en segundos desde el 1 de junio de 1970 a las 00:00:00 (lo que se llama época Unix) con un nombre generado aleatoriamente con el método *generateFileName()* del objeto *sfvalidatedfile()*. Entonces se guarda el archivo con ese nombre en la carpeta del usuario que lo ha enviado a través del formulario. Si alguno de estos subprocessos no ha ido bien, se aborta el procedimiento y se realiza una redirección a la acción *gesdoc/index*, enviando un mensaje de error a través de la sesión de usuario.
2. Utilizando un objeto de *Propel* del tipo *Documentos*, se crea un nuevo registro en la tabla *documentos* con los datos enviados.
3. Se crean tantos registros como categorías se hayan seleccionado en la tabla *categorias* asociados al documento recién creado. Para ello se utilizan objetos *DocumentoCategoria* de *Propel*.

Implementación de la modificación de documentos existentes

4. Si se subió un fichero, que corresponde a la primera versión del documento, se crea un nuevo registro en la tabla *versiones* asociado al documento recién creado. El nombre del campo *nombre_fichero* es el nombre calculado en el punto uno. De nuevo utilizamos un objeto de *Propel*, en este caso *Visiones*, para realizar la creación del registro.

Las operaciones sobre la base de datos (pasos del 2 al 4) se realizan dentro de una transacción, para garantizar que si algo va mal no se realice ninguna inserción de datos que crearían datos huérfanos no deseados en la base de datos. Dicha transacción se ha encajado dentro de una estructura *try-catch*.

Para la comprobación de los tipos *mime* permitidos hemos ampliado la clase de *Propel Tipos* con un método denominado *dameTiposMimePermitidos()* que devuelve un *array* con los valores separados por coma del campo *tipo_mime* de la tabla *tipos*. Este es el código:

Trozo del archivo: *lib/model/Tipos.php*

```
<?php  
...  
public function dameTiposMimePermitidos()  
{  
    $tipos_mime = explode(',', $this -> getTipoMimeType());  
    return $tipos_mime;  
}
```

Sería mucho más elegante construir un procedimiento que asociase a través del tipo *mime* del archivo que se ha subido al servidor su tipo de archivo (*openoffice*, *pdf*, etc), sin necesidad de que el usuario indicase lo indicase. Sin embargo, hemos optado por este procedimiento “manual” con el fin de no oscurecer los conceptos básicos que queremos mostrar enredándonos en la implementación de farragosos procedimientos.

Por último hemos hecho uso del operador *instanceof* para comprobar si el formulario validado contenía un archivo válido (*sfValidatedFile*). Este operador es muy útil para conocer el tipo (la clase) de los objetos que manipulamos en nuestros programas *PHP*.

Si todo ha salido bien habremos creado un nuevo documento con un número determinado de categorías asociadas, la primera versión del documento y un archivo físico correspondiente a dicha versión almacenado en la carpeta del usuario que utiliza la aplicación. O dicho en otros términos más concretos: Un nuevo registro en la tabla *documentos*, varios registros en la tabla *documento_categoria* asociados al anterior registro, un nuevo registro en la tabla *versiones* asociado a ese mismo registro, y un archivo físico en la carpeta del usuario que está manejando la aplicación.

Implementación de la modificación de documentos existentes

La modificación de los metadatos de cada documento se realiza a través del *link modificar* que aparece en la columna de acciones del listado de documentos. Dicho *link* aparece únicamente en los documentos que pertenecen al autor que está utilizando la aplicación. Ahora falta implementar la acción que lleva a cabo la operación de modificación. Y esto es lo que faremos en este apartado.

Comenzaremos por definir el formulario que utilizaremos para la modificación de los metadatos de la aplicación. Para ello, en primer lugar, debemos analizar en qué consiste la operación de modificación. Nos remitimos para ello a las especificaciones de la aplicación. Allí se especifica que los ficheros de las versiones subidas al repositorio deben ser del tipo definido en el documento al que pertenecen. Además, una vez subidas al repositorio no se podrán eliminar. De estos dos requisitos podemos inferir como solución más sencilla, que en la edición de los metadatos de un documento no podemos cambiar el campo *id_tipo*. Además la modificación de los metadatos no incluye la posibilidad de subir un fichero como

Implementación de la modificación de documentos existentes

en el caso de la creación de un documento. Por estas razones no podemos utilizar directamente el formulario *FormularioDocumento* para implementar la modificación, ya que nos sobra el campo *fichero* y no debemos mostrar el campo *id_tipo*. Parece que la solución pasa por definir otro formulario distinto para la modificación. Y aunque es así, debido a que este nuevo formulario es prácticamente el mismo que el que se utilizó para la creación de documentos, la solución más elegante y aconsejada es utilizar el concepto de herencia tan útil en la *POO*, y crear el nuevo formulario, al que llamaremos *FormularioModificacionDocumentos*, como una clase hija del formulario *FormularioDocumento*, y retocar esta nueva clase modificando los elementos diferentes, es decir, redefiniendo el campo *id_tipo*, que ahora deseamos que sea un campo oculto, y eliminando el campo *fichero* ya que no lo necesitamos para la operación de modificación. El siguiente código implementa el nuevo formulario para la modificación de documentos basado en el que ya existe para la creación de los mismos:

Trozo del archivo: *apps/frontend/lib/FormularioModificacionDocumentos.class.php*

```
<?php

class FormularioModificacionDocumentos extends FormularioDocumentos
{
    public function configure()
    {
        parent::configure();

        $this -> widgetSchema['id_tipo'] = new sfWidgetFormInputHidden();

        unset($this -> widgetSchema['fichero']);
        unset($this -> validatorSchema['fichero']);
    }
}
```

El resto de los *widgets* los podemos reutilizar, así como los validadores.

Una vez definido el formulario vamos a construir la acción *executeModificar()*, la cual se encargará de enviar el formulario al cliente para que este lo rellene.

Trozo del archivo: *apps/frontend/modules/gesdoc/action/action.class.php*

```
<?php
...
public function executeModificar(sfWebRequest $request)
{
    $this -> forward404Unless($request -> hasParameter('id_documento'));

    $documento = DocumentosPeer::retrieveByPK($request -> getParameter('id_documento'));

    $this -> forward404Unless($documento instanceof Documentos);

    $categorias = array();
    foreach ($documento -> getDocumentoCategorias() as $categoria)
    {
        $categorias[] = $categoria -> getIdCategoria();
    }

    $this -> formulario = new FormularioModificacionDocumentos();

    $datos = array(
        'id_documento' => $documento -> getIdDocumento(),
    )
}
```

Implementación de la modificación de documentos existentes

```
'titulo'      => $documento -> getTitulo(),
'descripcion' => $documento -> getDescripcion(),
'documento_categoria_list' => $categorias,
'publico'      => $documento -> getPublico(),
'id_usuario'   => $documento -> getIdUsuario(),
'id_tipo'      => $documento -> getIdTipo(),
'_csrf_token'  => $this -> formulario -> getCSRFToken(sfConfig::get('sf.csrf.secret'))
);

$this -> formulario -> bind($datos, array());
}
```

La acción que acabamos de mostrar declara un objeto formulario del tipo *FormularioModificacionDocumentos* y le asocia (método *bind()*) los datos del documento que se desea modificar. Por ello hemos recuperado previamente el documento cuyo id nos viene en la petición (*request*). Ahora construimos la plantilla asociada *modificarSuccess.php*:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/modificarSuccess.php*

```
<div id="sf_admin_container">
    <h1>Modificación del Documento</h1>
    <div id="sf_admin_header">
    </div>
    <div id="sf_admin_content">
        <ul class="sf_admin_actions">
            <li class="sf_admin_action_list"><a href="php echo url_for('gesdoc/index') ?&gt;"&gt;volver al listado&lt;/a&gt; &lt;/li&gt;
        &lt;/ul&gt;
        &lt;div class="sf_admin_form"&gt;
            &lt;form name="form" action="<?php echo url_for('gesdoc/guardarModificacion?id_documento='.$id_documento) ?&gt;" method="post"&gt;
                &lt;?php echo $formulario -&gt; renderHiddenFields() ?&gt;
                &lt;?php echo $formulario -&gt; renderGlobalErrors() ?&gt;
                &lt;fieldset id="fieldset_1"&gt;
                    &lt;h2&gt;Modificar Metadatos Documento&lt;/h2&gt;
                    &lt;div class="sf_admin_form_row"&gt;
                        &lt;?php echo $formulario['titulo'] -&gt; renderError() ?&gt;
                        &lt;div&gt;
                            &lt;?php echo $formulario['titulo'] -&gt; renderLabel() ?&gt;
                            &lt;?php echo $formulario['titulo']-&gt;render() ?&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;
                    &lt;div class="sf_admin_form_row"&gt;
                        &lt;?php echo $formulario['descripcion'] -&gt; renderError() ?&gt;
                        &lt;div&gt;
                            &lt;?php echo $formulario['descripcion'] -&gt; renderLabel() ?&gt;
                            &lt;?php echo $formulario['descripcion']-&gt;render() ?&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;
                    &lt;div class="sf_admin_form_row"&gt;
                        &lt;?php echo $formulario['publico'] -&gt; renderError() ?&gt;
                        &lt;div&gt;
                            &lt;?php echo $formulario['publico'] -&gt; renderLabel() ?&gt;
                            &lt;?php echo $formulario['publico']-&gt;render() ?&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;
                    &lt;div class="sf_admin_form_row"&gt;
                        &lt;?php echo $formulario['documento_categoria_list'] -&gt; renderError() ?&gt;
                        &lt;div&gt;
                            &lt;?php echo $formulario['documento_categoria_list'] -&gt; renderLabel() ?&gt;
                            &lt;?php echo $formulario['documento_categoria_list']-&gt;render() ?&gt;
                        &lt;/div&gt;
                    &lt;/div&gt;
                &lt;/fieldset&gt;
                &lt;input type="submit" /&gt;
            &lt;/form&gt;
        &lt;/div&gt;
        &lt;div id="sf_admin_footer"&gt;
        &lt;/div&gt;
    &lt;/div&gt;</pre
```

La cual es muy parecida a la plantilla correspondiente a la creación de un documento. Ahora ya puedes probar el funcionamiento de la acción en tu navegador picando en el *link modificar* de alguno de los documentos. Observa que la acción que dispara el formulario es *gesdoc/guardarModificacion*, la cual debemos implementar aún. Pero antes de hacer esto vamos rehacer la acción que acabamos de implementar sustituyendo el formulario *FormularioModificacionDocumentos* por otro más adecuado que, además, fue creado automáticamente por *symfony* cuando al principio del proyecto invocamos la tarea:

Implementación de la modificación de documentos existentes

```
# symfony propel:build-forms
```

Esta tarea construye en el directorio *lib/forms* un formulario por cada objeto de nuestro modelo de *Propel*, es decir, por cada tabla de nuestra base de datos. Si echas un vistazo a este directorio podrás ver un conjunto de clases cuyo nombre sigue el patrón *{ObjetoPropel}Form.php*, entre las que se encuentra *DocumentosForm.php*, asociada al objeto *Documentos*. Al igual que ocurría con los objetos de *Propel*, las clases formularios, en un principio, no contienen ninguna acción, simplemente derivan de las clases bases *Base{ObjetoPropel}Form.php* ubicadas en el directorio *lib/form/base*, que son las que contienen el código construido automáticamente a partir de la definición de los objetos del modelo de *Propel*. La idea que subyace bajo esta forma de organización del código es la misma que ya explicamos en la unidad 5 sobre los objetos de *Propel* y que resumimos a continuación: cada vez que modificamos el modelo de datos por cualquier razón (se añade, modifica o elimina algún campo a alguna tabla, se crean nuevas tablas, etcétera) debemos regenerar los formularios con la tarea *propel:build-form*, para que dichos formularios incorporen los cambios realizados. Pues bien esta regeneración únicamente tiene lugar en las clases bases del directorio *lib/form/base* de manera que no perdamos los cambios o adiciones que hayamos realizado en las clases hijas correspondientes.

Si miras el código de cualquier formulario base podrás comprobar que son clases que no derivan de *sfForm*, como los formulario que hasta el momento hemos estudiado en esta unidad, sino que derivan de *BaseFormPropel* que es una clase derivada de *sfForm* pero que incluye ciertas funcionalidades asociadas a los objetos de *Propel*. De hecho cada formulario está asociado a una tabla de la base de datos y sus *widgets* y validadores hacen referencia a los campos de cada una de las tablas. Gracias a esto, los formularios de *Propel* implementan una nueva operación, denominada *save()*, gracias a la cual el propio formulario se encarga de grabar los campos del mismo en la tabla que le corresponde, aliviando al programador de realizar dicha tarea y arrojando un código más sencillo y legible.

Vamos a ver como podemos utilizar dichos formularios para llevar a cabo la operación de modificación de los documentos. En primer lugar miramos el código del objeto *BaseDocumentosForm* y comprobamos que casi coincide con nuestros requisitos. Los cambios que debemos hacer para nuestros propósitos son:

- Los campos *id_usuario* y *id_tipo* deben ser ocultos,
- el campo descripción encajaría mejor si fuese un *TextArea*,
- el campo *publico* debe ser un *widget* de selección con dos opciones: 'no' y 'si',
- y el validador del campo *publico* debe garantizar que únicamente se acepten como valores 0 (correspondiente a 'no') y 1 (correspondiente a 'si')

Todo lo cual podemos realizar modificando adecuadamente la clase hija *DocumentosForm* de la siguiente manera:

Contenido del archivo: *lib/form/DocumentosForm.php*

```
<?php  
  
class DocumentosForm extends BaseDocumentosForm  
{  
    public function configure()  
    {  
        $this -> widgetSchema['publico'] = new sfWidgetFormChoice(array('choices' => array('no','si'), 'expanded' => false, 'multiple' => false));  
        $this -> widgetSchema['id_usuario'] = new sfWidgetFormInputHidden();  
        $this -> widgetSchema['id_tipo'] = new sfWidgetFormInputHidden();  
        $this -> widgetSchema['descripcion'] = new sfWidgetFormTextarea();  
  
        $this -> validatorSchema['publico'] = new sfValidatorInteger(array('min' => 0, 'max' => 1));  
    }  
}
```

Ahora rehacemos la acción *executeModificar()* para que utilice este formulario:

Implementación de la modificación de documentos existentes

Trozo del archivo: *apps/frontend/modules/gesdoc/action/action.class.php*

```
<?php
...
public function executeModificar(sfWebRequest $request)
{
    $this -> forward404Unless($request -> hasParameter('id_documento'));

    $documento = DocumentosPeer::retrieveByPK($request -> getParameter('id_documento'));

    $this -> forward404Unless($documento instanceof Documentos);

    $this -> formulario = new DocumentosForm($documento);
    $this -> id_documento = $request -> getParameter('id_documento');
}
```

Como puedes comprobar hemos conseguido un código bastante más escueto y sencillo. Igual que antes, lo primero que hacemos es recuperar el objeto que deseamos modificar a través del *id* que viene en la petición. Pero una vez que lo tenemos no tenemos que preocuparnos de extraer sus campos y asociarlos al formulario, directamente pasamos como argumento en la creación del nuevo formulario de *Propel* el objeto documento que deseamos modificar y él ya se encarga de realizar la asociación. Prueba de nuevo la acción *gesdoc/modificar* y comprueba que funciona exactamente igual que antes. Observa, además, que no hemos tenido que hacer ningún cambio en la plantilla *modificarSuccess.php* ya que está “preparada” para manipular un objeto formulario y la interfaz del nuevo formulario coincide con la del antiguo, es decir, tiene los mismos campos y ofrece los mismos métodos que requiere la plantilla.

Ahora vamos a implementar la acción que se ocupará de grabar los datos devueltos al servidor a través del formulario *HTML*. Hemos llamado a esta acción *gesdoc/guardarModificacion*, así pues la función asociada se denominará *executeGuardarModificacion()*:

Trozo de código del archivo: *apps/frontend/modules/gesdoc/actions/actions.class.php*

```
<?php
...
public function executeGuardarModificacion(sfWebRequest $request)
{
    $this -> forward404Unless($request -> hasParameter('documentos'));

    $documento = DocumentosPeer::retrieveByPK($request -> getParameter('id_documento'));

    $this -> formulario = new DocumentosForm($documento);
    $this -> formulario -> bind($request -> getParameter('documentos'));

    if($this -> formulario -> isValid())
    {
        $this -> formulario -> save();

        $this -> getUser() -> setFlash('mensaje', 'Los metadatos del documento ha sido modificado');

        $this -> redirect('gesdoc/index');
    }

    $this -> setTemplate('modificar');
}
```

En esta acción recuperamos el documento correspondiente al *id* que viene por la petición *HTTP* y creamos un formulario *DocumentosForm* asociado a este documento. Por lo pronto tenemos lo mismo que en la acción anterior. A continuación asociamos el formulario recién creado a los datos que vienen por la petición *HTTP* y si pasa el test de validez procedemos a guardarlo en la base de datos utilizando el método *save()* del formulario. Fíjate; es el formulario el que se encarga de realizar la operación sobre la base de datos, por que al ser

Subida de versiones

un formulario de *Propel* “sabe” como debe realizar dicha operación. Si hubiésemos utilizado el formulario *FormularioModificacionDocumentos*, tendríamos que realizar dicha operación nosotros, es decir, tendríamos que modificar manualmente cada uno de los campos del objeto *Documentos* con los valores de los campos del formulario validado y salvar los cambios con el método *save()* del objeto *Documentos*. Por último, como ya hemos visto en esta misma unidad, si todo va bien se genera una notificación para advertir el éxito de la operación y se redirige la acción hacia el listado de documentos, y si la validación no es correcta se vuelve a enviar el formulario (*setTemplate('modificar')*) con los mensajes de error que han provocado el rechazo de la validación.

Subida de versiones

Para finalizar la aplicación *frontend* de nuestro proyecto tan sólo nos queda desarrollar la subida de nuevas versiones. Cuando un usuario desee subir al servidor una nueva versión de algún documento tendrá que facilitar a la aplicación el fichero asociado a la versión, que debe ser del tipo especificado en el documento correspondiente, y una descripción de la misma. El resto de los campos de la tabla *versiones*: *nombre_fichero*, *numero*, *fecha_subida* y *id_documento*, deben ser calculados automáticamente por la aplicación.

Al igual que en el apartado anterior, comenzaremos por definir un formulario apropiado para llevar a cabo la operación de subida de ficheros para, posteriormente, implementar dicha operación.

El formulario *VisionesForm*

Como acabamos de indicar, para subir nuevas versiones, el autor necesita introducir los siguientes datos:

- la descripción de la nueva versión y
- el fichero que corresponde a la nueva versión

El primero será modelado como un *sfWidgetFormInput* y el segundo como un *sfWidgetFormInputFile*.

El resto de los datos; el nombre del fichero, el número de versión, la fecha de la subida y el *id* del documento asociado deben ser asignados automáticamente por la aplicación.

A continuación presentamos el código correspondiente al formulario para la subida de versiones:

Contenido del archivo: *app/frontend/lib/FormularioVersiones.class.php*

```
<?php

class FormularioVersiones extends sfForm
{
    public function configure()
    {
        $this->setWidgets(array(
            'descripcion'      => new sfWidgetFormTextarea(),
            'fichero'          => new sfWidgetFormInputFile()
        ));

        $this->setValidators(array(
            'descripcion'      => new sfValidatorString(array('max_length' => 255, 'required' => true)),
            'fichero'          => new sfValidatorFile(array('required' => 'true'))
        ));

        $this->widgetSchema->setNameFormat('versiones[%s]');
    }
}
```

Implementación de la subida de nuevas versiones

Subida de versiones

Construimos ahora la acción que enviará al cliente el formulario *HTML* para la subida de nuevas versiones. Denominaremos a tal acción *executeSubirVersion()*:

Trozo del archivo: *apps/frontend/modules/gesdoc/actions.class.php*

```
<?php
...
public function executeSubirVersion(sfWebRequest $request)
{
    $this -> forward404Unless($request -> hasParameter('id_documento'));
    $this -> formulario = new FormularioVersiones();

    $documento = DocumentosPeer::retrieveByPK($request -> getParameter('id_documento'));
    $this -> tipoDocumento = $documento -> getTipos() -> getNombre();
    $this -> id_documento = $request -> getParameter('id_documento');

    if($request -> isMethod('post'))
    {
        $this -> formulario -> bind($request -> getParameter('versiones'), $request -> getFiles('versiones'));

        if($this -> formulario -> isValid())
        {
            if($this -> formulario -> getValue('fichero') instanceof sfValidatedFile)
            {
                $fichero = $this -> formulario -> getValue('fichero');

                $nombreFichero = time() . $fichero -> generateFileName();
                $ruta = sfConfig::get('sf_upload_dir').'/usuario_'.$this -> getUser() -> getAttribute('id_usuario').'/'. $nombreFichero;
                $fichero -> save($ruta);

                if(!$fichero -> isSaved())
                {
                    $this -> getUser() -> setFlash('mensaje', 'el fichero no ha podido guardarse');
                    $this -> redirect('gesdoc/index');
                }
            }

            $version = new Versiones();
            $version -> setNombreFichero($nombreFichero);
            $version -> setNumero(VersionesPeer::calculaVersion($this -> id_documento));
            $version -> setFechaSubida(date('Y-m-d H:i:s'));
            $version -> setIdDocumento($this -> id_documento);
            $version -> setDescripcion($this -> formulario -> getValue('descripcion'));
            $version -> save();

            $this -> getUser() -> setFlash('mensaje', 'Nueva versión subida al repositorio');
            $this -> redirect('gesdoc/index');
        }
    }
}
```

Esta acción es parecida a la correspondiente a la creación de un nuevo documento. Vamos a comentarla muy brevemente, dejando una interpretación en profundidad al estudiante.

En primer lugar definimos un formulario del tipo *FormularioVersiones* para mostrarlo en la vista. Además obtenemos el *id* del documento pues nos hará falta en la plantilla para construir el parámetro *action* del formulario. También recuperamos el tipo del documento para colocar un mensaje en el formulario que indique al usuario que el archivo a subir debe ser de ese tipo. Si la petición no proviene del envío del formulario, es decir, si no es de tipo *POST*, simplemente se envía un formulario vacío para ser llenado por el usuario. El código de la vista correspondiente es el que sigue:

Contenido del archivo: *apps/frontend/modules/gesdoc/templates/subirVersionSuccess.php*

```
<div id="sf_admin_container">
    <h1>Nueva Versión</h1>
    <div id="sf_admin_header">
    </div>
    <div id="sf_admin_content">
        <ul class="sf_admin_actions">
            <li class="sf_admin_action_list"><a href="php echo url_for('gesdoc/index') ?&gt;">volver al listado</a> </li>
        </ul>
        <div class="sf_admin_form">
            <form name="form" action="php echo url_for('gesdoc/subirVersion?id_documento='.$id_documento) ?&gt;" method="post" enctype="multipart/form-data"&gt;</pre
```

```
            <?php echo $formulario -> renderHiddenFields() ?>
            <?php echo $formulario -> renderGlobalErrors() ?>
            <fieldset id="fieldset_1">
                <h2>Datos de la versión</h2>
                <div class="sf_admin_form_row">
                    <div>
```

Subida de versiones

```
<label>Tipo de documento</label>
<?php echo $tipoDocumento ?>
</div>
</div>

<div class="sf_admin_form_row">
<?php echo $formulario['descripcion'] -> renderError() ?>
<div>
<?php echo $formulario['descripcion'] -> renderLabel() ?>
<?php echo $formulario['descripcion']->render() ?>
</div>
</div>

<div class="sf_admin_form_row">
<?php echo $formulario['fichero'] -> renderError() ?>
<div>
<?php echo $formulario['fichero'] -> renderLabel() ?>
<?php echo $formulario['fichero']->render() ?>
<br/>
<help> El archivo debe ser un <?php echo $tipoDocumento ?></help>
</div>
</div>
</fieldset>
<input type="submit" />
</form>
</div>
</div>
<div id="sf_admin_footer">
</div>
</div>
```

Si por el contrario se llega a la acción a través del envío del formulario *HTML*, entonces hay que proceder al procesamiento de sus datos. Para ello asociamos el formulario a los valores que llegan en la petición (datos y ficheros, fíjate que el formulario es de tipo *form-data/multipart*) mediante el método *bind()* y procedemos a la validación. Si los datos enviados son válidos entonces comenzamos por guardar el fichero en la carpeta correspondiente al usuario que lo ha subido al servidor y, posteriormente, creamos un nuevo registro de la tabla *versiones* cuyo campo descripción es el que envió el usuario, siendo el resto calculados automáticamente por la aplicación. Observa que para definir el valor del campo *numero* hemos ampliado la clase *VersionesPeer* para incorporar una función que calcule la última versión de un documento.

Contenido del archivo: *apps/lib/model/VersionesPeer.php*

```
<?php

class VersionesPeer extends BaseVersionesPeer
{
    static public function calculaVersion($id_documento)
    {
        echo $id_documento;
        $c = new Criteria();
        $c -> add(VersionesPeer::ID_DOCUMENTO, $id_documento);
        $c -> addDescendingOrderByColumn(VersionesPeer::NUMERO);

        $ultimaVersion = self::doSelectOne($c);

        if($ultimaVersion instanceof Versiones)
        {
            $numero = $ultimaVersion -> getNumero();
            $numero += 1;
        }
        else
        {
            $numero = 1;
```

Conclusión

```
    }

    return $numero;
}

}
```

Y ya puedes probar a subir nuevas versiones. Terminamos con esto la aplicación *frontend* del gestor documental.

Conclusión

Hemos finalizado la primera de las aplicaciones del gestor documental, la que hemos llamado *frontend* que constituye la parte esencial del gestor. La aplicación muestra los documentos disponibles, públicos y privados, en función del perfil del usuario que la esté utilizando. Se pueden realizar búsquedas sobre todos los metadatos de los documentos, se pueden crear nuevos documentos, modificar sus metadatos y subir al repositorio versiones de los documentos.

Hemos estudiado el funcionamiento y la estructura de los formularios de *symfony*, sus *widgets* y sus validadores y hemos mostrado como utilizarlos implementando las funcionalidades de creación de nuevos documentos, la modificación de sus metadatos y la subida de nuevas versiones asociadas a ellos. Es importante indicar aquí, aunque el estudiante ya lo habrá percibido, que para utilizar con soltura los formularios de *symfony* es imprescindible practicar intensiva y extensivamente con ellos. Además, debido a la gran cantidad de tipos de *widgets* y validadores resulta prácticamente imprescindible tener a mano la documentación de la *API* del *framework* de formularios para trabajar con ellos.

También hemos introducido otro tipo de formulario que está asociado con el modelo de *Propel* y, por tanto, con las tablas de la base de datos. Dichos formularios facilitan el intercambio de datos entre los formularios y las tablas de la base de datos y, como veremos en la unidad siguiente, son especialmente adecuados para el desarrollo de aplicaciones de administración de las bases de datos, aunque, como hemos mostrado en esta unidad, también pueden ser muy útiles en otras situaciones. Se trata siempre de estudiar qué objetos podemos reutilizar mediante su extensión y adaptación a nuestras necesidades.

Unidad 9: Desarrollo de la parte de administración.

Llegados a este punto sólo nos restan dos unidades para terminar el curso y aún tenemos que desarrollar la aplicación de administración (*backend*) completamente. Tal aplicación, como se especificó en la unidad 4, constará de tres módulos:

- la gestión de los usuarios,
- la gestión de tipos de archivos y
- la gestión de categorías,

entendiendo por gestión la consulta, creación, modificación y borrado de elementos. Es decir, la aplicación *backend* contendrá tres módulos que realizan las cuatro operaciones básicas que ofrecen los sistemas gestores de base de datos. Por esta razón a este tipo de módulos se les conoce como módulos *CRUD* (*Create - Retrieve - Update* y *Delete*). Además la aplicación debe incorporar un mecanismo de inicio de sesión para garantizar que únicamente los usuarios con perfil de administración puedan acceder a ella y utilizar sus funcionalidades.

A lo largo de esta unidad, aunque te cueste creerlo, construiremos la aplicación de *backend* completa. Y lo haremos dentro de las aproximadamente 25 páginas que vienen ocupando cada unidad. Este aparente milagro; desarrollar tres módulos de administración completos con control de acceso en una sola unidad, será posible gracias a la reutilización de código a través de los *plugins* de *symfony*, y al uso del generador automático de módulos de administración mediante el cual, como indica su nombre, se construyen al vuelo módulos para la gestión de las tablas de las bases de datos incorporadas a nuestro proyecto.

La hoja de ruta que marcará el desarrollo de esta unidad es la siguiente: primero convertiremos el módulo de inicio de sesión (*inises*) de la aplicación *frontend* en un módulo de un *plugin* del proyecto con el objetivo de que podamos utilizarlo también en la aplicación *backend* que nos disponemos a desarrollar. Cuando, después de esta transformación, hayamos comprobado que la aplicación *frontend* sigue funcionando bien, procederemos a la creación de la aplicación *backend* reutilizando el *layout*, las *CSS's* que ya utilizamos en la aplicación *frontend* y el proceso de inicio de sesión que, para entonces, ya forma parte de un *plugin* del proyecto compartido por ambas aplicaciones. Después generaremos todos los módulos *CRUD* utilizando el generador de módulos de administración de *symfony*. Finalmente, realizaremos algunos pequeños ajustes sobre los módulos generados automáticamente para adaptarlos a nuestras necesidades.

Transformación del módulo de inicio de sesión en un *plugin*

Los *plugins* de *symfony*

Los *plugins* de *symfony* ofrecen una importante vía de expansión para los proyectos *symfony*. En lugar de buscar una definición precisa, lo cual sería bastante complicado y no sé hasta qué punto comprensible, vamos a contar qué es lo que se puede hacer con ellos y como se utilizan.

En primer lugar los *plugins*, como los proyectos, pueden contener todo tipo de elementos: modelos, formularios, librerías de clases, módulos, *CSS's*, *javascripts*, etcétera. Qué elementos incluya cada *plugin* es cuestión de la finalidad del mismo. Un *plugin* puede ser tan sencillo como una librería de clases o tan sofisticado como un *CMS* completo. En ambos casos el *plugin* (y el nombre es bastante descriptivo), puede ser fácilmente conectado (enchufado) al proyecto *symfony* extendiendo las funcionalidades de este.

En nuestro caso el *plugin* que desarrollaremos consistirá en un módulo para realizar un inicio de sesión y, como *plugin* que será, podrá ser utilizado en cualquier aplicación de nuestro proyecto o de cualquier otro proyecto *symfony* que desarrollemos más adelante. Es más, si

Unidad 9: Desarrollo de la parte de administración.

pensamos que puede ser útil a otros desarrolladores podemos pensar en compartirlo a través del repositorio público de *plugins* de *symfony*. En este repositorio podemos encontrar cientos de *plugins* muy útiles y estables. Así pues un *plugin* de *symfony* puede ser pensado como una forma de reutilizar y compartir código, de no reinventar la rueda cada vez.

La estructura de directorios de un *plugin* es similar a la de una aplicación. Por ello podemos concebir a los *plugins* una forma de organizar el código por funcionalidad en lugar de por capas.

Los *plugins* deben ubicarse dentro del directorio *plugins* del proyecto, bajo una carpeta que representa el nombre del *plugin* y que debe terminar con el sufijo *Plugin*. El siguiente esquema muestra la estructura de un posible *plugin* con dos módulos, varias librerías y recursos web (*javascripts*, *CSS's* e imágenes)

Estructura de un *plugin* de *symfony*

```
{nombre_plugin}Plugin
|- config
|- lib
|   |- model
|   |- forms
|   |- filter
|   |- task
|   ` - helper
|- modules
|   |- module_1
|   |   |- actions
|   |   |- config
|   |   ` - templates
|   |- module_2
|   |   |- actions
|   |   |- config
|   |   ` - templates
|
` - templates
-
|- web
|   |- images
|   |- css
|   ` - js
```

Como ya hemos dicho un *plugin* no tiene por qué contener todos los directorios mostrados en este ejemplo. O bien puede ser aún más complejo, pero sea como sea siempre responderá a la estructura estándar de un proyecto *symfony*.

La manera de incorporar un *plugin* a un proyecto *symfony* es registrándolo en el archivo *config/ProjectConfiguration.class.php*. De hecho si lo abres verás que ya existe un *plugin* registrado en el momento de crear el proyecto; el *plugin* de *Propel*. Así pues, si quisieramos utilizar un *plugin* llamado, pongamos, *cmsPlugin* a nuestro proyecto, lo colocaríamos dentro del directorio *plugins*, y lo registraríamos en el archivo *config/ProjectConfiguration.class.php* de la siguiente manera:

Contenido del archivo config/ProjectConfiguration.class.php para registrar un plugin llamado cmsPlugin

Nuestro plugin de inicio de sesión

```
<?php

require_once '/Applications/MAMP/bin/php5/lib/php/symfony/autoload/sfCoreAutoload.class.php';
sfCoreAutoload::register();

class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
        $this->enablePlugins('sfPropelPlugin', 'cmsPlugin');
    }
}
```

A partir de este momento podríamos utilizar las librerías y *recursos web* que el *plugin* proporcionase en cualquiera de las aplicaciones del proyecto. Pero si el *plugin* además contiene módulos y queremos utilizarlos en alguna de las aplicaciones del proyecto, debemos indicarlo en el archivo *apps/nombre_aplicacion/config/settings.yml*. Supongamos que *cms* y *adminCms* son dos módulos del *plugin* que estamos utilizando como ejemplo. Entonces, si deseamos utilizar en una aplicación el módulo *adminCms* lo habilitaríamos añadiendo al fichero *settings.yml* de dicha aplicación la siguiente línea:

Línea añadida al archivo: *apps/nombre_aplicacion/config/settings.yml*

```
...
all:
    .settings:
        ...
            enabled_modules: [ default, adminCSS ]
...

```

Nota

El módulo *default* debemos añadirlo si queremos utilizar las acciones comunes que *symfony* proporciona para realizar algunas operaciones como pueden ser: mostrar que se ha producido alguna violación de seguridad o el famoso *error 404* al que se redirige la acción cuando el recurso no existe. Estas acciones puede ser sustituidas por algunas que nosotros hayamos desarrollado para tal fin.

En la guía de referencia de *symfony* puedes encontrar un capítulo dedicado a la descripción de cada uno de las directivas del archivo *settings.yml*:

http://www.symfony-project.org/reference/1_4/en/

Nuestro plugin de inicio de sesión

Ahora vamos a aplicar la teoría expuesta en el apartado anterior para crear nuestro *plugin* de inicio de sesión. Comenzamos por crear, dentro del directorio *plugins* del proyecto, una carpeta para alojar el código del *plugin*. La denominaremos *IniSesPlugin* (recuerda que debe finalizar con el sufijo *Plugin*):

```
# mkdir plugins/IniSesPlugin
```

Nuestro plugin de inicio de sesión

Como ya hemos dicho, el *plugin* de inicio de sesión consistirá en un módulo que será compartido por las dos aplicaciones de nuestro proyecto: *frontend* y *backend*. Así pues, siguiendo la estructura de los *plugins* de *symfony* (que es la misma que la de una aplicación), creamos dentro del directorio *IniSesPlugin* la carpeta *modules*:

```
# mkdir plugins/IniSesPlugin/modules
```

Entonces movemos el módulo *inises* de la aplicación *backend* al directorio que acabamos de crear:

```
# mv apps/frontend/modules/inises plugins/IniSesPlugin/modules
```

Y acabamos de construir el *plugin* de inicio de sesión. Para que esté disponible en la aplicación *frontend* tenemos que registrarlo en el proyecto:

Contenido del archivo *config/ProjectConfiguration.class.php* para registrar el plugin de inicio de sesión

```
<?php

require_once '/Applications/MAMP/bin/php5/lib/php/symfony/autoload/sfCoreAutoload.class.php';
sfCoreAutoload::register();

class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
        $this->enablePlugins('sfPropelPlugin', 'IniSesPlugin');
    }
}
```

Y habilitar el módulo *inises* del *plugin* en la aplicación *frontend*:

Línea añadida al archivo: *apps/frontend/config/settings.yml*

```
...
all:
  .settings:
    ...
      enabled_modules: [ default, inises ]
...
```

Ahora puedes probar a registrarte y desconectarte en la aplicación *frontend* y verás que todo sigue funcionando como si no hubiésemos hecho nada. Y es que, al habilitar el módulo *inises* del *plugin* en la aplicación *frontend*, este se puede considerar como otro módulo más de la misma.

Nota

Los cambios realizados exigen limpiar la cache para que sean efectivos.

Nuestro plugin de inicio de sesión

Bueno, ahora que hemos comprobado que todo sigue funcionando, hay que realizar algunos retoques para que el módulo *inises* pueda ser realmente utilizado por otras aplicaciones. En efecto, observa que la acción *executeSignin()* de dicho módulo realiza una redirección a la acción *executeIndex()* del módulo *gesdoc*:

Código de la acción *executeSignin()* del archivo: *plugins/IniSesPlugin/modules/inises/actions/actions.class.php*

```
<?php
...
public function executeSignIn(sfWebRequest $request)
{
    $this -> form = new LoginForm();
    if ($request->isMethod('post'))
    {
        $datos = $request -> getParameter($this->form -> getName());
        $this->form->bind($datos);
        if ($this->form->isValid())
        {
            $usuario = $this -> compruebaUsuario($datos);
            if($usuario instanceof Usuarios) // Existe el usuario con los datos dados
            {
                $this -> getUser() -> setAuthenticated(true);
                $this -> getUser() -> setAttribute('id_usuario', $usuario -> getIdUsuario());
                $this -> asociaCredenciales($usuario);
                $this -> redirect('gesdoc/index');
            }
            else
            {
                $this -> mensaje = 'Usuario no autorizado';
            }
        }
    }
}
```

Observa la línea resaltada que es donde se hace dicha redirección. Esto tiene sentido cuando es la aplicación *frontend* la que utiliza el inicio de sesión, ya que dispone de un módulo denominado así y es allí donde queremos ir cuando el proceso de registro de un usuario en la aplicación *frontend* tiene éxito. Pero cuando sea otra la aplicación que haga uso del módulo *inises* el proceso fallará, ya que por lo general no existirá un módulo denominado *gesdoc* en la nueva aplicación, y la redirección cuando se registre satisfactoriamente un usuario debe realizarse al módulo y acción que esta nueva aplicación decida. Para arreglar esto, y aunque aún no lo hemos explicado, utilizaremos el sistema de enrutamiento de *symfony*.

En lugar de redirigir a una acción concreta, realizaremos una redirección a una ruta y será la aplicación en cuestión quien defina el módulo y la acción correspondiente a esa ruta. Las rutas se especifican en los argumentos de las funciones que esperan pares módulos/acción (como *redirect()*, *url_for()*, *link_to()*, etcétera) con el nombre que se les haya asignado precedido por el carácter '@'. De esa forma *symfony* reconoce que debe buscar un par módulo/acción en el archivo *apps/nombre_aplicacion/config/routing.yml* bautizado con el nombre indicado tras el carácter '@'.

Así pues sustituimos en la acción *executeSignin()* del módulo *inises* la ruta concreta '*gesdoc/index*' por la ruta parametrizada '*@pagina_inicial*' (este nombre lo hemos decidido nosotros).

Parametrización de la redirección a la acción *@pagina_inicial* en el archivo *plugins/IniSesPlugin/modules/inises/actions/actions.class.php*

```
<?php
...
$this -> redirect('@pagina_inicial');
```

Nuestro plugin de inicio de sesión

```
...
```

Y en el archivo `apps/frontend/config/routing.yml` añadimos la ruta denominada `@pagina_inicial`:

Trozo del archivo `apps/frontend/config/routing.yml` donde se define la ruta `@ pagina_inicial`

```
...
pagina_inicial:
    url:    /inicio
    param: { module: gesdoc, action: index }
...
```

Por la misma razón debemos parametrizar la redirección que se hace en la acción `executeSignout()` del módulo `inises` hacia '`gesdoc/index`'. Llamaremos a esta nueva ruta '`@logout`'. Dicha acción quedaría:

Código de la acción `executeSignOut` del archivo `plugins/IniSesPlugin/modules/inises/actions/actions.class.php`

```
<?php
...
public function executeSignOut(sfRequest $request)
{
    session_destroy();
    $this -> redirect('@logout');
}
```

Y tenemos que añadir dicha ruta a la aplicación `frontend`:

Trozo del archivo `apps/frontend/config/routing.yml` donde se define la ruta `@ logout`

```
...
logout:
    url:    /salir
    param: { module: gesdoc, action: index }
...
```

Como hemos tocado archivos de configuración debemos borrar la caché de `symfony` antes de volver a ejecutar la aplicación.

```
# symfony cc
```

Y ya puedes comprobar que todo sigue funcionando igual que antes. La diferencia es que tenemos parametrizadas las redirecciones del módulo de inicio de sesión de manera que es la aplicación la que decide, mediante la definición de las rutas `@pagina_inicial` y `@logout`, las acciones a las que se debe redirigir el inicio de sesión cuando un usuario se registra adecuadamente o cuando desea desconectarse de la aplicación. Ahora el `plugin` sí es completamente reutilizable. Es importante que observes que aunque sean muchas las líneas que ha ocupado explicar la transformación del módulo de inicio de sesión en un `plugin`, los cambios realizados sobre el código son muy pocos. Resumiendo:

- Se crea el directorio para alojar el `plugin`

Desarrollo de la aplicación de administración (backend).

- Se reubica el módulo de inicio de sesión
- Se parametrizan las redirecciones de este módulo con rutas de *symfony*
- Se definen las rutas en la aplicación.

Cualquier aplicación que utilice este *plugin* para llevar a cabo el inicio de sesión tan solo tendrá que habilitar el módulo *inises* y definir en su archivo *routing.yml* las rutas *@pagina_inicial* y *@logout*.

Desarrollo de la aplicación de administración (backend).

Creación de la aplicación backend

Según lo especificado en la unidad 4, la aplicación de administración debe facilitar la gestión de las categorías, los tipos de ficheros admitidos y los usuarios de la aplicación. Además el acceso a estas funcionalidades debe estar restringido a los usuarios con perfil administrador.

En este apartado construiremos el esqueleto funcional de la aplicación, entendiendo por tal lo siguiente:

- el *layout* y el menú de la aplicación,
- las *CSS's* para la visualización de la aplicación,
- el procedimiento de inicio de sesión para garantizar que únicamente los administradores puedan utilizar la aplicación de administración.

Una vez hecho esto generaremos automáticamente los módulos de administración y la aplicación quedará prácticamente finalizada.

Comenzamos por generar la aplicación *backend*:

```
# symfony generate:app backend
```

Ahora vamos a construir el *layout* y el menú de la misma. En realidad lo que haremos será reutilizar el mismo *layout* de la aplicación *frontend* y modificar su menú:

```
# cp apps/frontend/templates/layout.php apps/backend/templates
# cp apps/frontend/templates/_menu.php apps/backend/templates
```

Y cambiamos el código del menú por el siguiente:

Código del partial: *apps/backend/templates/_menu.php*

```
<?php if($sf_user -> hasCredential('administracion')): ?>
    <?php echo link_to('Usuarios','gesusu/index') ?> |
    <?php echo link_to('Categorías','gescat/index') ?> |
    <?php echo link_to('Tipos de archivos','gestip/index') ?> |
    <?php echo link_to('Ayuda','gesdoc/ayuda') ?> |
<hr/>
<?php endif; ?>
```

Reutilizaremos las mismas *CSS's* que hemos aplicado en la aplicación *frontend*. Esto se indica en el archivo *view.yml* de la aplicación:

Desarrollo de la aplicación de administración (backend).

Contenido del archivo: apps/backend/config/view.yml

```
default:
  http_metas:
    content-type: text/html

  metas:
    title:      Gestor Documental
    description: Un gestor documental construido con symfony para un curso de Mentor
    keywords:   symfony, gestor_documental, mentor
    language:   es
    robots:     index, follow

  stylesheets: [ default_administrador.css, admin.css, menu.css ]
  javascripts: [ ]

  has_layout: true
  layout:      layout
```

Hemos aprovechado para definir los atributos *metas* que se enviarán en las respuestas *HTTP* generadas por la aplicación *backend*.

Ahora vamos a definir la seguridad de la aplicación exigiendo que todas sus acciones sean seguras por defecto (requieran autentificación) y sólo puedan ser utilizadas por aquello usuarios con credencial de administración. Como ya hemos estudiado, todo esto se realiza a través del fichero *security.yml* de la aplicación:

Contenido del archivo: apps/backend/config/security.yml

```
default:
  is_secure: true
  credentials: administracion
```

Y ahora nos queda implementar el proceso de inicio de sesión, para lo cual utilizaremos nuestro recién horneado plugin *IniSesPlugin* que ya tenemos registrado en el proyecto. Únicamente nos queda habilitar el módulo *inises* en la aplicación *backend*:

Línea añadida al archivo: apps/backend/config/settings.yml

```
...
all:
  .settings:
    ...
      enabled_modules: [ default, inises ]
...

```

Y ya lo tenemos disponible, pero recuerda que este módulo requiere que se definan las rutas *@pagina_inicial* y *@logout* para saber donde tiene que redirigir la acción cuando el usuario se registre con éxito y cuando se desconecte. Dichas rutas se añaden al archivo *routing.yml* de la aplicación:

Rutas añadidas al archivo: apps/backend/config/routing.yml

```
...
pagina_inicial:
  url:  /inicio
```

El generador automático de módulos de administración.

```
param: { module: gesusu, action: index }

logout:
    url: /salir
    param: { module: inises, action: signIn }

...
```

Hemos decidido que la acción inicial tras el proceso de *login* sea la acción *index* del módulo de gestión de usuarios *gesusu*, el cual aún tenemos que desarrollar. Y la acción que se ejecute tras la desconexión sea la acción *signIn* del módulo *inises*, el cual presenta el formulario de inicio de sesión al usuario.

Con esto ya tenemos lo que hemos llamado esqueleto de la aplicación *backend* finalizado. Ahora puedes probar la aplicación mediante la siguiente *url*:

```
http://localhost/gestordocumental/web/backend_dev.php/inises/signIn
```

Te mostrará el formulario de inicio de sesión. Puedes probar a introducir un usuario con perfil administrador para probar que todo marcha, el problema es que como aún no hemos desarrollado el módulo *gesusu*, *symfony* te advertirá de ello cuando tenga lugar la redirección.

El generador automático de módulos de administración.

Casi todas las aplicaciones *web* incorporan una sección para la administración de la misma. Qué es lo que se vaya a administrar dependerá obviamente de la aplicación en sí aunque, generalmente, una parte importante de “lo administrable” se corresponde con colecciones de elementos almacenados en una base de datos. Si, por ejemplo, la aplicación admite el registro de usuarios, será necesario una sección para la administración de los mismos. La administración de este tipo de colecciones de elementos suele presentar una interfaz común que consiste en 4 operaciones:

- creación de datos (Create)
- recuperación de datos (Retrieve)
- actualización de datos (Update) y
- borrado de datos (Delete)

Lo que se conoce ampliamente como una interfaz *CRUD*, acrónimo de *Create*, *Retrieve*, *Update* y *Delete*.

Symfony incorpora una poderosa herramienta mediante la cual se generan de forma automática módulos *CRUD* para la administración de los objetos de su modelo. Estos objetos, como ya hemos aprendido a lo largo del curso, presentan este tipo de interfaz a través de los métodos que se muestran en la tabla siguiente:

Operación	Método	Ejemplo
<i>Create</i>	<i>new()</i>	<i>Usuarios -> new()</i>
<i>Retrieve</i>	<i>retrieveByPk()</i> <i>doSelect()</i> <i>doSelectOne()</i>	<i>UsuariosPeer -> retrieveByPK(5)</i>
<i>Update</i>	<i>save()</i>	<i>Usuarios -> save()</i>
<i>Delete</i>	<i>delete()</i>	<i>Usuarios -> delete();</i>

Generación de los módulos de administración

Gracias a esta interfaz uniforme que presentan los objetos de *Propel* es posible la generación automática de módulos *CRUD* para cualquier objeto del modelo.

Por otro lado, los módulos *CRUD*, necesitan presentar al usuario formularios para solicitar los datos que requieren para dar de alta un nuevo objeto, actualizarlo, eliminarlo o recuperarlo. Como a lo mejor habrás deducido, el generador automático de *symfony* utiliza los formularios y filtros que se han generado mediante la tarea *propel:build-forms* y *propel:build-filter*, alguno de los cuales hemos introducido y utilizado en la unidad anterior. Así pues, las bases sobre las que se asientan los módulos de administración generados automáticamente son los objetos, formularios y filtros del *ORM* que utilicemos en nuestro proyecto: *Propel* o *Doctrine*, lo cual proporciona al desarrollador, no solo la posibilidad de reutilización, sino una gran flexibilidad para cambiar el comportamiento de las operaciones *CRUD* mediante la extensión de dichos objetos.

Como veremos en breve, el código que se genera en cada uno de estos módulos es prácticamente inexistente, y se limita por una parte a extender unas clases bases comunes a todos los módulos de administración, y por otra a la generación de un importante fichero de configuración, denominado *generator.yml*, cuya manipulación por parte del desarrollador, posibilitará modificar el comportamiento por defecto del módulo generado. Cuando el *framework* ejecuta uno de estos módulos, construye a partir de dicho fichero de configuración el código real que será ejecutado en el servidor. Dicho código se graba en el directorio *cache* del proyecto la primera vez que se solicita la ejecución del módulo y, a partir de ese momento y hasta que la caché vuelva a ser limpiada mediante la tarea *cache:clear (symfony cc)*, el *framework* hará uso de este código generado “al vuelo”.

En ocasiones el módulo generado por defecto cubre las necesidades de la aplicación, pero la mayor parte de las veces será necesario modificar el fichero *generator.yml* para configurar el módulo como deseamos. La principal dificultad a la que se enfrenta el programador cuando utiliza la generación automática de módulos de administración es conocer la enorme cantidad de elementos que admite este fichero. Si se desea realizar muchos cambios sobre el módulo por defecto es imprescindible conocer en profundidad dichos elementos y su sintaxis.

Además, si la manipulación del fichero de configuración del módulo de administración generado automáticamente no fuese suficiente, también podemos modificar el comportamiento de las acciones y plantillas por defecto mediante su redefinición en el fichero de acciones o plantillas del propio módulo. También debemos tener en cuenta que podemos extender tanto los objetos *Propel* como los formularios para conseguir nuestro propósitos. Con lo cual las posibilidades de adaptación de estos módulos sólo tienen como límite el conocimiento y habilidad del desarrollador.

Hasta el momento todo ha podido sonar un poco abstracto, no obstante es la base teórica mínima necesaria para enfrentarse a la generación automática de módulos de administración de *symfony*. En los siguientes apartados mostraremos cómo utilizar en la práctica esta potente herramienta.

Generación de los módulos de administración

Y ahora la magia. Primero invocamos el abracadabra:

```
# symfony propel:generate-admin backend Usuarios --module=gesusu
# symfony propel:generate-admin backend Tipos --module=gestip
# symfony propel:generate-admin backend Categorias --module=gescat
```

Y entonces ... itachán!, itachán! con todos ustedes ... ¡Los módulos de administración!. Fin de la aplicación. Bueno casi. Aún nos quedan unos cuantos retoques y algunas explicaciones.

Cambios propuestos para adaptar los módulos de administración a nuestras necesidades.

Por lo pronto vamos a probar los módulos que nos han generado las instrucciones anteriores. Como habrás imaginado cada una de las líneas genera un módulo de la aplicación *backend* sobre los objetos *Usuarios*, *Tipos* y *Categorías* respectivamente y cuyos nombres son *gesusu*, *gestip* y *gescat*.

Para probarlos debemos registrarnos en la aplicación *backend* como usuario administrador, ya que hemos especificado en los archivos de seguridad de la misma que todos los módulos requieren, por defecto, autentificación por parte del usuario. Tras el proceso de inicio de sesión la aplicación mostrará la pantalla correspondiente a la acción *index* del módulo *gesusu*, que como podrás ver se corresponde con un listado de usuarios con filtro de búsqueda. Desde el menú podemos acceder también a las acciones *index* de los módulos *gestip* y *gescat* que nos muestran listados con filtros de búsqueda para tipos de archivos y categorías respectivamente. Juega un rato con los tres módulos. Advertirás que prácticamente tenemos la aplicación resuelta.

Como has podido comprobar cada módulo consiste en dos pantallas; la primera es un listado de elementos con un filtro para realizar búsquedas, y la segunda es una pantalla de edición para añadir nuevos elementos o editar los que ya existen. También podemos borrar los elementos uno a uno, o varios de una vez seleccionándolos y eligiendo la opción borrar del desplegable que aparece bajo el listado. Es decir, se pueden realizar las cuatro operaciones básicas de gestión *C-R-U-D*.

Una vez que salgamos de nuestro asombro caeremos en la cuenta de hay unas pocas cosas que no cuadran. Para comenzar la interfaz se presenta en inglés. Así que vamos a realizar un listado de las cosas que nos gustaría modificar. El resto de la unidad explicará como realizar estos cambios. De esta manera ofreceremos de una forma operativa la técnica a seguir para trabajar con los módulos de administración generados automáticamente.

Cambios propuestos para adaptar los módulos de administración a nuestras necesidades.

Vamos a construir una lista de tareas (*TO-DO*) con los cambios que nos gustaría realizar sobre los módulos generados:

Para todos los módulos

- Presentar la interfaz en castellano.
- Los títulos de las pantallas se corresponden con los nombre de los módulos (*Gesusu List*, *New Gesusu*, *Edit Gesusu*), lo cual no es muy presentable, sería preferible títulos más descriptivos como *Listado de usuarios*, *Edición del usuario 'anselmo'*, etcétera.
- Los listados muestran páginas de 20 elementos. Deseamos reducirlos a 10.

Módulo gesusu

• Listado

- Quitar el campo '*password*' del listado, ya que no da ninguna información útil y ocupa un espacio innecesario.
- Quitar el campo '*password*' del filtro, ya que no tiene sentido realizar una búsqueda por un *password* encriptado.

• Edición

- Convertir el campo '*perfil*' en un desplegable con los valores '*lector*', '*autor*' y '*administrador*', para evitar que se introduzcan valores que no reconoce la aplicación.
- Cambiar el proceso de grabado de los datos para que el valor del '*password*' que introduzca el usuario administrador se grabe en la base de datos encriptado con el

Estructura de un módulo de administración generado automáticamente.

algoritmo *MD5*, ya que si no se hace así los usuarios que se den de alta mediante este módulo no podrán registrarse jamás en la aplicación.

Módulo gescat

• Listado

Quitar del filtro el elemento '*Documento categoria list*'. Este campo representa las relaciones de cada categoría con los documentos existentes. En nuestro caso, el número de documentos que se registrarán será muy grande y, por tanto, este campo de búsqueda será prácticamente inmanejable.

• Edición

Quitar el elemento '*Documento categoria list*', ya que es el usuario que sube documentos quien realiza la asociación de categorías.

Módulo tipos

No necesita más cambios que los que se han propuesto comunes a todos los módulos.

Estructura de un módulo de administración generado automáticamente.

Antes de emprender las acciones pertinentes para realizar los cambios que hemos propuesto en el apartado anterior, echaremos un vistazo a la estructura de uno de los módulos de administración generados automáticamente. Pongamos por caso el módulo de gestión de tipos de archivos.

Los módulos de administración generados automáticamente, como todos los módulos de *symfony*, están compuesto por los directorios *actions* y *templates*. Además requieren el directorio de configuraciones (*config*) y el de librerías (*lib*):

Estructura de un módulo de administración generado automáticamente:

```
gestip
|-actions
| ` -actions.class.php
|-config
| ` -generator.yml
|-lib
| |-gesttipGeneratorConfiguration.class.php
| ` -gesttipGeneratorHelper.class.php
` -templates
```

Si abres cada uno de los ficheros que componen el módulo podrás comprobar que, salvo *generator.yml*, están prácticamente vacíos, únicamente contienen declaraciones de clases que derivan de otras clases base. Además no hay ningún archivo de plantilla. La razón de todo esto es que el auténtico código que se ejecuta cuando el módulo es invocado no es exactamente este, sino que sirve como "semilla" para la generación "al vuelo" del código que realmente será ejecutado y que se almacena en el directorio *cache* de la aplicación. La generación de este nuevo código se realiza según lo especificado en el archivo de configuración *generator.yml*, el cual podemos modificar para cambiar el comportamiento por defecto del módulo. Si quieras ver este código generado "al vuelo" no tienes más que ejecutar al menos una vez el módulo en cuestión y navegar por el directorio *cache*. En el caso del módulo *gestip*, puedes ver el código en

cache/backend/dev/modules/autoGestip/

si has utilizado el controlador de desarrollo, o en

cache/backend/prod/modules/autoGestip/

Estructura de un módulo de administración generado automáticamente.

si utilizaste el controlador de producción.

Ahora puedes ver código contundente, puro y duro, con acciones concretas y gran cantidad de elementos parciales en las vistas (*partials*) que implementan de una manera modular las vistas arrojadas por estos módulos de administración. Echa un vistazo a este código, es un buen ejemplo de código bien horneado, elegante y bien refactorizado.

La siguiente tabla muestra el nombre y un pequeño resumen de las acciones que se generan en los módulos de administración:

Acción	Descripción
<code>executeIndex()</code>	Elabora un listado paginado de elementos.
<code>executeFilter()</code>	Ejecuta el filtrado con los datos que les llega desde el formulario de búsqueda de la pantalla donde se listan los elementos
<code>executeNew()</code>	Acción que se ejecuta cuando se solicita la creación de un nuevo elemento. Muestra un formulario de edición vacío.
<code>executeCreate()</code>	Realiza el grabado (creación) del elemento cuyos datos se han introducido en el formulario enviado por la acción anterior.
<code>executeEdit()</code>	Acción que se ejecuta cuando se solicita la edición de un elemento existente.
<code>executeUpdate()</code>	Realiza el grabado (actualización) del elemento cuyos datos se han introducido en el formulario enviado por la acción anterior.
<code>executeDelete()</code>	Realiza el borrado de un elemento
<code>executeBatch()</code>	Realiza la acción por lotes seleccionada en el desplegable de la pantalla donde se muestra el listado de elemento.
<code>executeBatchDelete()</code>	Realiza la acción de borrado por lotes. En realidad, esta acción es llamada por la acción anterior cuando se ha elegido la opción 'Borrar' del desplegable de acciones por lotes.
<code>processForm()</code>	Procesa el formulario de edición o creación de elementos
<code>getFilters()</code>	Devuelve un array con los filtros aplicados que están almacenados en la sesión.
<code>setFilters()</code>	Define los filtros a aplicar en el listado
<code>getPager()</code>	Devuelve el objeto <i>Pager</i> (paginado)
<code>getPage()</code>	Devuelve la página actual del paginado
<code> setPage()</code>	Define la página que se debe mostrar en el paginado
<code>buildCriteria()</code>	Construye el criterio con el listado
<code>addSortCriteria()</code>	Añade un criterio de ordenación para el listado
<code>getSort()</code>	Devuelve la columna por la que se está ordenando actualmente el listado
<code>setSort()</code>	Define la columna por la que debe ordenarse el listado

Todas estas funciones pueden ser redefinidas en el archivo de acciones del módulo `apps/nombre_aplicacion/modules/nombre_modulo/actions/actions.class.php`, el cual, como ya hemos visto, originalmente está vacío. Es decir, si creamos una función en el archivo de funciones que se llame como algunas de las que hemos enumerado en la tabla anterior, *symfony* ejecutará esta nueva acción en lugar de la existente en el archivo de acciones

El fichero de configuración generator.yml

generado al vuelo y ubicado en el directorio *cache*. Así pues, si deseamos cambiar el comportamiento de una de estas acciones podemos copiar el código del fichero de la caché en el archivo de acciones del módulo y modificarlo a nuestro antojo.

La siguiente tabla muestra las plantillas y *partials* que se generan en los módulos de administración:

Como puedes observar las tres plantillas principales están fragmentadas en numerosos *partials* que permiten un control flexible sobre su aspecto final. Si deseamos cambiar algunas de estas plantillas o *partials* no tenemos más que crear un fichero con el nombre de la plantilla o *partial* en el directorio *templates* del módulo (*apps/nombre_aplicacion/modules/nombre_module/templates*) y escribir el código correspondiente. El *framework* utilizará este nuevo fichero en lugar del creado “al vuelo” en el directorio *cache*. Como punto de partida del código de estos ficheros podemos utilizar una copia del original (el del directorio *cache*).

Por tanto las posibilidades de modificación son espectaculares. De todas formas, antes de redefinir una función de las acciones o una plantilla/*partial*, lo más natural es procurar realizar la modificación a través del archivo de configuración *generator.yml*, el cual nos permite un amplio control sobre el comportamiento del módulo.

En definitiva, combinando la capacidad de control que ofrece el fichero *generator.yml*, las posibilidad de redifinir las acciones y plantillas del módulo y la posibilidad de modificar y extender los objetos, los formularios y los filtros de *Propel*, podremos hacer prácticamente cualquier cosa que se nos ocurra a partir de los módulos de administración generados automáticamente. La aplicación *frontend* que hemos desarrollado a lo largo del curso, por ejemplo, se podría haber construido a partir de un módulo de administración generado automáticamente sobre el objeto *Documentos*. Aunque dado las peculiaridades de dicha aplicación se precisaría un conocimiento bastante profundo de esos módulos de administración, conocimiento que únicamente con la práctica y mostrando una actitud de aprendizaje continuo y autónomo se puede lograr.

El fichero de configuración generator.yml

Antes de enfrentarnos a la implementación de las modificaciones propuestas en el apartado 2.4 vamos a realizar una descripción básica del fichero de configuración *generator.yml*, a partir del que se pueden realizar muchísimos cambios sobre el comportamiento por defecto de los módulos generados automáticamente. De hecho es la primera vía que debemos tomar cuando el módulo de administración no satisface completamente nuestras necesidades.

El número de opciones disponibles es enorme y no vamos a tratarlas todas. La guía de referencia de *symfony* es el recurso más completo y organizado donde podemos encontrar todos los elementos admitidos por el fichero de configuración *generator.yml* así como su sintaxis. En este apartado nos limitaremos a describir la estructura de dicho fichero y los elementos más usados.

Comenzaremos por echar un vistazo al contenido del fichero *generator.yml* original, antes de realizar ningún cambio:

Contenido del fichero:
apps/nombre_aplicacion/modules/nombre_modulo/config/generator.yml

```
generator:
    class: sfPropelGenerator
    param:
        model_class:           Tipos
        theme:                 admin
        non_verbose_templates: true
```

El fichero de configuración generator.yml

```
with_show:           false
singular:          Tipos
plural:            Tiposs
route_prefix:      tipos
with propel_route: 1
actions_base_class: sfActions

config:
  actions: ~
  fields: ~
  list: ~
  filter: ~
  form: ~
  edit: ~
  new: ~
```

La modificación de los módulos de administración se lleva a cabo añadiendo elementos a cada una de las 7 subsecciones de que consta la sección *config* del archivo anterior. Cada subsección define una parte del módulo de administración.

Subsección	Descripción
<i>actions</i>	Define las acciones por defecto tanto del listado de elementos como del formulario de edición/creación
<i>fields</i>	Configuración de los campos del objeto (tabla) que se gestiona
<i>list</i>	Configuración del listado de elementos
<i>filter</i>	Configuración del filtro de búsqueda
<i>form</i>	Configuración del formulario de edición/creación
<i>edit</i>	Configuración específica de la pantalla de edición de elementos
<i>new</i>	Configuración específica de la pantalla de creación de elementos

Columnas reales y virtuales

Hay muchas opciones que toman como argumento una lista de campos. Cada campo puede ser el nombre de una columna real o virtual. Una columna real es aquella que existe en la definición de la tabla representada por el modelo en cuestión, mientras que la columna virtual no existe como tal. En ambos casos es necesario que se haya definido en el modelo correspondiente un método *getter* (*get{NombreColumna()}*) con el nombre de la columna que se quiere mostrar. En el caso de que la columna sea real, estos métodos ya están definidos en el modelo. Pongamos a nuestro modelo como ejemplo: para el modelo *Usuarios*, que representa registros de la tabla *usuarios*, ya existen *getters* para las columnas reales *nombre*, *apellidos*, *username*, *password* y *perfil*. Pero además podemos definir columnas virtuales sin más que definir nuevos *getters*. Supongamos que deseamos mostrar las iniciales de del nombre y apellidos del usuario. Podemos definir el *getter* *getIniciales()* de manera que devuelva las *iniciales* que deseamos. Entonces podremos tratar en el archivo de configuración el término *iniciales* (que es una columna virtual) como cualquier otro campo del modelo.

Placeholders

Otras opciones toman como argumentos los denominados *placeholders* que son cadenas de la forma `%%NAME%%`, donde la cadena *NAME* puede ser cualquier cosa que pueda ser convertida a un *getter* existente. Por ejemplo, si estamos trabajando en el módulo de administración del modelo *Usuarios*, el *placeholder* `%%nombre%%` será reemplazado por el

Implementación de las modificaciones propuestas

resultado de ejecutar `$usuario -> getNombre()`. Los *placeholders* son dinámicamente reemplazados en tiempo de ejecución según el objeto asociado con el contexto actual.

Credenciales

Las acciones que se muestran tanto en el listado como en los formularios de edición y creación, pueden ser ocultadas en función de las credenciales del usuario. Esto se hace mediante la opción *credential*. Es importante indicar que esta opción del fichero *generator.yml* únicamente se ocupa de ocultar las acciones, no de evitar su ejecución. Es decir, si introducimos directamente la *url* que provoca la acción que se ha ocultado mediante la opción *credential* del módulo, dicha acción se ejecutará. La manera de evitar esto, como habrás podido deducir, es mediante la creación de un fichero *security.yml* para el módulo con las opciones adecuadas de seguridad.

Herencia de la configuración.

Hemos visto que la configuración del módulo de administración se establece en 7 subsecciones. Pues bien, dicha configuración se realiza en cascada según las siguientes reglas:

- las subsecciones *new* y *edit* derivan de la subsección *form*, que a su vez deriva de *fields*,
- la subsección *list* deriva de *fields* y
- la subsección *filter* deriva de *fields*.

El término herencia tiene el significado que se le da en la *POO*, es decir, los elementos derivados heredan las propiedades de los elementos padres por defecto, y si es necesario se puede cambiar este comportamiento en la definición de ellos mismos.

En la guía de referencia de *symfony* encontrarás todas las opciones que permiten cada una de las subsecciones de configuración. Utilizaremos algunas de ellas para realizar las modificaciones que hemos propuesto en el apartado 2.4.

Implementación de las modificaciones propuestas

Llegó el momento de aplicar toda la teoría anterior para realizar las modificaciones que hemos propuesto en el apartado 2.4. Dejaremos la traducción de la interfaz para la próxima unidad, ya que allí trataremos el tema de la internacionalización de aplicaciones con *symfony*. Para las demás modificaciones ya conocemos las herramientas necesarias.

Ajustes globales

Títulos

Lo primero que haremos es cambiar los títulos de las pantallas por otros más apropiados. Para lo cual utilizamos la opción *title*, que está disponible en las subsecciones *list*, *edit* y *new*.

Uso de la opción *title* al fichero de configuración:
`apps/backend/modules/gesusu/config/generator.yml`

```
...
list:
    title: Listado de Usuarios
...
edit:
    title: Edición del usuario %%nombre%% %%apellidos%%
new:
    title: Nuevo Usuario
```

Ajustes del módulo usuario

Observa el uso de los *placeholders* `%%nombre%%` y `%%apellidos%%` en el título para la pantalla de edición. Cuando se edita un usuario, tenemos disponible un objeto usuario determinado. Cuando se genera “al vuelo” el código del módulo, *symfony* interpreta los *placeholders* anteriores como `$usuario -> getNombre()` y `$usuario -> getApellidos()` respectivamente.

Hacemos lo mismo con los módulos *gestip* y *gesusu*:

Uso de la opción `title` al fichero de configuración:
`apps/backend/modules/gestip/config/generator.yml`

```
...
list:
  title: Listado de tipos de archivos permitidos
...
edit:
  title: Edición del tipo %%nombre%%
new:
  title: Nuevo tipo de archivo permitido
```

Uso de la opción `title` al fichero de configuración:
`apps/backend/modules/gescat/config/generator.yml`

```
...
list:
  title: Listado de categorías
...
edit:
  title: Edición de la categoria %%nombre%%
new:
  title: Nueva categoría
```

De nuevo hemos aplicado *placeholders* para definir el título de la pantalla de edición tanto de tipos de archivos permitidos como de categorías.

Ahora puedes probar a navegar por todas las pantallas de la aplicación y comprobar el resultado de las modificaciones realizadas en los ficheros de configuración.

Paginado

A continuación vamos a definir paginados de 10 elementos en cada uno de los módulos de administración. La opción `max_per_page`, disponible en la subsección `list` ajusta este comportamiento:

Uso de la opción `max_per_page` en los ficheros de configuración:
`apps/backend/modules/ges{usu,tip,cat}/config/generator.yml`

```
...
list:
  ...
  max_per_page: 10
```

Ahora puedes comprobar como los listados se generan con un paginado y con 10 elementos como máximo.

Ajustes del módulo usuario

Eliminación de los campos *id_usuario* y *password* del listado

Vamos a eliminar los campos *id_usuario* y *password*, ya que no ofrecen información relevante, además vamos a incluir una columna virtual para mostrar las iniciales del usuario que, aunque tampoco ofrece ninguna información relevante, si que muestra el uso de columnas virtuales.

La columna virtual se crea sin más que añadir un nuevo *getter* al modelo *Usuarios*:

Getter añadido a la clase *Usuarios* (archivo: *lib/Usuarios.php*)

```
<?php  
...  
public function getIniciales()  
{  
    return substr($this -> getNombre(), 0,1).'.'.substr($this -> getApellidos(), 0,1).'.';  
}  
...
```

Y ahora utilizamos la opción *display*, la cual requiere el listado de campos que serán mostrados. Esta opción está disponible en las subsecciones: *list*, *form*, *filter*, *edit* y *new*. Como queremos actuar sobre el listado definimos la opción *display* sobre la subsección *list*:

Uso de la opción *display* sobre la subsección *list* en el fichero de configuración: *apps/backend/modules/gesusu/config/generator.yml*

```
...  
list:  
...  
display: [ =nombre, apellidos, username, iniciales, perfil ]  
...
```

Fíjate que la columna *iniciales* se utiliza como un campo más, *symfony* se encargará de utilizar para cada campo de la lista un *getter* que coincida con él. Observa además que hemos colocado delante del campo *nombre* el carácter '='. Esto provoca que el listado convierta el campo seleccionado (*nombre*, en este caso) en un *link* que enlaza con la pantalla de edición del registro.

Eliminación del campo *password* del filtro de búsqueda

De nuevo se trata de usar la opción *display*, pero esta vez en la subsección *filter*:

Uso de la opción *display* sobre la subsección *filter* en el fichero de configuración: *apps/backend/modules/gesusu/config/generator.yml*

```
...  
filter:  
display: [ nombre, apellidos, username, perfil ]  
...
```

Y volvemos a probar.

Convertir el campo *perfil* de los formularios de edición y creación en un desplegable con los valores lector, autor y usuarios

En realidad esto no es algo que podamos hacer a través del fichero de configuración. El código que se genera se limita a utilizar el formulario de *Propel* correspondiente, en este caso, al modelo *Usuarios*. Por tanto lo que debemos modificar es dicho formulario, que es el responsable de pintar los distintos *widgets* correspondientes a cada uno de los campos del formulario. Se trata de redefinir tanto el *widget* *perfil*, que en el formulario base se ha

Ajustes del módulo usuario

definido como una caja de texto (*sfWidgetFormInput*), como el validador *perfil*, definido originalmente como un validador de cadenas (*sfValidatorString*), y colocar en su lugar un *widget* de selección y un validador apropiado a dicha selección:

Código del archivo: *lib/form/UsuariosForm.class.php* en el que se redefine el widget *perfil* y el validador correspondiente.

```
<?php

class UsuariosForm extends BaseUsuariosForm
{
    public function configure()
    {
        $this -> widgetSchema['perfil'] = new sfWidgetFormChoice(array(
            'choices' => array('lector' => 'lector', 'autor' => 'autor', 'administrador' => 'administrador'),
            'multiple' => false,
            'expanded' => false
        ));

        $this -> validatorSchema['perfil'] = new sfValidatorChoice(array(
            'choices' => array('lector', 'autor', 'administrador')
        ));
    }
}
```

Ahora puedes comprobar como las pantallas de edición y creación del módulo de gestión de usuarios muestran el campo *perfil* como un desplegable. Es importante que comprendas que la forma en que se muestran los campos del formulario no es responsabilidad del módulo en sí, sino que la responsabilidad es del formulario donde se definen los *widgets* y *validadores*. El módulo no hace más que utilizar el formulario.

Encriptación MD5 del password

Esto que suena complicado es, en realidad, de lo más sencillo de todo lo que estamos haciendo. Lo importante es saber donde hay que realizar el cambio. Y para ello tenemos que reflexionar y deducir cual es el elemento responsable de dicho cambio. Pensamos un poco y llegamos a la conclusión de que el elemento encargado de definir el valor del campo *password* es el método *setPassword()* (*setter*) del objeto *Usuario*. Como cualquier otro *setter* recibe como argumento el valor que deseamos para ese campo y se almacena en la base de datos cuando es invocado el método *save()* del objeto. Lo que haremos, entonces, es redefinir el método *setPassword()* para que en lugar de asignar el valor del argumento, asigne resultado de aplicar el algoritmo *MD5* a dicho valor. Para realizar dicha redefinición partimos del código original de la función *setPassword()* en la clase base *BaseUsuarios*:

Código original de la función *setPassord()* en el archivo *lib/model/om/BaseUsuarios.php*

```
<?php
public function setPassword($v)
{
    if ($v !== null) {
        $v = (string) $v;
    }

    if ($this->password !== $v) {
        $this->password = $v;
        $this->modifiedColumns[] = UsuariosPeer::PASSWORD;
    }

    return $this;
} // setPassword()
```

Ajustes del módulo de gestión de categorías.

Analizando un poco el código podemos concluir que la primera asignación simplemente garantiza que en adelante la función utilice una variable de tipo *string*. Más adelante encontramos la auténtica asignación en la línea siguiente:

Línea donde se realiza la asignación del atributo *password* del objeto *Usuarios* en el archivo: *lib/model/om/BaseUsuarios.php*

```
<?php  
...  
$this->password = $v;  
...
```

Lo que haremos será copiar el código de esta función en la clase hija *Usuario* y modificar la línea anterior por la siguiente:

```
<?php  
...  
$this->password = md5($v);
```

La cual asigna al atributo *password* del objeto *Usuario* la encriptación *MD5* del argumento pasado a la función. Nos quedará lo siguiente:

Código modificado de la función *setPassword()* de la clase *Usuarios* en el archivo *lib/model/Usuarios.php*

```
<?php  
...  
public function setPassword($v)  
{  
    if ($v !== null) {  
        $v = (string) $v;  
    }  
  
    if ($this->password !== $v) {  
        $this->password = md5($v);  
        $this->modifiedColumns[] = UsuariosPeer::PASSWORD;  
    }  
  
    return $this;  
} // setPassword()
```

Y de nuevo a probar. Cuando se envía el formulario de edición o creación de usuarios, el campo *password* se guarda codificado en MD5.

Ajustes del módulo de gestión de categorías.

Queremos eliminar del formulario el campo *documento_categoria_list*. Pues vamos al formulario *CategoriasForm* y lo eliminamos:

Código del formulario de categorías definido en el archivo: *lib/form/CategoriasForm.class.php*

```
<?php  
class CategoriasForm extends BaseCategoriasForm  
{
```

Ajustes del módulo de gestión de tipos de ficheros permitidos

```
public function configure()
{
    unset($this -> widgetSchema['documento_categoria_list']);
    unset($this -> validatorSchema['documento_categoria_list']);
}
```

Y ya está, ya no aparece en las pantallas de edición ni de creación de categorías.

Ahora debemos quitar el campo *documento_categoria_list* del filtro de búsqueda de la pantalla donde se listan las categorías. Una posible solución sería indicarlo a través de la opción *display* de la subsección *filter*:

```
...
filter:
    display: [ nombre ]
...
```

Aunque esto funciona, como puedes comprobar, lo más correcto es eliminar el campo del formulario *CategoriasFormFilter* que define el filtro de búsqueda, de la misma forma que hemos hecho con el formulario *CategoriasForm*.

Código del formulario de búsqueda de categorías definido en el archivo: *lib/filter/CategoriasFormFilter.class.php*

```
<?php
...
class CategoriasFormFilter extends BaseCategoriasFormFilter
{
    public function configure()
    {
        unset($this -> widgetSchema['documento_categoria_list']);
        unset($this -> validatorSchema['documento_categoria_list']);
    }
}
```

Ahora ya no es necesaria la opción *display* de la subsección *filter*.

Ajustes del módulo de gestión de tipos de ficheros permitidos

Aunque en un principio dijimos que, a excepción de los títulos de las pantallas y de la traducción al castellano de la interfaz, no íbamos a realizar modificaciones en este módulo, para hacerlo homogéneo y coherente con los anteriores eliminaremos el campo *id_tipo* del listado de elementos:

Uso de la opción *display* sobre la subsección *filter* para eliminar el campo *id_tipo* en el fichero: *apps/backend/modules/gestip/config/generator.yml*

```
...
filter
    display: [ =nombre, tipo_mime ]
...
```

Conclusión

Ajustes del módulo de gestión de tipos de ficheros permitidos

A lo largo de la unidad hemos desarrollado completamente la aplicación de administración, denominada *backend*, del gestor documental. Ya disponemos, por tanto, de una versión completa del gestor documental que cumple todos los requisitos propuestos en la unidad 4.

Construyendo la aplicación de administración, la unidad que acabas de finalizar ha demostrado la potencia desarrollada cuando la arquitectura del *software* está diseñada para reutilizar y compartir código. Uno de los objetivos de la programación orientada a objetos es precisamente lograr un código reusable en el que cada objeto realice adecuadamente las funciones para las que ha sido diseñado y colabore eficazmente con sus objetos asociados. Esto es lo que hace *symfony* en su conjunto, lo que se ha mostrado a lo largo del curso y, de forma contundente, a lo largo de esta unidad.

Primero convertimos, con muy poco esfuerzo, el módulo de inicio de sesión de la aplicación *frontend* en un módulo compartido por cualquier otra aplicación gracias a los *plugins* de *symfony*. Ello hizo posible que en cuestión de minutos tuviésemos la aplicación *backend* provista de un mecanismo para realizar el registro de usuarios y el correspondiente inicio de sesión. Es más, este mismo *plugin* de inicio de sesión lo puedes reutilizar en cualquier otra aplicación que desarrolles. Y prácticamente cualquier aplicación *web* requiere el registro de usuarios. Así que ya tienes un problema prácticamente resuelto para tus futuras aplicaciones.

En segundo lugar hemos construidos tres módulos de administración para la gestión otras tantas tablas en menos de cinco minutos (todo depende de lo veloz que seas tecleando la frase: *symfony propel:generate-admin backend Usuarios --module=gesusu*), y hemos explicado las bases teóricas necesarias para modificar a nuestro antojo el comportamiento por defecto de estos módulos, las cuales se basan en:

- conocer las posibilidades del fichero de configuración del módulo de administración *generator.yml*,
- redefinir y extender, si fuese necesario, las acciones y plantillas del código generado “al vuelo” a partir del archivo *generator.yml*,
- redefinir y extender las clases del modelo, los formularios y los filtros de *Propel*.

Además hemos ilustrado estos puntos adaptando a nuestras necesidades los módulos de administración. A medida que los utilices y estudies la herramienta de generación automática de módulos de administración irás descubriendo nuevas y poderosas posibilidades que ofrecen. Esto, por supuesto, es extensible al resto del *framework* y a cualquier asunto relativo al desarrollo de software. Practicar, investigar, ensayar, errar, conseguir resultados y asimilarlos es la mejor, si no la única receta, que te permitirá desarrollar sistemas software de calidad.

Unidad 10: Internacionalización y enrutamiento

Comenzamos la última unidad del curso con el gestor de contenidos finalizado y completamente funcional. Su desarrollo nos ha servido como hilo conductor para explorar y aprender a utilizar una potente herramienta para la construcción de aplicaciones web: el *framework symfony*. Son muchos los conceptos, las ideas y estrategias que se han trabajado a lo largo del curso, siendo muchos de ellos de alcance bastante general, aunque se hayan concretados en el dominio de *symfony*. A pesar de todo, si continuamos explorando los detalles y recovecos del *framework*, seguiremos extrayendo nuevos tesoros. Hemos dado varias vueltas a nuestro objeto de estudio, y en cada una de ellas nos hemos acercado un poco más al centro, revelando nuevos y poderosos aspectos de *symfony* que nos permitirán mejorar la calidad de nuestros futuros desarrollos web. Sin embargo aún se pueden dar varias vueltas más. Y no dudes que extraerás conocimiento muy provechosos en cada nueva vuelta que des.

Esta unidad se plantea como el inicio de otra vuelta panorámica en la que se presentan, sin entrar en profundidad, nuevos aspectos de *symfony* que no han sido imprescindibles para desarrollar nuestro gestor documental pero que, sin lugar a duda, contribuirían a mejorarlo o serán muy útiles en la construcción de otras aplicaciones. Concretamente trataremos el tema de la internacionalización y localización de aplicaciones web y del enrutamiento de *symfony*.

Internacionalización y Localización

Internacionalizar una aplicación informática significa diseñarla de tal manera que pueda ser adaptada a distintos idiomas sin necesidad de realizar cambios en su ingeniería. Esto significa que, de alguna manera, los textos que se utilizan en su interfaz deben actuar en la aplicación como si de un componente intercambiable se tratase. Además también debe existir un proceso mediante el cual la aplicación presente la interfaz en un lenguaje u otro. Se suele utilizar la abreviatura *I18N* para referirse a la **Internacionalización**, ya que entre la primera letra (I) y la última (N) hay 18 letras más.

Por otro lado, localizar una aplicación es el proceso de adaptarla para que muestre ciertos datos, como las fechas, las monedas y los números, según los aspectos locales de distintas regiones o países sin alterar la ingeniería de la misma. Se suele utilizar la abreviatura *L10N* para referirse a la **Localización**. Adivina por qué.

En este apartado explicaremos cómo tratar estos aspectos con *symfony*.

La cultura del usuario.

La necesidad de internacionalizar y localizar una aplicación del tipo que sea surge del hecho de que dicha aplicación será utilizada por usuarios de distintas partes del mundo. Es natural, por tanto, que la información que recoja este hecho sea un atributo del usuario que está utilizando la aplicación. *Symfony* utiliza un atributo del objeto *sfUser*, o sea de la sesión de usuario, denominado *culture* para almacenar esta información.

La cultura de usuario es un parámetro compuesto por dos partes: el lenguaje y la localización. Los códigos de lenguaje consisten en dos letras minúsculas que siguen el estándar ISO-639-2. La siguiente tabla muestra los lenguajes más usuales para un desarrollador europeo:

Lenguaje	Código ISO-639-2
Español	es
Inglés	en
Francés	fr

Unidad 10: Internacionalización y enrutamiento

Alemán	de
Italiano	it
Portugués	pt

Los códigos de localización se definen por dos letras mayúsculas según el estándar ISO-3166. La siguiente tabla muestra alguno de los código correspondiente distintas regiones en las que se puede hablar el mismo idioma:

Región	Código ISO-3166
España	ES
Argentina	AR
Reino Unido	UK
Estados Unidos	US
Francia	FR
Bélgica	BE

La combinación de ambos código separados por un carácter “_” conforman el atributo cultura:

Cultura	Significado
es_ES	Español con localización de España
es_AR	Español con localización de Argentina
en_UK	Inglés con localización del Reino Unido
en_US	Inglés con localización de EEUU
fr_FR	Francés con localización de Francia
fr_BE	Francés con localización de Bélgica

Pues muy bien, la cuestión que importa realmente, al margen de todas estas convenciones, es saber como definir y recuperar el atributo *culture* de la sesión. Y ello se hace mediante los métodos *setCulture()* y *getCulture()* del objeto *sfUser*:

Definición de la cultura

```
<?php
// En una plantilla
$sf_user -> setCulture('es_ES');

// En una acción
$this -> getUser() -> setCulture('es_ES');
```

Recuperación de la cultura:

```
<?php
// En una plantilla
$culture = $sf_user -> getCulture();

// En una acción
```

```
$culture = $this -> getUser() -> getCulture();
```

Es obvio que trabajar con el parámetro *culture* tiene sentido si pretendemos internacionalizar nuestra aplicación. Por defecto *symfony* no tiene en cuenta para nada este parámetro. Si deseamos utilizar las características *I18N* de *symfony* lo primero que debemos hacer es indicárselo en el fichero de configuración *settings.yml* de la aplicación mediante el parámetro *i18n*:

Uso del parámetro i18n en el fichero apps/nombre_aplicacion/config/settings.yml

```
...
all
  .settings
    i18n: true
...
```

También podemos indicar la cultura por defecto, esto es, la cultura que utilizará *symfony* si no se especifica explícitamente con el objeto *sfUser*. Para ello utilizamos el parámetro *default_culture*: del archivo *settings.yml* de la aplicación.

Uso del parámetro default_culture en el fichero apps/nombre_aplicacion/config/settings.yml

```
...
all
  .settings
    default_culture: es_ES
...
```

Con lo que acabamos de ver basta para traducir al español la interfaz de la aplicación de administración de nuestro gestor documental. Para ello añade al archivo *apps/backend/config/settings.yml* el parámetro *i18n* y defínelo como *true*. Después modifica la acción *executeSignin()* del módulo *inises* para que coloque la cultura '*es_ES*':

Modificación de la acción executeSignin() del módulo inises para que tenga en cuenta la cultura

```
<?php
...
public function executeSignIn(sfWebRequest $request)
{
    $this -> form = new LoginForm();
    if ($request->isMethod('post'))
    {
        $datos = $request -> getParameter($this->form -> getName());
        $this->form->bind($datos);
        if ($this->form->isValid())
        {
            $usuario = $this -> comprouebaUsuario($datos);
            if($usuario instanceof Usuarios) // Existe el usuario con los datos dados
            {
                $this -> getUser() -> setAuthenticated(true);
                $this -> getUser() -> setAttribute('id_usuario', $usuario -> getIdUsuario());
...
```

```
        $this -> asociaCredenciales($usuario);
        $this -> getUser() -> setCulture('es_ES');
        $this -> redirect('@pagina_inicial');

    }
}
```

Traducción de la interfaz de las aplicaciones internacionalizadas*

```
        else
        {
            $this -> mensaje = 'Usuario no autorizado';
        }
    }
}
```

Ahora cuando entres en cualquiera de los módulos generados automáticamente comprobarás que se muestran en español. Esto es así por que dichos módulos están internacionalizados y cuentan con varias traducciones de la interfaz, entre ellas la traducción al español. En el próximo apartado explicaremos como podemos realizar las traducciones de nuestras interfaces.

En lugar de alterar el código del inicio de sesión podríamos haber definido en el archivo *settings.yml* de la aplicación *backend* el parámetro *default_culture* con el valor *es_ES*. Hemos preferido hacerlo en el código para mostrar un lugar apropiado donde se podría llevar a cabo la definición de la cultura en la sesión de usuario.

Según acabamos de ver, internacionalizar una aplicación *symfony* consiste por un lado en indicar al *framework* mediante el parámetro *i18n* del archivo de configuración *settings.yml* de la aplicación que deseas internacionalizarla, y por otro en indicar en la sesión de usuario qué cultura utilizar durante el uso de la aplicación. La selección de la cultura se puede hacer de muchas maneras.

Una posible manera es utilizando el proceso de inicio de sesión. Se podría utilizar un campo de la tabla de usuarios para indicar la cultura preferida de cada usuario. Entonces, cuando se inicia la sesión se definiría la cultura del objeto *sfUser* según el valor almacenado en dicho campo. Modificar el código anterior para implementar esta idea es muy sencillo.

Otra posible manera sería mediante la implementación de un formulario mediante el cual el usuario pueda seleccionar una de las culturas admitidas por la aplicación. El *plugin* público *sfFormExtraPlugin* ofrece una solución de este tipo, con lo que el esfuerzo para implementar dicha solución es mínimo.

También se puede obtener la cultura a partir de la información proporcionada por el header *HTTP Accept-Language* de la petición. El método *getLanguages()* del objeto *request* devuelve un array de lenguajes aceptados por el navegador web del cliente. Existe un método del mismo objeto que puede resultar incluso más útil: *getPreferredCulture(array())* que devuelve la cultura aceptada por el cliente web que mejor se ajusta a la lista de culturas proporcionada en el argumento de la función.

En fin, la forma concreta que utilices para definir la cultura de usuario dependerá de las especificaciones de la aplicación que desarrolles. Aunque los más probable es que una de estas tres soluciones o alguna mezcla de ellas te resulte suficiente.

Una vez que la cultura de usuario esté definida, *symfony* presentará la interfaz en el idioma correspondiente. Obviamente el *framework* no es tan listo como para realizar él mismo la traducción, simplemente sabe como buscar los textos, que nosotros debemos facilitarle, en los distintos idiomas que soporte la aplicación. En el siguiente apartado veremos como se lleva a cabo la traducción de la interfaz.

Traducción de la interfaz de las aplicaciones internacionalizadas*

Si queremos presentar los textos de la interfaz de usuario en distintos idiomas, lo primero que debemos hacer es internacionalizar la aplicación, es decir, poner el parámetro *i18n* del archivo *settings.yml* a *true*.

Traducción de campos de tablas de la base de datos

Posteriormente, en las plantillas que vayamos a traducir, utilizamos el helper *I18NHelper* ya que en él se encuentra la función clave del proceso de traducción de la interface de usuario. Esta función se llama `__()`. Un nombre un poco raro, ya que estrictamente no es un nombre, se trata dos veces el carácter “_”, lo cual es válido como nombre de función *PHP*. El misterio de la traducción es que todos los textos que pasemos como argumento a la función `__()` serán sustituidos, si existe, por el valor indicado en un fichero *XML* donde se encuentran las traducciones al idioma en cuestión. En caso de no existir traducción, la función devuelve el mismo texto que se le pasó como argumento. Veámoslo con un ejemplo:

Ejemplo de una plantilla traducible

```
<?php echo use_helper('I18N') ?>

<h1><?php echo __('Listado de frutas'); ?></h1>

<ul>
    <li><?php echo __('Manzana') ?></li>
    <li><?php echo __('Naranja') ?></li>
    <li><?php echo __('Limon') ?></li>
</ul>
```

Por otro lado, para que la función `__()` sepa qué texto debe colocar en función de la cultura, debemos crear un fichero de traducciones por cada idioma que deseemos utilizar. Los argumentos que se pasan a la función se consideran como idioma fuente (*source-language*). Este fichero de traducciones sigue el standard *XLIFF* y se debe ubicar en el directorio *i18n* de la aplicación. El archivo de traducción al inglés de los términos anteriores tendría el siguiente aspecto:

Archivo XLIFF de traducción español-inglés: apps/nombre_aplicacion/i18n/en/messages.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN" "http://www.oasis-open.org/committees/xliff/documents/xliff.dtd">
<xliff version="1.0">
    <file source-language="es" target-language="en" datatype="plaintext" original="messages" date="2010-04-29T19:12:10Z" product-name="messages">
        <header/>
        <body>
            <trans-unit id="1">
                <source>Listado de frutas</source>
                <target>List of fruits</target>
            </trans-unit>
            <trans-unit id="2">
                <source>Manzanas</source>
                <target>Apple</target>
            </trans-unit>
            <trans-unit id="3">
                <source>Naranjas</source>
                <target>Orange</target>
            </trans-unit>
            <trans-unit id="4">
                <source>Limon</source>
                <target>Lemon</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Traducción de campos de tablas de la base de datos

En ocasiones también será necesario traducir algunos de los campos de ciertas tablas de la base de datos. *Symfony* también proporciona un procedimiento para realizar esta tarea. Lo explicaremos con un ejemplo para que resulte más concreto y sencillo de seguir. La generalización a cualquier otro caso es inmediata.

Supongamos que deseamos traducir una tabla denominada *frutas* que cuenta, en principio, con los siguientes campos:

- *id (clave principal)*
- *nombre*

Enrutamiento

- *descripcion*
- *kcal_gramo*
- *proteinas_gramo*

Lo normal es que deseemos tener traducciones de los campos *nombre* y *descripcion*, ya que el valor de los demás no dependen del idioma. Lo que se hace es repartir los campos en dos tablas, una que contendrá los campos cuyos valores son dependientes del idioma y la otra con los que no lo son:

- tabla *frutas*: (independiente del idioma) * *id* (*clave principal*) * *kcal_gramo* * *proteinas_gramo*
- tabla *frutas_i18n*: (dependiente del idioma) * *id* (*clave principal*) * *cultura* (*clave principal*) * *nombre* * *descripcion*

En la segunda tabla, denominada tabla de idiomas o tabla i18n, se especifica como clave principal el par [*id*, *cultura*], y el campo *id* se corresponde con el campo *id* de la tabla *frutas*. De esta manera cada registro de la tabla *frutas* puede tener N registros asociados de la tabla *frutas_i18n* con distintos valores del campo *cultura*.

Ahora hay que especificar en el archivo *schema.yml* el hecho de que la tabla *frutas_i18n* contiene valores dependientes de la cultura para realizar las traducciones:

Contenido del archivo config/schema.yml para definir la internacionalizacion de los campos de una tabla

```
miConexion:  
    frutas:  
        _attributes: { phpName: Frutas, isI18N: true, i18nTable: frutas_i18n }  
        id: { type: integer, required: true, primaryKey: true, autoincrement: true }  
        kcalGramo: { type: float }  
        proteinasGramo: { type: float }  
    frutas_i18n:  
        _attributes: { phpName: FrutasI18n }  
        id: { type: integer, required: true, primaryKey: true, foreignTable: frutas, foreignReference: id }  
        cultura: { isCulture: true, type: varchar, size: 7, required: true, primaryKey: true }  
        nombre: { type: varchar, size: 50 }  
        descripcion: { type: varchar, size: 250 }
```

Observa los valores resaltados en negrita en la definición de ambas tablas, pues son los que indican la internacionalización de la tabla *frutas*. En la tabla principal, es decir, la que no depende del idioma, se indica tanto que es traducible (mediante el atributo *is18N*), como la tabla asociada que tiene sus campos traducibles (mediante el atributo *i18nTable*). Por otro lado, se debe especificar en la tabla de traducciones cual es el campo que representa la cultura (mediante el atributo *isCulture*).

Cuando generemos el modelo tendremos disponible un objeto denominado *Frutas* que se utilizará como cualquier otro. La peculiaridad es que cuando utilicemos los métodos *getters* y *setters* correspondientes a los campos traducibles *nombre* y *descripcion*, *symfony* tendrá en cuenta la cultura de usuario especificada en la sesión, para mostrar o definir el valor que le corresponda al campo en función de tal cultura.

Por ejemplo, si la cultura de usuario es '*es_ES*', el método *\$fruta -> getNombre()* devolverá el valor del registro de la tabla *frutas_i18n* con valor del campo *cultura* '*es_ES*', y si la cultura es '*en_UK*' pues el valor devuelto será el del registro de la tabla *frutas_i18n* con valor del campo *cultura* '*en_UK*'. Con menos palabras, el método devuelve la traducción que le corresponda en el idioma de la cultura que tenga el usuario. Por supuesto para que esto ocurra deben existir las traducciones en los idiomas que se manejen.

Enrutamiento

Enrutamiento

La *URL* (*Uniform Resource Locator*) es una pieza clave del protocolo *HTTP*. Mediante ellas se especifican de manera única los recursos *HTTP* disponibles en la red. Son cadenas de texto que contienen toda la información necesaria para solicitar un recurso a un servidor *web*. En dicha cadena, tal información aparece estructurada utilizando algunos caracteres especiales como separador. Vamos a diseccionar una URL en cada una de sus partes. Partimos del siguiente ejemplo ficticio que representa una URL completa, con todos sus elementos obligatorios y opcionales:

`http://usuario:clave@nombredelhost.org:80/ruta/recurso.html?var1=valor1&var2=valor2#ancla`

La siguiente tabla muestra la *URL* diseccionada:

Parte	Descripción
http	Esquema o protocolo (obligatorio)
usuario	Nombre de usuario (opcional)
clave	Clave del usuario (opcional)
nombredelhost.org	Ubicación del <i>host</i> . Normalmente esta ubicación se hace mediante un nombre que está registrado en un espacio <i>DNS</i> , pero también se puede usar la dirección <i>IP</i> del <i>host</i> (obligatorio)
80	Puerto donde escucha el <i>host</i> . (opcional, si no se especifica se usa 80 como valor)
ruta/recurso.html	Ruta interna del servidor <i>web</i> al recurso. (obligatorio)
var1=valor1&var2=valor2	Cadena con datos de la petición, más conocida como <i>query string</i> . El carácter & se usa para separar los datos (opcional)
ancla	Fragmento del recurso que se desea mostrar (opcional)

A lo largo del curso hemos comprobado que las *URL*'s que utiliza *symfony* prescinden de los caracteres "?", "&" y "=" propios de la *query string* dando lugar a *URL*'s más homogéneas y "elegantes" que únicamente utilizan el carácter "/" como separador. Así una *URL* como la siguiente:

`http://nombredelhost.org/gestordocumental/index.php?module=gesdoc&action=verVersion&id_ve`

En *symfony* se convierte en:

`http://nombredelhost.org/gestordocumental/index.php/gesdoc/verVersion/id_version/4`

Esta simplificación de la *URL* es posible gracias al sistema de enrutamiento de *symfony* el cual permite traducir *URL*'s "estilizadas" como esta última, a *URL*'s "clásicas" que son más largas y engorrosas como la primera. A las *URL*'s del tipo "estilizado", *symfony* las denomina *URL*'s **externas**, ya que son las que un cliente manipula para realizar las peticiones, mientras que las *URL*'s "clásicas" son denominadas **internas**.

Observa que además de la "estilización" que provoca el hecho de eliminar los caracteres "?", "&" y "=" de la *URL*, también se obtienen cadenas más cortas, ya que se prescinde del nombre de algunos parámetros. Concretamente no aparecen en las *URL*'s externas los nombres de parámetro *module* y *action*; *symfony* ya reconoce que los dos primeros valores tras el controlador se corresponden con los parámetros *module* y *action* precisados por el *framework* para su ejecutar una determinada acción de un módulo.

Así pues se ha conseguido estilizar y disminuir la longitud de las *URL* que se manejan fuera de la aplicación. Pero lo más importante de todo es que se ha eliminado de la *URL* información acerca de la estructura de la aplicación, manteniendo exclusivamente

Enrutamiento

información que describe al recurso localizado por dicha *URL*. Y es que idealmente las *URL's*, como “localizadores de recursos” que son, deberían diseñarse de manera que ofrezcan información relacionada con el recurso que tienen asignado y nada más, de manera que no se pueda deducir a través de ella ningún detalle interno de la aplicación que proporcione el recurso solicitado. Ni siquiera el lenguaje de programación con el que se ha programado. Esto último podemos lograrlo eliminando de la *URL* el nombre del controlador frontal (*index.php* normalmente) activando la opción *no_script_name* en el fichero *settings.yml* de la aplicación:

Activación del ocultamiento del nombre del controlador frontal de las URL's externas de la aplicación.

```
prod:  
  .settings:  
    no_script_name: true
```

Con este último ajuste las URL's externas de symfony quedarían como sigue:

http://nombredelhost.org/gestordocumental/gesdoc/verVersion/id_version/4

Ahora la *URL* informa únicamente sobre el recurso, sin dar ninguna información sobre la tecnología utilizada para recuperar el recurso, ni sobre aspectos propios de la propia aplicación como son el nombre de los parámetros *module* y *action*. Es más, en principio no se sabe ni siquiera si se trata de un recurso estático o generado dinámicamente por una aplicación del lado de servidor. Aún queda en la *URL* el nombre del parámetro *id_version*. Veremos en breve como también podemos eliminarlo gracias a este gran invento que es el sistema de enrutamiento de symfony.

La pregunta que surge después de esta digresión es: si eliminamos los nombres de los parámetros de las URL's externas, ¿cómo sabe la aplicación qué valores corresponde a qué parámetros?, ¿cómo sabe lo que debe hacer con los valores que se le están pasando?.

Y la respuesta: mediante el sistema de enrutamiento, el cual utiliza una técnica de mapeo de URL's externas estilizadas y desprovistas de información propia de la aplicación en URL's internas con toda la información que la aplicación requiere (y viceversa). Este mapeo o transformación se realiza, como cualquier otro, según unas reglas predefinidas. De otra manera sería imposible añadir la información que falta cuando se realiza la transformación de URL externa a interna, o eliminar la información que sobra cuando la transformación es de URL interna a externa. La clave del enrutamiento está, por tanto, en este conjunto de reglas, las cuales se definen en el archivo de configuración *routing.yml* de la aplicación.

En el momento en que se crea una nueva aplicación, el fichero *routing.yml* de la misma muestra el siguiente contenido:

Archivo de rutas (*routing.yml*) de una aplicación recién generada

```
# default rules  
homepage:  
  url: /  
  param: { module: default, action: index }  
  
# generic rules  
# please, remove them by adding more specific rules  
default_index:  
  url: /:module  
  param: { action: index }
```

```
default:  
    url:   /:module/:action/*
```

En él se definen 3 rutas (o reglas de enrutamiento) que son las rutas por defecto, suficientes para desarrollar una aplicación con *symfony*. Aunque si deseamos que nuestra aplicación presente *URL's* estilizadas para todas sus acciones, debemos añadir nuevas reglas.

Cada ruta consta de un nombre de la ruta (*homepage*, *default_index* y *default*, son los nombres de las 3 rutas anteriores), de un patrón de la ruta identificado por la clave *url*, y unos parámetros identificados por la clave *param*.

El nombre de la ruta puede ser cualquiera, pero debe ser único. Desde dentro de la aplicación podemos referirnos a ellas anteponiendo el carácter '@' a su nombre. Recuerda que hicimos esto en el capítulo anterior cuando adaptábamos el módulo de inicio de sesión a un *plugin* para que pudiese ser compartido por varias aplicaciones.

La clave *url*, es un patrón que se aplica a las *URL's* externas (únicamente a la parte que va inmediatamente después del controlador frontal, es decir, donde se ubica la *query string* de la *URL*), de manera que en el momento en que se encuentra una coincidencia se construye la *URL* interna a partir de los datos ofrecidos por la regla (ruta) en cuestión a través de las claves *url* y *params*. Los patrones de ruta son cadenas de elementos separados por el carácter "/". Cada elemento puede ser un literal o un parámetro, en cuyo caso comienza con el carácter ":".

Vamos a mostrarlo con ejemplos, ya que es la mejor manera de comprender el funcionamiento del *routing*. Comenzamos por interpretar las tres rutas creadas por defecto cuando se genera una nueva aplicación.

La primera de ellas, denominada *homepage*, dice lo siguiente: "Cuando la *URL* externa no lleve ningún elemento, entonces usa *default* como valor del parámetro *module*, y *index* como valor del parámetro *action*".

La segunda, denominada *default_index*, dice que: "Cuando la *URL* externa presente un único elemento, se trata de un parámetro cuyo nombre es *module* (eso es lo que significa el carácter ":" delante del elemento), y que se utilice *index* como valor del parámetro *action*".

La tercera, denominada *default*, dice que: "Cuando la *URL* externa presente dos o más elementos, el primero es un parámetro cuyo nombre es *module* y el segundo es otro parámetro cuyo nombre es *action* (observa de nuevo el carácter ":") el resto de los elementos deberían venir dados por pares del tipo *nombre_parametro/valor_parametro*, para que la petición sea parseada adecuadamente".

Estas tres reglas explican el comportamiento por defecto que venimos comprobando desde el principio del curso de las rutas "estilizadas" de *symfony*. Concretamente la última de ellas hace posible que para ejecutar una acción determinada de un módulo tan sólo tengamos que indicar los nombres de la acción y el módulo en cuestión, obviando el nombre de los parámetros *module* y *action*.

Hemos dado una lectura de las reglas desde la perspectiva de transformar una *URL* externa en una interna. No obstante las mismas reglas son utilizadas para transformar las *URL* internas en externas. Esto se hace desde dentro de la aplicación cuando utilizamos *helpers* de *URL* como *url_for()* o *link_to()*. Existen varias formas de expresar las *URL's* internas, la más común de todas ellas es la que venimos utilizando durante todo el curso:

Forma típica de especificar una ruta interna en symfony:

```
nombre_modulo/nombre_accion?param1=valor1&param2=valor2&... .
```

Enrutamiento

En el ejemplo que venimos mostrando en este apartado, la transformación desde una plantilla de una *URL* interna en externa se haría así:

```
<?php  
url_for('gesdoc/verVersion?id_version=4');
```

Para lo cual estamos usando la regla denominada *default*. Pero también podemos indicar la misma *URL* interna haciendo uso del nombre de la regla. En ese caso se especificaría de la siguiente forma:

```
<?php  
url_for('@default?module=gesdoc&action=verVersion&id_version=4');
```

Aún existe una tercera forma de especificar las url internas, más estructurada si cabe:

```
url_for(array(  
    'module'      => 'gesdoc',  
    'action'       => 'verVersion',  
    'id_version'  => 4  
));
```

Ahora que ya sabemos los fundamentos de la transformación de *URL* internas y externas (y viceversa) a través de las reglas definidas en el fichero de configuración *routing.yml*, vamos a ver como podemos añadir nuevas reglas para estilizar aún más las rutas de las aplicaciones de nuestro proyecto de gestión documental. Comenzamos con la aplicación *frontend*, que está compuesta por un sólo módulo denominado *gesdoc*. Repasaremos todas las acciones del módulo con sus *URL* internas respectivas, y a partir de ellas realizaremos una propuesta para las *URL* externas deseadas:

URL interna	URL externa deseada
<i>inises/signIn</i>	<i>conectar</i>
<i>inises/signOut</i>	<i>desconectar</i>
<i>gesdoc/index</i>	/
<i>gedoc/index?page=n</i>	<i>/pagina/n</i>
<i>gesdoc/verMetadatos?id_documento=n</i>	<i>metadatos/n</i>
<i>gesdoc/verVersion?id_version=n</i>	<i>version/n</i>
<i>gesdoc/modificar?id_documento=n</i>	<i>modificar/n</i>
<i>gesdoc/subirVersion?id_documento=n</i>	<i>nueva_version/n</i>

Según lo que hemos explicado a lo largo de este apartado, el mapeo de rutas propuesto se puede conseguir añadiendo las siguientes reglas de enrutamiento al fichero *routing.yml* de la aplicación *frontend*.

Rutas añadidas al archivo apps/frontend/config/routing.yml para estilizar todas las acciones de la aplicación frontend.

```
...  
conectar:  
    url: /conectar  
    param: { module: inises, action: signIn }
```

Enrutamiento

```
desconectar:  
    url: /desconectar  
    param: { module: inises, action: signOut }  
  
ver_metadatos:  
    url: /metadatos/:id_documento  
    param: { module: gesdoc, action: verMetadatos }  
  
ver_version:  
    url: /version/:id_version  
    param: { module: gesdoc, action: verVersion }  
  
modificar:  
    url: /modificar/:id_documento  
    param: { module: gesdoc, action: modificar }  
  
subir_version:  
    url: /nueva_version/:id_documento  
    param: { module: gesdoc, action: subirVersion }  
  
pagina:  
    url: /pagina/:page  
    param: { module: gesdoc, action: index }  
...
```

Y para rizar el rizo podemos eliminar en el entorno de producción el nombre del controlador frontal *index.php*:

Activación del ocultamiento del nombre del controlador frontal de las URL's externas de la aplicación.

```
prod:  
    .settings:  
        no_script_name: true
```

Ya puedes probar las nuevas rutas añadidas. No te olvides de borrar la cache antes de probar los cambios. Ahora las *URL* externas, que podemos considerar como parte integrante de la interfaz gráfica de usuario, presentan un aspecto mucho más elegante de cara al usuario, a la vez que le ocultan datos estructurales relativos a la aplicación, mejorando de esta forma la seguridad de la misma.

Por último, en la aplicación *backend* no tenemos que realizar ninguna estilización de las rutas ya que todos sus módulos han sido construidos con el generador automático de módulos de administración, el cual se encarga de añadir al archivo *routing.yml* correspondiente las reglas necesarias para conseguir ocultar la información sensible de las *URL* externas y estilizarlas. Échale un vistazo al archivo *routing.yml* de la aplicación *backend* y podrás comprobar el aspecto de estas rutas. Comprobarás que son más complejas que las que hemos descrito. Ello se debe a que utilizan el concepto de colecciones de rutas.

Nota

Toda la información sobre las posibilidades que brindan las rutas de *symfony* las puedes encontrar en la guía de referencia.

También puedes encontrar más explicaciones sobre el sistema de enrutamiento en la siguiente URL:

http://www.symfony-project.org/gentle-introduction/1_4/en/09-Links-and-the-Routing-System

Conclusión

En esta unidad hemos tratado dos nuevos aspectos que enriquecerán nuestras aplicaciones *web*: la internacionalización y el *routing*. Aspectos para los que *symfony*, una vez más, ofrece potentes soluciones.

La internacionalización es resuelta, tanto a nivel de interfaz de usuario como de registros de la base de datos, mediante la existencia de un parámetro perteneciente a la sesión de usuario y que se denomina *culture*. Mirando el valor que dicho parámetro contiene en una determinada sesión, el *framework* sabe qué textos, en el caso de la interfaz de usuario, o qué registros, en el caso de la base de datos, debe mostrar. Además, para la traducción de la interfaz de usuario se utilizan los catálogo *XLIFF* que son ficheros *XML* estándar bien conocidos en el gremio de los profesionales de la traducción, con lo que se facilita la intercomunicación y operatividad entre los programadores y los traductores, de la misma manera que la adopción del patrón *MVC* facilita el trabajo entre programadores y diseñadores *web*.

Por otro lado el *routing* proporciona la estilización de las *URL's* que son manipuladas en la parte del cliente, es decir, las que aparecen tanto en la barra de direcciones del navegador como en los enlaces que contengan los documentos *HTML* que reciba. Aunque en un primer momento esto no parezca muy importante para la aplicación, pues no es imprescindible para que funcione, si nos proponemos construir aplicaciones realmente profesionales y seguras no tendremos más remedio que utilizar el *routing*. Por un lado simplifica y estiliza las largas *URL's* que se muestran en el cliente. Pero lo más importante es que dicha simplificación se realiza eliminando información asociada a la estructura de la aplicación, información que no es necesaria para describir el recurso que se solicita y que podría ser utilizada por el usuario con fines maliciosos. Es una regla básica de seguridad no dar más información que la estrictamente necesaria.

Con estos dos “extras” finaliza el curso. Ya se han tratado los principales aspectos del *framework* con los que se pueden construir aplicaciones *web* de calidad. De hecho el desarrollo de una aplicación completa ha vertebrado el desarrollo del curso que ahora terminas. Esta misma aplicación, con más o menos cambios, junto con las herramientas que has adquirido durante el seguimiento del curso, te servirán como semilla para desarrollar prácticamente cualquier tipo de aplicación. No obstante, a nuestro modo de ver, lo más importante es que ya dispones del bagaje suficiente para sentirte cómodo con *symfony* y comenzar a explorar cada una de sus características en profundidad a medida que las vayas necesitando en tu trabajo.

Indices y tablas

- *genindex*
 - *search*
-

1 "Patrones de Diseño" de los autores Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos como *The Gun of Four*) es un clásico en la literatura sobre este tema.

2 http://en.wikipedia.org/wiki/Separation_of_concerns

3 En el caso de *Apache con PHP*, que es el que nos interesa en este curso, el servidor debe estar configurado adecuadamente para que se puedan incluir archivos *PHP* que están fuera del *Document root**. Esto se hace con la directiva *open_basedir*.

4 http://es.wikipedia.org/wiki/Decorator_%28patr%C3%B3n_de_dise%C3%BDo%29