

Título

mes año



Licencia CC-BY-SA: <http://creativecommons.org/licenses/by-sa/3.0/deed.es>

Este proyecto y todos sus documentos han sido realizados utilizando únicamente **Software Libre**

Índice de contenido

1. PREPARANDO EL ENTORNO DE DESARROLLO.....	4
Instalando los requisitos necesarios.....	4
Configuración Netbeans.....	4
Instalar Symfony2.....	5
Configuración de apache2.....	5
Configuración de PostgreSQL.....	7
Verificar los requerimientos para Symfony2.....	8
Configuración de XDebug (Opcional).....	9
Utilizar reportes de jasperreport en PHP (Opcional).....	11
2. FUNDAMENTOS DEL FRAMEWORK MVC.....	17
Patrón de diseño MVC.....	17
De archivos PHP planos a Symfony.....	18
Flujo de una aplicación en Symfony.....	29
3. CONFIGURACIÓN INICIAL DEL PROYECTO.....	31
La estructura de directorios.....	31
web/: Este es el directorio raíz de la aplicación web y contiene todos los archivos accesibles publicamente	31
Entornos.....	31
Realizar la configuración inicial.....	33
4. EL MODELO.....	36
Base de datos y Symfony2.....	36
Un modelo no es una tabla.....	36
Doctrine DBAL.....	37
Configuración.....	38
Doctrine ORM.....	39
5. PRIMEROS PASOS.....	42

Creando páginas en Symfony2.....	42
Los Bundles.....	42
El nombre de un bundle.....	43
Crear la ruta.....	43
Crear el controlador.....	45
Crear la plantilla.....	47
Acceso a los datos.....	50
Mostrar los datos de un producto.....	52
6. FORMULARIOS.....	56
Creando un simple formulario.....	56
Contruyendo el formulario.....	56
Gestionando el envío del formulario.....	61
Validar el formulario.....	63
Renderizando el formulario en la plantilla.....	65
Renderizando manualmente cada campo.....	66
Utilizar temas de formularios.....	67
Estilos.....	69
7. MÁS CON EL MODELO.....	74
El objeto QueryBuilder.....	75
Usando el QueryBuilder en el campo Entity.....	77
Clases repositorio.....	78
8. GENERADORES INTERACTIVOS: Intefaces CRUD.....	83
Crear la aplicación para la planificación de compras.....	83
9. JAVASCRIPT.....	95
Gestión del detalle de plan de compras.....	95
Índice de tablas.....	111
Índice de ilustraciones.....	111

1. PREPARANDO EL ENTORNO DE DESARROLLO

Instalando los requisitos necesarios

```
$ sudo apt-get install php5 php5-pgsql php5-sqlite php5-xdebug php-apc  
php5-cli php5-xsl php5-intl apache2 postgresql acl
```

Github

Después de seguir la guía de configuración de github cuando se prueba la conexión debe hacerse esto para que funcione: `ssh-add ~/.ssh/id_rsa`

Configuración Netbeans

En el desarrollo de este manual se ha utilizado Netbeans 7.0 para PHP como IDE de desarrollo, no se explicará la instalación en sí (es necesario instalar la máquina virtual de java y bajar Netbeans para php), sino algunos ajustes para mejorar el desempeño de este:

1. Una de las tareas que relentiza Netbeans es “scanning projects”.
 - 1.1. Tools → Options
 - 1.2. Miscellaneous → Files
 - 1.3. Desmarcar la opción “Enable auto-scanning” of sources”
2. Instalar plugin para plantillas Twig
 - 2.1. Descargar la versión más actualizada del plugin <https://github.com/blogsh/Twig-netbeans/downloads>
 - 2.2. Tools → Plugins
 - 2.3. Downloaded → Add Plugins...
 - 2.4. Buscar y abrir el archivo descargado en el punto 2.1
 - 2.5. Marcar el plugin y clic en Install.

2.6. Reiniciamos Netbeans

3. Desactivar plugin CVS, también carga Netbeans y no lo utilizaremos

3.1. Tools → Plugins

3.2. Installed

3.3. Buscar y seleccionar el plugin CVS

3.4. Clic en Deactivate

Ahora creemos un proyecto en Netbeans con el nombre *compras*.

Instalar Symfony2

Descargamos [Symfony2](#) y descromprimos el archivo; se creará una carpeta Symfony, todo el contenido lo movemos en la carpeta correspondiente (En este caso ya se creó un proyecto en Netbeans con el nombre *compras* ~/NetbeansProjects/compras).

Para que no tengamos problemas recordemos dar permisos a las carpetas con los siguientes comandos (desde este momento y hasta el final de la guía todos los comandos que ejecutemos en la consola asumirán que estamos dentro de la carpeta ~/NetbeansProjects/compras)

```
$ rm -rf app/cache/*  
$ rm -rf app/logs/*
```

Dar los permisos por medio de acl

```
$ sudo setfacl -R -m u:www-data:rwX -m u:nombre_usuario:rwX app/cache app/logs  
$ sudo setfacl -dR -m u:www-data:rwX -m u:nombre_usuario:rwX app/cache app/logs
```

En ubuntu, para que se pueda utilizar acl se debe editar el archivo /etc/fstab, buscar la línea del punto de montaje donde está el proyecto de symfony y

agregar **,acl** después de **default**. Será necesario reiniciar el sistema.

Configuración de apache2

Lo primero es definir el VirtualHost que utilizaremos, para esta guía he utilizado esta definición:

```
<VirtualHost 127.0.0.4>
    ServerName compras.localhost
    DocumentRoot /ruta_a_proyecto_symfony/compras/web
    <Directory /ruta_a_proyecto_symfony/compras/web >
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Order allow,deny
        allow from all
    </Directory>
    ErrorLog ${APACHE_LOG_DIR}/error-sf2.localhost.log
    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn
    CustomLog ${APACHE_LOG_DIR}/access-sf2.localhost.log combined
</VirtualHost>
```

En ubuntu eso se hace de la siguiente forma:

En una terminal escribimos:

```
$ sudo nano /etc/apache2/sites-available/compras.localhost
```

Ahí escribimos la definición para el VirtualHost mostrada arriba (las rutas cambiarán según la ubicación que quieras darle), presionamos ctrl+o para guardar el archivo y ctrl+x para salir.

Y luego activamos el VirtualHost:

```
$ cd /etc/apache2/sites-available/
$ sudo a2ensite compras.localhost
```

Luego será necesario registrar el nombre de servidor que hemos definido en el VirtualHost en el archivo *hosts*

```
$ sudo nano /etc/hosts
```

Y se agrega la línea :

```
127.0.0.4      compras.localhost
```

Puedes utilizar cualquier ip que haga referencia al localhost con el cuidado de que sea la misma en la definición del VirtualHost y en el archivo hosts, si se utiliza una diferente a 127.0.0.1 será necesario modificar algunos archivos (Ej.: *app_dev.php* como veremos más adelante), cuando al ejecutar obtengas un error que el script solo se puede ejecutar desde 127.0.0.1 se debe agregar la ip que hayas utilizado en la definición del VirtualHost

Ahora reiniciemos el servicio de apache:

```
$ sudo /etc/init.d/apache restart
```

Configuración de PostgreSQL

En esta guía se utilizará PostgreSQL como gestor de base de datos con una base ya existente, primero realizaremos un cambio en la configuración de PostgreSQL para que permita la autenticación de usuarios por medio de una clave.

```
$ sudo nano /etc/postgresql/8.4/main/pg_hba.conf
```

Modificar la línea, en lugar de `ident` colocar `md5`:

```
# "local" is for Unix domain socket connections only
local    all                all                                md5
```

Reiniciar el servicio

```
$ sudo nano /etc/init.d/postgresql restart
```

Asumiendo que la base de datos se llamará *comprasdb* (se posee una copia de respaldo en el archivo *comprasdb.dump* el cual será restaurado) y el usuario a crear será *admin*, ejecutemos una a una las siguientes líneas:

```
$ sudo su postgres
$ createuser --createdb --pwprompt admin
$ createdb comprasdb --owner admin
$ pg_restore -d comprasdb comprasdb.dump
```

Verificar los requerimientos para Symfony2

En la carpeta *app/* existe un archivo para verificar los requerimientos de Symfony2, éste está pensado para ejecutarse en consola, pero como podría utilizarse un archivo de configuración de PHP diferente en consola y en la página web, por eso copiaremos el archivo *app/check.php* al directorio *web* (debemos borrarlo al finalizar la verificación). Antes de ejecutarlo vamos a realizar unos cambios, puesto que el archivo está pensado para ejecutarse en la consola y no en la web, abrir el archivo y reemplazar “\n” por “
” para mejorar la lectura de los resultados. Luego desde el navegador escribiremos:

```
http://compras.localhost/check.php
```

Donde *compras.localhost* es el nombre del VirtualHost en la configuración de

apache

La salida del script *check.php* puede ser algo similar a esto:

```
*****
* *
* Symfony requirements check *
* *
*****

php.ini used by PHP: /etc/php5/apache2/php.ini

** WARNING **
* The PHP CLI can use a different php.ini file
* than the one used with your web server.
* If this is the case, please ALSO launch this
* utility from your web server.
** WARNING **

** Mandatory requirements **

OK Checking that PHP version is at least 5.3.2 (5.3.5-1ubuntu7.2 installed)
OK Checking that the "date.timezone" setting is set
OK Checking that app/cache/ directory is writable
OK Checking that the app/logs/ directory is writable
OK Checking that the json_encode() is available
OK Checking that the SQLite3 or PDO_SQLite extension is available

** Optional checks **

OK Checking that the PHP-XML module is installed
OK Checking that the libxml version is at least 2.6.21
OK Checking that the token_get_all() function is available
OK Checking that the mb_strlen() function is available
```

Illustration 1: Salida del script check.php

Antes de continuar debemos solvertar todos los problemas que encontremos, aunque algunos son opcionales deberíamos tratar de solventarlos.

Configuración de XDebug (Opcional)

Al instalar Xdebug se agrega y activa pero no con todas sus capacidades, por ejemplo vamos a agregar los parámetros necesarios para realizar una depuración desde Netbeans, para esto agregamos al archivo `/etc/php5/conf.d/xdebug.ini` las siguientes líneas:

```
xdebug.remote_enable=true
xdebug.remote_host=127.0.0.1
xdebug.remote_port=9000
```

```
xdebug.remote_handler=dbgp
```

Además será necesario editar el archivo `/etc/php5/apache2/php.ini`, buscar la línea

```
html_errors = Off
```

y cambiarla por `html_errors = On`

La instalación de XDebug es opcional pero recomendada ya que nos permitirá realizar depuración paso a paso desde el IDE, además sustituye los errores y la función `var_dump` por algo más útil:

Sin Xdebug:

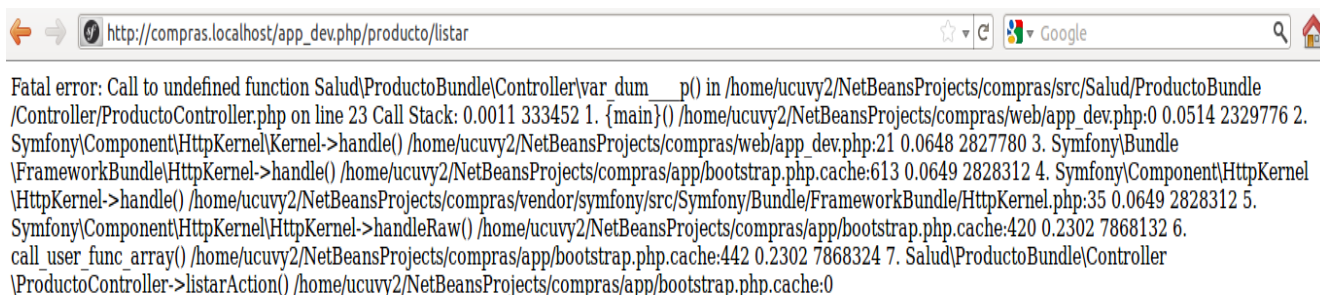
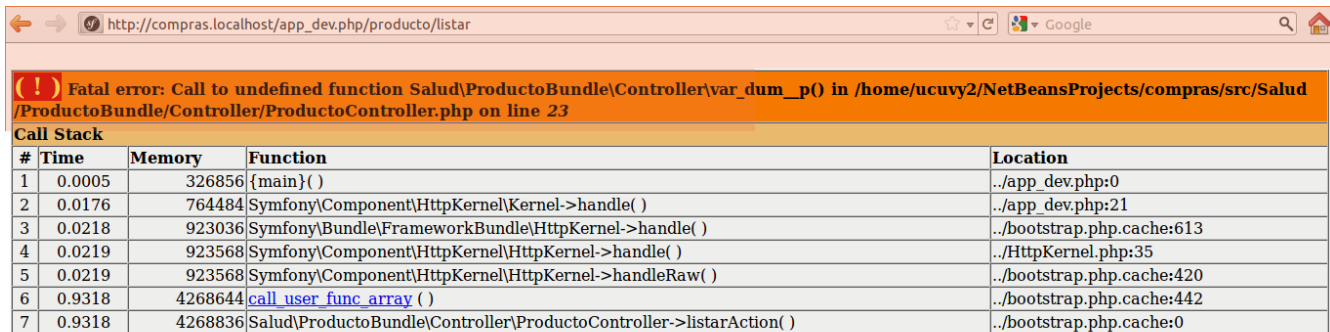


Illustration 2: Salida en una función mal escrita (Sin XDebug)



Illustration 3: Salida de la función var_dump aplicada sobre un arreglo (Sin XDebug)

Con Xdebug Activado:



Call Stack

#	Time	Memory	Function	Location
1	0.0005	326856	{main}()	./app_dev.php:0
2	0.0176	764484	Symfony\Component\HttpFoundation\HttpKernel->handle()	./app_dev.php:21
3	0.0218	923036	Symfony\Bundle\FrameworkBundle\HttpKernel->handle()	./bootstrap.php.cache:613
4	0.0219	923568	Symfony\Component\HttpFoundation\HttpKernel->handle()	./HttpKernel.php:35
5	0.0219	923568	Symfony\Component\HttpFoundation\HttpKernel->handleRaw()	./bootstrap.php.cache:420
6	0.9318	4268644	call_user_func_array()	./bootstrap.php.cache:442
7	0.9318	4268836	Salud\ProductoBundle\Controller\ProductoController->listarAction()	./bootstrap.php.cache:0

Illustration 4: Salida de una función mal escrita (Con XDebug)



```
array
  0 =>
    object(Salud\ComprasBundle\Entity\Item) [217]
      private 'id' => int 40
      private 'descripcionitem' => string 'Cinturón portaherramientas ajustable' (length=37)
      private 'autorizado' => boolean true
      private 'descontinuado' => boolean false
      private 'preciounitario' => float 50
      private 'bloqueado' => boolean false
      private 'observaciones' => null
      private 'idEspecifico' =>
        object(Proxies\SaludComprasBundleEntityEspecificoProxy) [251]
          private 'entityPersister' =>
            object(Doctrine\ORM\Persisters\BasicEntityPersister) [249]
              ...
              private '_identifier' =>
                array
                ...
              public '_isInitialized_' => boolean false
              private 'id' (Salud\ComprasBundle\Entity\Especifico) => null
              private 'codigoespecifico' (Salud\ComprasBundle\Entity\Especifico) => null
              private 'descripcionespecifico' (Salud\ComprasBundle\Entity\Especifico) => null
              private 'idCatalogoProducto' (Salud\ComprasBundle\Entity\Especifico) => null
              private 'idRubro' (Salud\ComprasBundle\Entity\Especifico) => null
          private 'idEspecificoOnu' =>
            object(Proxies\SaludComprasBundleEntityEspecificoProxy) [248]
              private 'entityPersister' =>
                object(Doctrine\ORM\Persisters\BasicEntityPersister) [249]
                  ...
                  private '_identifier' =>
                    array
                    ...
            ...
            ...
```

Illustration 5: Salida de la función var_dump aplicada sobre un arreglo (Con XDebug)

Utilizar reportes de jasperreport en PHP (Opcional)

*NOTA: Aun no está adecuado para symfony se mostrará un ejemplo en PHP plano.

1. Instalar los paquetes necesarios (Será necesario activar repositorios extras en ubuntu el repositorio *partner*)

```
$ apt-get install sun-java6-bin sun-java6-fonts sun-java6-jre java-common php5-cgi
```

2. Instalar PHP/Java Bridge

Descargar y descomprimir el paquete de [PHP/JAVA BRIDGE](#)

Se recomienda descargar la versión que viene con la documentación, es importante descargar esa versión ya que trae ejemplos y mucha información que a la larga es útil, se descomprime el archivo descargado en la carpeta que se desee.

Ejecutar la prueba, en la carpeta donde se descomprimió Java Bridge:

```
java -classpath JavaBridge.war TestInstallation
```

Es importante hacer este test, ya que esto genera las clases necesarias para configurar la aplicación PHP, además te dice donde están o de donde carga las extensiones de java (fíjate en `java.ext.dirs:`)

Ahora vamos a detener el proceso que se abrió con la prueba para eso buscamos su PID

```
$ ps aux | grep JavaBridge
```

Fíjate en el PID (segunda columna) y escribe luego

```
$ sudo kill #PID
```

3. Copiar las librerías requeridas a la carpeta de extensiones de java

- Descargar iReport (versión para linux con las fuentes) de la [página oficial](#)

- Descomprimir el archivo descargado de iReport y entrar a él. Copiar todo el contenido de las carpetas ireport/libs y ireport/modules/ext a la carpeta de extensiones de java.
- Descargar el driver para la conexión a la base de datos, en este caso PostgreSQL de la [página oficial](#). Descomprimir el archivo y copiar su contenido a la carpeta de extensiones de java.

4. Configurar la aplicación

- En la raíz de tu proyecto crea una carpeta con el nombre *java*
- Entra a la carpeta de JavaBridge copia el contenido de las carpetas *java* y *ext* a la carpeta *java* de tu proyecto.

5. Iniciar PHP JavaBridge: Dentro de la carpeta *java* de tu proyecto ejecuta

```
$ sudo java -jar JavaBridge.jar SERVLET_LOCAL:8080 3 JavaBridge.log &
```

Lo anterior levanta el servicio de puente entre JAVA y PHP, servicio que corre en el puerto 8080 y tiene que estar levantado para poder crear los reportes desde PHP.

Opcionalmente se puede hacer un script para levantar el servicio automáticamente al iniciar el sistema operativo.

6. Un ejemplo sencillo

<?php

```
/*  
 * Para este ejemplo se asume  
 * que hay una carpeta "java" que contiene PHP JavaBridge  
 * otra "src_reportes" que tiene los reportes en formato de jasperreport  
 * y otra "reportes" donde se colocarán los reportes generados  
 */
```

```
require_once("java/Java.inc");

require_once("java/JavaBridge.inc");


//$paramA = 'Reporte generado el día ' . date('d/m/Y');
//$paramB = 60;

try {
    try {

        $tiempo_inicio = microtime(true); //tiempo de inicio del script

        $file_name = "Reporte_" . date('dmY') . "_".pdf";

        //instancio java

        $class = new JavaClass("java.lang.Class");

        $class->forName("org.postgresql.Driver");

        $driverManager = new JavaClass("java.sql.DriverManager");

        $cs = 'jdbc:postgresql://localhost/comprasdb?user=admin&password=admin';

        $conn = $driverManager->getConnection($cs);


        //cargo la clase para compilar el reporte

                                                                 $compileManager = new
JavaClass("net.sf.jasperreports.engine.JasperCompileManager");

        //compilo el reporte

        $jrxml = 'report1.jrxml';

        $report = $compileManager->compileReport(realpath(realpath(".")) . "/src_reportes/
$jrxml"));

        //cargo la clase para escribir en el reporte
```

```
$fillManager = new JavaClass("net.sf.jasperreports.engine.JasperFillManager");

//carga la clase para pasarle parametros al reporte si es necesario
$params = new Java("java.util.HashMap");

//seteo un parametro del reporte
//$params->put('generado', $paramA);
//$params->put('limit', $paramB);

//imprimo el reporte en la memoria
$jasperPrint = $fillManager->fillReport($report, $params, $conn);

//carga la clase para exportar
$exportManager = new
JavaClass("net.sf.jasperreports.engine.JasperExportManager");

//seteo el archivo de salida
$outputPath = realpath(".") . "/reportes/" . $file_name;

//exporto a PDF
$exportManager->exportReportToPdfFile($jasperPrint, $outputPath);
//tiempo de fin del proceso
$tiempo_fin = microtime(true);

//Si se desea que se descargue
//header("Content-type: application/pdf");
//readfile($outputPath)

unlink($outputPath);
```

```
//imprimo en pantalla cuanto demoré

    echo "El archivo $file_name se gener&oacute; en " . ($tiempo_fin -
$tiempo_inicio);

    } catch (JavaException $ex) {

        $trace = new Java("java.io.ByteArrayOutputStream");

        $ex->printStackTrace(new Java("java.io.PrintStream", $trace));

        print "java stack trace: $trace";

    }

} catch (Exception $e) {

    echo $e->getMessage();

}

?>
```


2. FUNDAMENTOS DEL FRAMEWORK MVC

Patrón de diseño MVC

Si has desarrollado sitios web con PHP sin utilizar ningún framework, seguramente sigues el razonamiento de crear un archivo PHP por cada página HTML del sitio. Además, todos esos archivos PHP contienen seguramente la misma estructura: inicialización y configuración global, lógica de negocio relacionada con la página solicitada, obtención de registros de la base de datos y por último, el código PHP que se emplea para generar la página.

También es posible que utilices un sistema de plantillas para separar el código PHP y las etiquetas HTML. Puede que también utilices una capa de abstracción de base de datos para separar la lógica de negocio y la interacción con el modelo de datos. A pesar de estas mejoras, la mayoría de las veces te encuentras con una gran cantidad de código que es muy difícil de mantener. Programar la aplicación de esa manera quizás te costó muy poco tiempo, pero modificarla y añadirle nuevas características se convierte en una pesadilla, sobre todo porque nadie más que tu sabe cómo está construida y cómo funciona.

Para cada problema siempre hay buenas soluciones y para la programación web, la solución más utilizada actualmente para organizar el código es el [patrón de diseño MVC](#). En pocas palabras, el patrón de diseño MVC organiza el código en base a su función. De hecho, este patrón separa el código en tres capas:

- La capa del **modelo** define la lógica de negocio (la base de datos pertenece a esta capa). Symfony, por defecto, guarda todas las clases y archivos relacionados con el modelo en el directorio *Entity* del bundle.
- La **vista** es lo que utilizan los usuarios para interactuar con la aplicación (los gestores de plantillas pertenecen a esta capa). En Symfony la capa de la vista está formada principalmente por plantillas en PHP o Twig. Estas

plantillas se guardan en varios directorios llamados *Resources/views/* pudiendo estar en varios puntos del proyecto.

- El **controlador** es un bloque de código que realiza llamadas al modelo para obtener los datos y se los pasa a la vista para que los muestre al usuario. Cuando instalamos Symfony el primer día, explicamos que todas las peticiones se canalizan a través de los controladores frontales (`app.php` y `app_dev.php`). Estos controladores frontales realmente delegan todo el trabajo en las **acciones**.

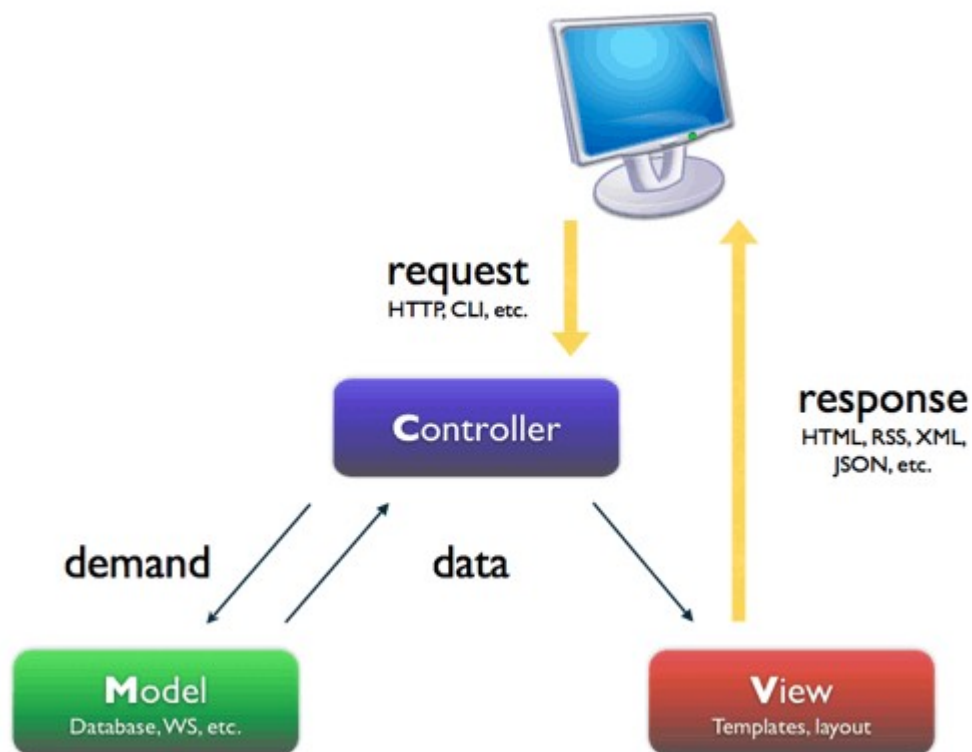


Illustration 6: Patrón de Diseño MVC

De archivos PHP planos a Symfony

Crearemos una página para mostrar el listado de productos, utilizando php plano.

```
<?php
// index.php

$link = pg_connect('dbname=comprasdb user=admin password=admin');

$sql = 'SELECT A.id, A.descripcionitem, A.preciounitario, B.descripcionunidadmedida,
C.descripcionespecifico
      FROM item A
      INNER JOIN unidad_medida B ON (A.id_unidad_medida = B.id)
      INNER JOIN especifico C ON (A.id_especifico = C.id)
      LIMIT 30';

$result = pg_query($link, $sql);
$productos = pg_fetch_all($result);
?>

<html>
  <head>
    <title>Listado de productos</title>
  </head>
  <body>
    <h1>Listado de productos</h1>
    <ul>
      <?php foreach($productos as $producto): ?>
        <li>
          <a href="/show.php?id=<?php echo $producto['id'] ?>" >
            <?php echo $producto['descripcionitem'] .' '
            . $producto['descripcionunidadmedida'] .
            ' ('.$producto['descripcionespecifico'] .')' ?>
          </a>
        </li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>

<?php
pg_close($link);
```

Eso se escribe rápido, se ejecuta rápido y, a medida que la aplicación crezca, imposible de mantener. Hay varios problemas que necesitan ser revisados:

- **Falta verificación de errores.** ¿Qué pasa si la conexión a la base de datos falla?
- **Pobre organización.** Si la aplicación crece, este simple archivo se incrementaría y se volvería muy difícil de mantener. ¿Dónde colocarías el código para manejar el envío del formulario? ¿Cómo validarías los datos?
- **Código difícil de reutilizar.** Dado que todo está en una sola página, no hay forma de utilizar parte de la aplicación en otras páginas.

Separar la presentación

El código puede inmediatamente beneficiarse de separar la “lógica” del código que prepara la “presentación” HTML.

```
<?php
// index.php

$link = pg_connect('dbname=comprasdb user=admin password=admin');

$sql = 'SELECT A.id, A.descripcionitem, A.preciounitario, B.descripcionunidadmedida,
C.descripcionespecifico
FROM item A
      INNER JOIN unidad_medida B ON (A.id_unidad_medida = B.id)
      INNER JOIN especifico C ON (A.id_especifico = C.id)
LIMIT 30';
$result = pg_query($link, $sql);
$productos = pg_fetch_all($result);
pg_close($link);

require 'templates/list.php';
```

El código HTML está ahora en un archivo separado (templates/list.php), el cual es principalmente un archivo HTML que usa una sintaxis de una plantilla con código PHP.

```
<html>
  <head>
    <title>Listado de productos</title>
  </head>
  <body>
    <h1>Listado de productos</h1>
    <ul>
      <?php foreach($productos as $producto): ?>
        <li>
          <a href="/show.php?id=<?php echo $producto['id'] ?>" >
            <?php echo $producto['descripcionitem'] . ' '
              . $producto['descripcionunidadmedida'] .
              ' ('.$producto['descripcionespecifico'] .')' ?>
          </a>
        </li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```

Por convención, el archivo que contiene toda la lógica de la aplicación - `index.php` - es conocido como “controlador”. El término controlador es una palabra que se escuchará mucho, sin tener en cuenta el lenguaje o framework que se utilice. Este se refiere simplemente al área de tu código que procesa la entrada del usuario y prepara la respuesta.

En este caso nuestro controlador prepara los datos obtenidos de la base de datos y luego incluye una plantilla que presenta dichos datos. Con la separación entre el controlador y la plantilla, fácilmente se podría cambiar solamente el archivo de la plantilla si se necesitara mostrar el resultado en otro formato (Ej.: `json.php` para un formato JSON)

Separando el dominio de la lógica de la aplicación

Hasta ahora, la aplicación contiene sólo una página. Pero ¿qué pasa si se requiere que en una segunda utilizar la conexión misma base de datos, o incluso la misma matriz de productos? Refactorizar el código para que el comportamiento de las funciones básicas y de acceso a datos estén separados en un nuevo archivo llamado model.php:

```
<?php
// model.php

function open_database_connection()
{
    $link = pg_connect('dbname=comprasdb user=admin password=admin');
    return $link;
}

function close_database_connection($link)
{
    pg_close($link);
}

function get_all_products()
{
    $link = open_database_connection();

    $sql = 'SELECT A.id, A.descripcionitem, A.preciounitario, B.descripcionunidadmedida,
C.descripcionespecifico
FROM item A
      INNER JOIN unidad_medida B ON (A.id_unidad_medida = B.id)
      INNER JOIN especifico C ON (A.id_especifico = C.id)
LIMIT 30';
    $result = pg_query($link, $sql);
    $productos = pg_fetch_all($result);
    close_database_connection($link);
    return $productos;
}
```

El nombre del archivo `model.php` es utilizado porque la lógica y el acceso a los datos de la aplicación es tradicionalmente conocido como la capa “modelo”. En una aplicación bien organizada, la mayoría del código que representa la “lógica del negocio” debería residir en el modelo (en lugar de estar en el controlador). Y a diferencia de este ejemplo, solo una parte (o nada) del código del modelo está actualmente asociado con el acceso a la base de datos.

El controlador (`index.php`) es ahora muy simple:

```
<?php
// index.php

require 'model.php';

$productos = get_all_products();

require 'templates/list.php';
```

Separando el Layout

Hasta este punto, la aplicación ha sido refactorizada en tres piezas distintas ofreciendo varias ventajas y oportunidades de reusar casi todo en páginas diferentes.

La parte de código que aún no podemos reutilizar es la página con el diseño general (Layout). Para esto crearemos un nuevo archivo *layout.php*

```
<!-- templates/layout.php -->
<html>
  <head>
    <title><?php echo $title ?></title>
```

```
</head>
<body>
    <?php echo $content ?>
</body>
</html>
```

La plantilla (templates/list.php) puede ahora ser simplificada para “extender” el layout:

```
<?php $title = 'Listado de productos2' ?>

<?php ob_start() ?>
    <h1>Listado de productos</h1>
    <ul>
        <?php foreach($productos as $producto): ?>
            <li>
                <a href="/show.php?id=<?php echo $producto['id'] ?>" >
                    <?php echo $producto['descripcionitem'] . ' '
                        . $producto['descripcionunidadmedida'] .
                        ' ('.$producto['descripcionespecifico'] .')' ?>
                </a>
            </li>
        <?php endforeach; ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Con esto se ha introducido una metodología que permite reutilizar el layout.

Agregando una página mostrar producto

La página para listar productos ha sido refactorizada, así el código está mejor

organizado y es reutilizable. Para probar esto, agregaremos una página para mostrar un producto, la cual desplegará un producto individual identificado por un parámetro id.

Para empezar, creamos una nueva función en el *model.php* que recuperar un solo producto basado en un id dado.

```
//model.php
function get_product_by_id($id)
{
    $link = open_database_connection();

    $id = pg_escape_string($id);

    $query = 'SELECT A.descripcionitem, A.preciounitario, B.descripcionunidadmedida,
C.descripcionespecifico
            FROM item A
            INNER JOIN unidad_medida B ON (A.id_unidad_medida = B.id)
            INNER JOIN especifico C ON (A.id_especifico = C.id)
            WHERE A.id = '.$id;
    $result = pg_query($query);

    $row = pg_fetch_array($result,0,PGSQL_ASSOC);

    close_database_connection($link);

    return $row;
}
```

Lo siguiente es crear un nuevo archivo con el nombre *show.php* - El controlador de esta nueva página:

```
<?php
// show.php

require 'model.php';
```

```
$producto = get_product_by_id($_GET['id']);  
  
require 'templates/show_html.php';
```

Finalmente, creamos el nuevo archivo para la plantilla - *templates/show_html.php* - para renderizar el producto de forma individual.

```
<?php $title = 'Detalle de producto' ?>  
  
<?php ob_start() ?>  
    <h1><?php echo $title ?></h1>  
    <div>  
        Producto: <?php echo $producto['descripcionitem'] ?><BR />  
        Unidad de medida: <?php echo $producto['descripcionunidadmedida'] ?><BR />  
        Espec&iacute;fico: <?php echo $producto['descripcionespecifico'] ?>  
    </div>  
<?php $content = ob_get_clean() ?>  
  
<?php include 'layout.php' ?>
```

Crear la segunda página ha resultado más fácil y el código no se ha duplicado. Aunque aún, esta página introduce otros problemas que el framework puede manejar. Por ejemplo, que falte o sea inválido el parámetro id causará que la página falle. Peor si se olvida limpiar el parámetro id con la función `pg_escape_string()`, pues la base de datos podría estar bajo el riesgo de un ataque de inyección SQL.

Otro problema es que cada controlador individual debe incluir el archivo *model.php*. Qué si de repente cada archivo necesita incluir un archivo adicional o realizar una tarea global (Ej.: reforzar la seguridad). Tal como está en este momento, sería necesario agregar el código a todos los archivos. Si se olvida agregarlo en alguno de los archivos, ojalá no sea algo relacionado a la seguridad....

Un “controlador frontal” al rescate.

La solución es utilizar un controlador frontal: un archivo PHP a través del cual *todas* las peticiones son procesadas. Con un controlador frontal, las URIs para la aplicación cambian ligeramente, pero se vuelven más flexibles:

Sin controlador frontal

```
/index.php      => Página de listado de productos (index.php ejecutado)
/show.php       => Página de mostrar producto (show.php ejecutado)
```

Con index.php como controlador frontal

```
/index.php      => Página de listado de productos (index.php ejecutado)
/index.php/show => Página de mostrar producto (index.php ejecutado)
```

Cuando se utiliza un controlador frontal, un simple archivo PHP (index.php en este caso) procesa todas las peticiones. Para la página mostrar producto, */index.php/show* será ejecutado el archivo *index.php*, el cual por el momento es el responsable internamente de realizar el ruteo de las peticiones basado en la URI completa. Como verás, un controlador frontal es un herramienta muy poderosa

Creando el controlador frontal

Estás a punto de dar un gran paso con la aplicación. Con un archivo que maneje todas las peticiones, se puede centralizar cosas como el manejo de la reguridad, carga de la configuración, y el ruteo. En esta aplicación, *index.php* ahora debe ser lo suficientemente inteligente para procesar la página listado de productos o la página mostrar producto basado en la URI de la solicitud.

```
<?php
// index.php

// cargar e inicializar cualquier libreria global
```

```
require_once 'model.php';
require_once 'controllers.php';

// hacer el ruteo interno de la petición
$uri = $_SERVER['PHP_SELF'];
if ($uri == '/index.php') {
    list_action();
} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Pa&acute;gina no encontrada</h1></body></html>';
}
```

Por organización, los dos controladores que teníamos (index.php y show.php) ahora son funciones PHP y han sido movidos a un archivo, *controllers.php*:

```
<?php

function list_action()
{
    $productos = get_all_products();
    require 'templates/list.php';
}

function show_action($id)
{
    $producto = get_product_by_id($id);
    require 'templates/show_html.php';
}
```

Será necesario cambiar en la plantilla *templates/list.php* cambiar la línea

```
<a href="/show.php?id=<?php echo $producto['id'] ?>" >
```

Para que tome en cuenta el nuevo formato de las “rutas” que manejará el controlador frontal.

```
<a href="/index.php/show?id=<?php echo $producto['id'] ?>" >
```

Como un controlador frontal, *index.php* ha tomado un nuevo rol, uno que incluye el cargar las librerías base y realizar el ruteo de la aplicación hacia uno de los dos controladores (las funciones *list_action()* y *show_action()*). El controlador está empezando a verse y actuar mucho como el mecanismo de Symfony2 para la manipulación y enrutamiento de solicitudes.

Por ahora, la aplicación se ha desarrollado de un solo archivo PHP en una estructura que se organiza y permite la reutilización de código. Debes ser feliz, pero lejos de ser satisfecho. Esto solo muestra de manera ligera las capacidades de la utilización del framework. Además, al desarrollar la aplicación , una gran cantidad de tiempo se gasta trabajando en la "arquitectura" del código (por ejemplo, enrutamiento, llamando a los controladores, plantillas, etc.) Más tiempo tendrá que ser gastado para manejar los envíos de formularios, validación de entrada, registro y seguridad. ¿Por qué tienes que reinventar soluciones a todos estos problemas de rutina?

Flujo de una aplicación en Symfony

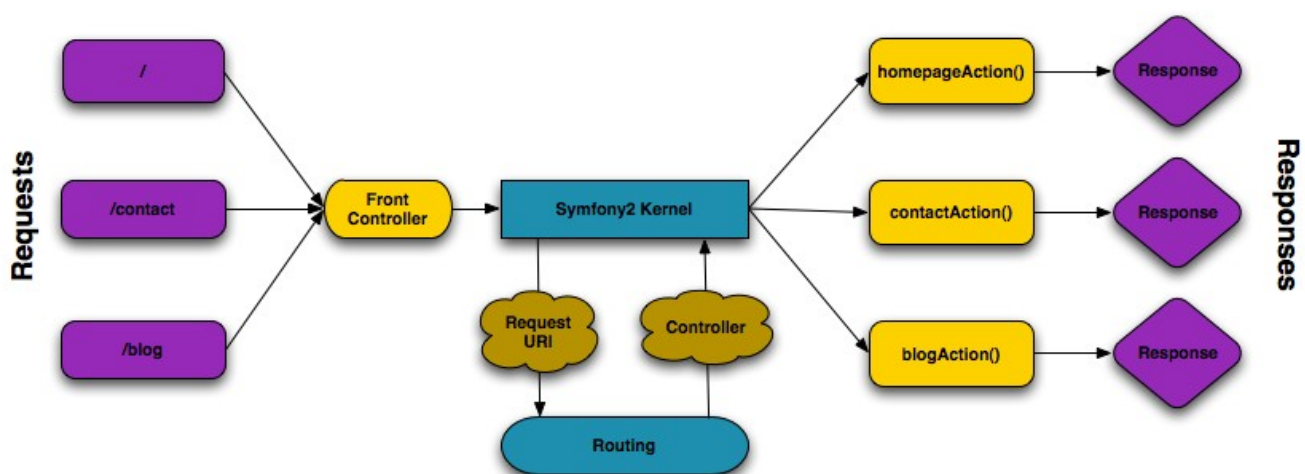


Illustration 7: Flujo de una aplicación en Symfony2

Las peticiones son interpretadas por el sistema de ruteo y pasadas al respectivo controlador que retorna una respuesta (un objeto Response)

Cada página de tu sitio está definida en un archivo de configuración de ruteo que se encarga de mapear las URLs a diferentes funciones PHP. El trabajo de cada función PHP, llamada *controlador*, es usar la información de la petición – junto con muchas otras herramientas que Symfony pone disponibles – para crear y retornar una respuesta. En otras palabras el controlador es el lugar donde se ubicará tu código: es dónde interpretas una petición y creas una respuesta.

Es así de fácil! Repasemos:

- Cada petición ejecuta un controlador frontal.
- El sistema de ruteo determina que función de PHP deberá ser ejecutada en base a la información de la petición y la configuración de ruteo que has configurado.
- La correspondiente función PHP es ejecutada, tu código crea y retorna la respuesta apropiada.

3. CONFIGURACIÓN INICIAL DEL PROYECTO

La estructura de directorios

Después de unas pocas sesiones, ya has comprendido la filosofía detrás de la creación y renderización de páginas en Symfony2.

A pesar de que es perfectamente flexible, cada aplicación de Symfony2 recomienda la misma estructura básica de directorios:

- `app/`: Este directorio contiene la configuración de la aplicación.
- `src/`: Todo el código PHP de tu proyecto está bajo este directorio.
- `vendor/`: Cualquier librería de terceros es colocada aquí por convención.
- `web/`: Este es el directorio raíz de la aplicación web y contiene todos los archivos accesibles públicamente .

Entornos

Una aplicación puede ejecutarse en varios entornos. Los diferentes entornos comparten el mismo código PHP (excepto el controlador frontal), pero pueden tener configuraciones completamente diferentes. Por ejemplo un entorno de desarrollo **dev** creará bitácora de advertencias y errores, mientras un entorno de producción **prod** solamente creará la bitácora de errores. Algunos archivos son reconstruidos en cada petición de la página en un entorno dev, pero son almacenados en caché en un entorno prod. Todos los entornos están alojados en la misma máquina.

Un proyecto de Symfony2 generalmente empieza con tres entornos (dev, test y prod), aunque puedes crear nuevos entornos.

Dado que el entorno prod está optimizado para velocidad; la configuración, ruteo y plantillas Twig son compiladas en clases PHP y almacenadas en caché.

Cuando hacés cambios en un entorno prod, será necesario limpiar la cache para que esos archivos sean recompilados:

```
app/console cache:clear
```

O también puedes utilizar (para asegurarte que se borra totalmente):

```
rm -rf app/cache/*
```

El entorno test es utilizado cuando se ejecutan pruebas y no puede ser accesado directamente a través del navegador.

El controlador frontal es el que indica que entorno se utilizará; ya hemos ejecutado el entorno prod (<http://compras.localhost/app.php>) ahora veamos el entorno dev:

Entramos la dirección a nuestro navegador http://compras.localhost/app_dev.php

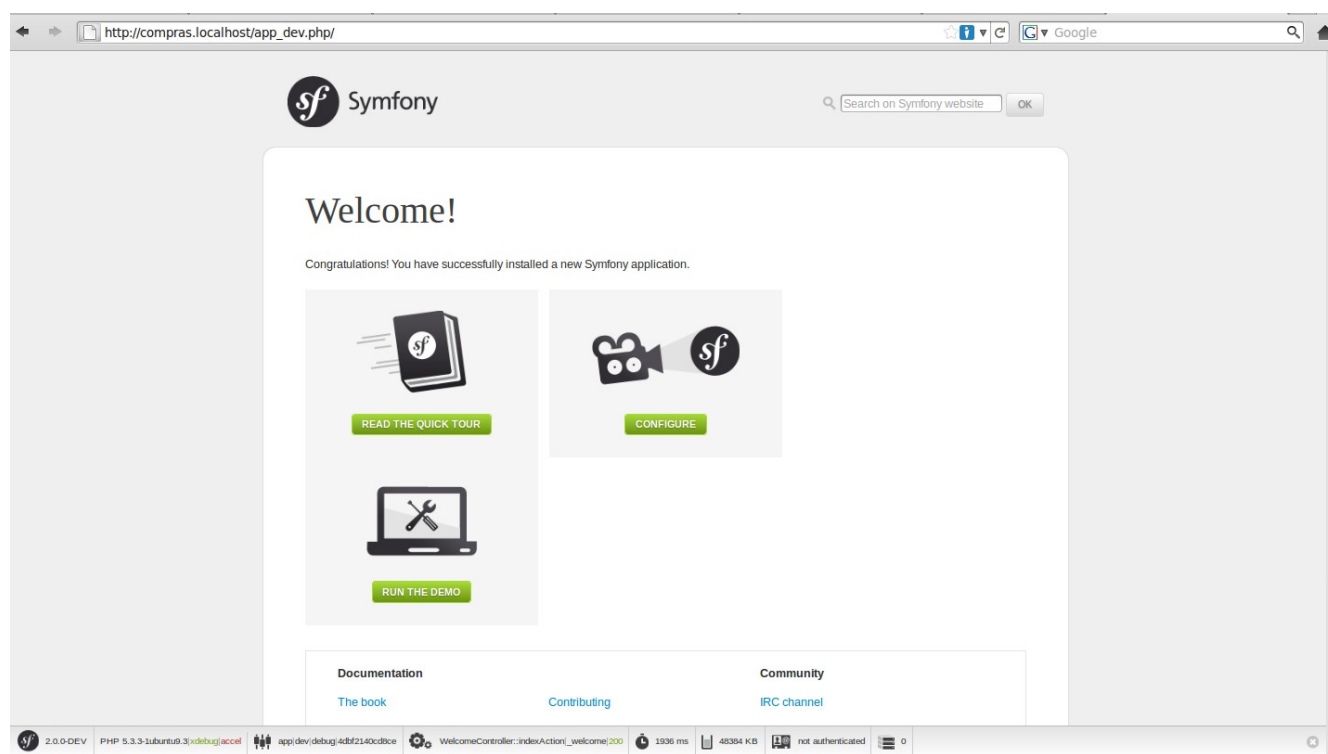


Illustration 8: Pantalla de inicio en un entorno de depuración

Se han agregado dos elementos en modo de depuración: la barra de depuración que está situada en la parte inferior.

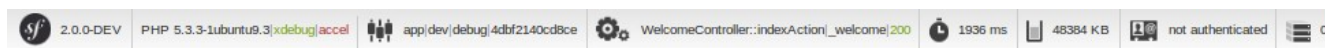


Illustration 9: Barra de depuración

Y además la opción para realizar la configuración inicial:



Illustration 10: Botón de configuración inicial

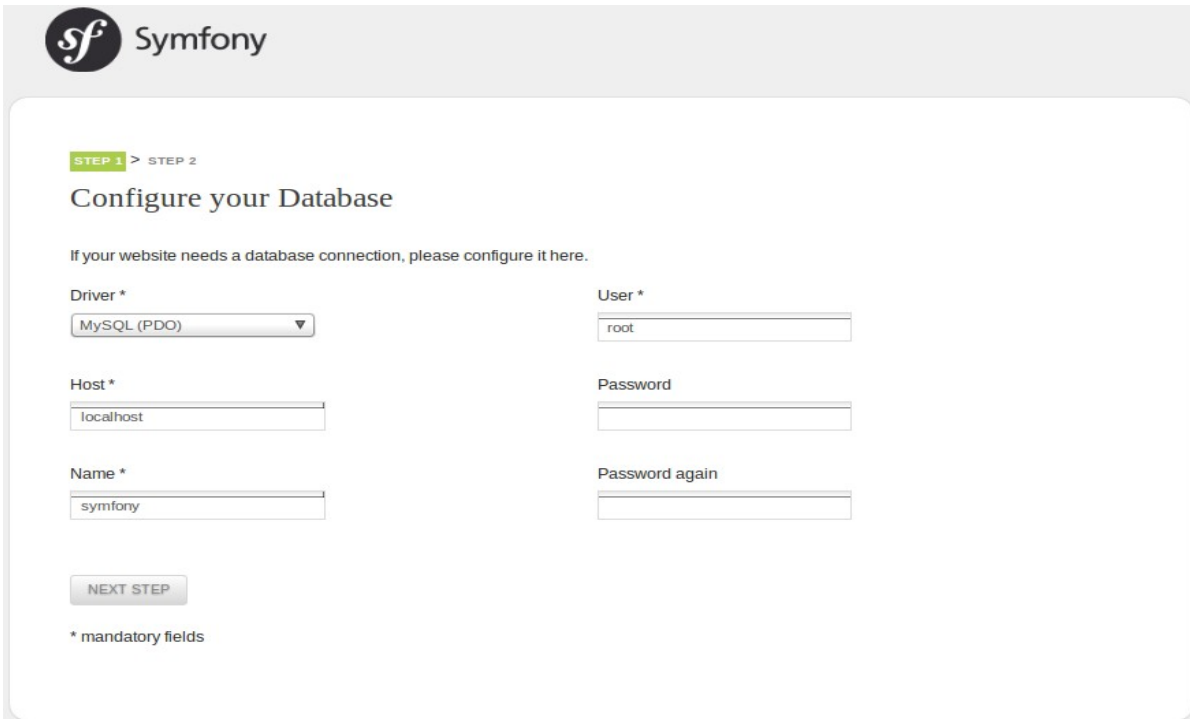
Si recibimos el siguiente mensaje:

You are not allowed to access this file. Check app_dev.php for more information.

Debemos agregar la dirección ip con la cual hemos configurado nuestro virtual host en el archivo que intentamos cargar.

Realizar la configuración inicial

En el modo de depuración de la aplicación, damos clic en el enlace ***Configure***

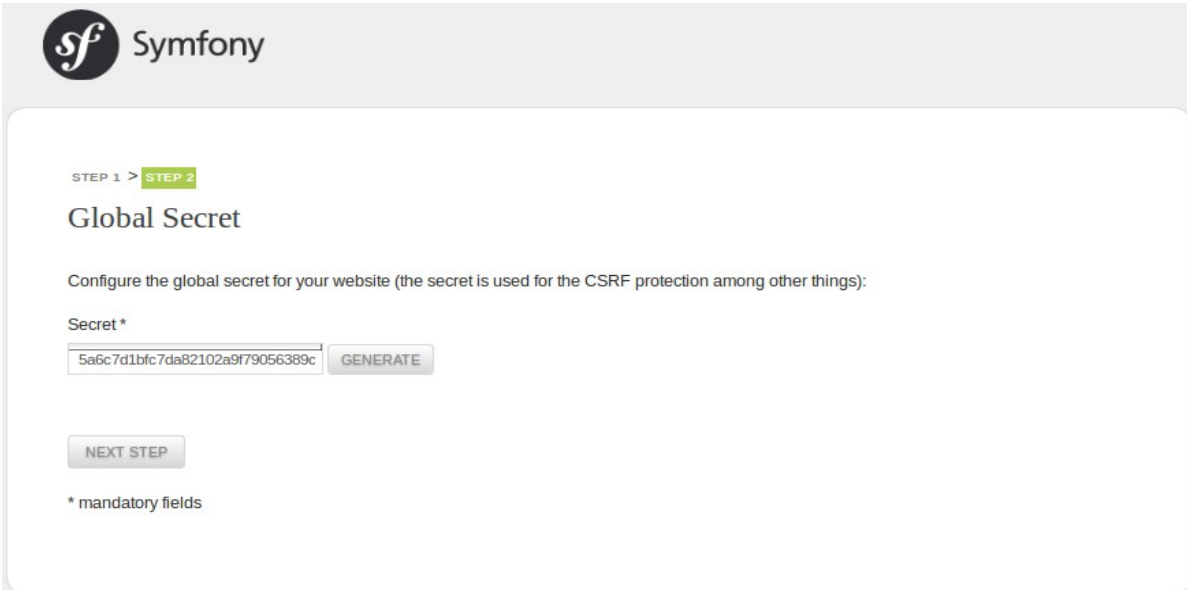


The screenshot shows the Symfony installation interface. At the top left is the Symfony logo. Below it, a progress bar indicates 'STEP 1' is active, followed by 'STEP 2'. The main heading is 'Configure your Database'. A subtext says 'If your website needs a database connection, please configure it here.' The form contains several fields: 'Driver *' is a dropdown menu with 'MySQL (PDO)' selected; 'User *' is a text input with 'root' entered; 'Host *' is a text input with 'localhost' entered; 'Name *' is a text input with 'symfony' entered; 'Password' is an empty text input; and 'Password again' is an empty text input. At the bottom left is a 'NEXT STEP' button. Below the button is a note '* mandatory fields'.

Illustration 11: Inicio de la configuración

El primer paso es la conexión a la base de datos, especificamos los parámetros según nuestro caso.

El siguiente paso se configura la llave secreta que se utilizará para crear la protección contra [ataques CSRF](#)



The screenshot shows the Symfony web interface for configuring the global secret. At the top left is the Symfony logo. Below it, a progress bar indicates 'STEP 1 > STEP 2', with 'STEP 2' highlighted in green. The main heading is 'Global Secret'. Below this, a text instruction reads: 'Configure the global secret for your website (the secret is used for the CSRF protection among other things):'. A label 'Secret *' is positioned above a text input field. The input field contains the alphanumeric string '5a6c7d1bfc7da82102a9f79056389c'. To the right of the input field is a 'GENERATE' button. Below the input field is a 'NEXT STEP' button. At the bottom left, a note states '* mandatory fields'.

Illustration 12: Configurar la protección contra ataques CSRF

De acuerdo a cómo estén configurados los permisos sobre los archivos la aplicación de configuración puede haber escrito ya los cambios, sino será necesario agregarlos en el archivo ***app/config/parameters.ini*** sustituyendo el contenido que pueda existir en el archivo.

4. EL MODELO

El modelo es la “M” del patrón “MVC”. Un modelo es responsable de cambiar internamente el estado basado en peticiones desde el Controlador y dar el estado actual de la información a la Vista. Es el principal contenedor de la lógica de la aplicación.

Base de datos y Symfony2

Symfony2 no tiene de forma propia un Mapeador Objeto-Relacional (ORM por sus siglas en inglés) ni una capa de abstracción de base de datos (DBAL); sin embargo Symfony2 tiene una profunda integración con librerías como Doctrine y Propel, los cuales proveen los paquetes para ORM y DBAL, permitiendo elegir cuál utilizar, por defecto se utiliza Doctrine.

Un modelo no es una tabla

La percepción de un clase del modelo como una tabla de la base de datos, donde cada objeto instanciada representa una fila o registro, fue popularizada por el framework Ruby on Rails y el patrón de diseño Active Record. Esto es una buena forma de un primer pensamiento acerca de la capa del modelo de la aplicación, especialmente si estás desarrollando una simple interface Crear, Recuperar, Actualizar y Borrar (CRUD por sus siglas en inglés) para modificar los datos de un modelo.

Pero este enfoque actualmente puede causar problemas si se pasa de la parte de CRUD de la aplicación y se empieza a agregar mas lógica del negocio. Hay limitaciones comunes del enfoque descrito arriba:

- Una base de datos debe ser adecuada a las necesidades de la aplicación no

al revés.

- Algunos motores de almacenamiento de datos (como base de datos documentales) no tienen una noción de tablas, filas o incluso un esquema, por lo que es difícil utilizarla con la percepción de que una clase del modelo es una tabla.
- Mantener en mente el esquema mientras se diseña el dominio de la aplicación es problemático, lo cual puede darte lo peor de los dos mundos.

El ORM de Doctrine2 está diseñado para quitar la necesidad de mantener en mente la estructura de la base de datos y permite concentrarse en escribir lo más limpio posible el modelo que solventará las necesidades del negocio. Te permite diseñar primero las clases y sus interacciones, antes de requerir que se piense en cómo se almacenarán los datos.

Doctrine DBAL

La Capa de Abstracción de Base de Datos de Doctrine es una capa que está situado sobre PDO y ofrece una intuitiva y flexible API para comunicarse con las más populares bases de datos relacionales.

Para empezar se configura las conexiones a la base de datos (Esto ya se realizó en la configuración inicial del capítulo anterior):

```
# app/config/config.yml
doctrine:
  dbal:
    driver:   %database_driver%
    host:    %database_host%
    dbname:  %database_name%
    user:    %database_user%
    password: %database_password%
```

Se puede escribir los valores de los parámetros (*driver, host, etc*) o utilizar las

variables (%database_driver% y demás) cuyos valores se encuentran en el archivo *app/config/parameters.ini*

Se puede acceder a la conexión DBAL de Doctrine por medio del servicio *database_connection*

```
class UserController extends Controller
{
    public function indexAction()
    {
        $conn = $this->get('database_connection');
        $users = $conn->fetchAll('SELECT * FROM users');

        // ...
    }
}
```

Configuración

Además de las opciones por defecto de Doctrine, hay algunas más que se pueden configurar. El siguiente bloque muestra todas esas posibles configuraciones:

```
doctrine:
  dbal:
    dbname:           database
    host:             localhost
    port:             1234
    user:             user
    password:         secret
    driver:           pdo_mysql
    driver_class:     MyNamespace\MyDriverImpl
    options:
      foo: bar
    path:             %kernel.data_dir%/data.sqlite
    memory:           true
    unix_socket:      /tmp/mysql.sock
    wrapper_class:    MyDoctrineDbalConnectionWrapper
    charset:          UTF8
    logging:          %kernel.debug%
    platform_service: MyOwnDatabasePlatformService
```

Y si se necesita múltiples bases de datos:

```
doctrine:
```

```
dbal:
  default_connection:      default
  connections:
    default:
      dbname:              Symfony2
      user:                 root
      password:            null
      host:                 localhost
    customer:
      dbname:              customer
      user:                 root
      password:            null
      host:                 localhost
```

El servicio *database_connection* siempre se refiere a la conexión por defecto, la cual es la primera definida o la que está especificada en el parámetro *default_connection*. Si se quiere acceder a otra conexión dentro del controlador se puede utilizar: `$this->getDoctrine()->getConnection('nombre_conexion');`

Doctrine ORM

Doctrine es un Mapeador Objeto-Relacional (ORM) para PHP que está sobre la Capa de Abstracción de Base de Datos (DBAL). Provee persistencia transparente para los objetos PHP.

Vamos a suponer en este manual que ya tenemos creada la base de datos (El proceso que se propone en Symfony2 es crear primero las clases anotadas y a partir de ahí la base de datos para ver mas detalle referirse al capítulo [Doctrine ORM](#) de la documentación).

En Symfony2 todo proyecto está compuesto por Bundles (paquete) incluso la aplicación en sí es un bundle, más adelante vamos a detallar un poco más sobre los bundles pero en este momento vamos a crear un bundle

En una consola, creamos el bundle (Recordar, para todos los comandos, estar dentro de la carpeta del proyecto):

```
app/console generate:bundle
```

Esto lanzará el generador de bundles que nos dará algunas recomendaciones y luego solicita el espacio de nombres bajo el cual registraremos el bundle, a lo cual ingresaremos:

```
Salud/ComprasBundle
```

Luego solicitará el nombre del bundle, dejaremos el nombre generado por defecto *SaludComprasBundle*, para lo cual solo presionamos ENTER.

Y a continuación se nos solicitará la ubicación en que crearemos el bundle, aceptaremos la ubicación por defecto, presionamos ENTER.

Luego solicita el formato en que se generarán los archivos de configuraciones, usaremos el formato *annotation* que es la opción por defecto, solo presionamos ENTER.

Ahora nos preguntará si deseamos generar la estructura de directorios por defecto, presionemos ENTER para elegir la opción por defecto.

Para las demás opciones (confirmar la generación, actualizar el kernel y actualizar el archivo de rutas) elijamos los valores por defecto.

Registramos el espacio de nombres en el archivo *app/autoload.php*. A causa del sistema de autocarga, nunca debes preocuparte por usar las instrucciones *include* o *require* . En su lugar, Symfony2 usará el espacio de nombres de una clase para determinar su ubicación y automáticamente incluir el archivo. en el arreglo del método

En el método *\$loader->registerNamespaces* agregamos:

```
'Salud' => __DIR__.'/../src',
```

Con esta configuración, Symfony2 buscará dentro del directorio *src* para

cualquier clase dentro del namespace Salud.

Ahora vamos a importar la definición de la base de datos a Symfony2 (recordar que la base de datos junto a sus parámetros de conexión los configuramos en el capítulo anterior), en una consola tecleamos:

```
app/console doctrine:mapping:import SaludComprasBundle annotation
```

En el comando anterior le estamos indicando a doctrine que el mapeo de la base de datos la haga en el bundle *SaludComprasBundle*, esto crea un archivo que define cada una de las tablas de la base de datos, por defecto en la carpeta *src/Salud/ComprasBundle/Resources/config/doctrine/metadata/orm* pero como utilizamos el parámetro *annotation* crea las clases anotadas con sus respectivos atributos en *src/Salud/ComprasBundle/Entity*

Ya tenemos el mapeo de la base de datos a partir de esto vamos a crear las entidades en la siguiente ruta: *src/Salud/ComprasBundle/Entity*, si utilizamos el parámetro *annotation* en el comando anterior, las entidades ya están creadas y solo se agregarán los métodos para acceder a cada propiedad de la clase

```
app/console doctrine:generate:entities SaludComprasBundle
```

5. PRIMEROS PASOS

Creando páginas en Symfony2

Crear una nueva página en Symfony2 es un proceso simple de dos pasos:

- *Crear una ruta:* Una ruta define la URI (ejemplo /about) para la página y especifica el controlador (una función de PHP) que Symfony2 deberá ejecutar cuando la URI de una petición coincida con el patrón de la ruta;
- *Crear un controlador:* Un controlador es un función de PHP que toma la petición y la transforma en un objeto Response de Symfony2.

Este enfoque es simple y coincide con la forma en que trabaja las aplicaciones web. Cada interacción es iniciada por una petición HTTP. El trabajo de la aplicación es simplemente interpretar la petición y retornar la respuesta HTTP adecuada. Symfony2 sigue esta filosofía y provee las herramientas y convenciones para mantener tu aplicación organizada a medida que crece en usuarios y complejidad.

Crearemos la página para mostrar el listado de productos como lo iniciamos en PHP plano.

Los Bundles

Un bundle es un directorio que tiene una estructura bien definida y puede contener muchas cosas desde clases a controladores y recursos web como archivos javascript, imágenes, etc.

En Symfony2 un bundle es como un plugin excepto que el código de tu aplicación reside dentro de él.

Es recomendable seguir algunas buenas prácticas si se piensa redistribuir el bundle

El nombre de un bundle

Un bundle utiliza el namespace de PHP. El namespace debe seguir los estándares de interoperabilidad para PHP 5.3 y los nombres de las clases deben empezar con un segmento de vendor (como la empresa o la entidad creadora del bundle), seguido por ninguna o más segmentos de categorías, y finalizar con nombre corto de namespace, el cual llevará el sufijo *Bundle*.

El nombre de la clase del bundle debe seguir los siguientes reglas sencillas:

- Usar solo caracteres alfanuméricos y guión bajo.
- Usar un nombre CamelCased
- Usar un nombre descriptivo y corto (no más de 2 palabras)
- Como prefijo debe llevar el vendor (y opcionalmente la categoría)
- Agregar como sufijo la palabra *Bundle*

Ejemplos de namespace y nombres de clases válidos:

Namespace	Nombre de la clase del bundle
Acme\Bundle\BlogBundle	AcmeBlogBundle
Acme\Bundle\Social\BlogBundle	AcmeSocialBlogBundle
Acme\BlogBundle	AcmeBlogBundle

Crear la ruta

Existen tres formas de declarar las rutas:

- Creando la ruta directamente en el archivo principal de rutas *app/config/routing.yml*:

```
_producto_list:
    pattern: /producto/list
```

```
defaults: { _controller: SaludComprasBundle:Producto:list }
```

- Importar desde un archivo de configuración del bundle en el cual estarán las rutas y en el principal (*app/config/routing.yml*) se obtendrían así:

```
compras:
```

```
resource: "@SaludComprasBundle/Resources/config/routing.yml"
```

- La tercer forma es por medio de anotaciones en cada controlador (se mostrará más adelante), y se importan agregando en *app/config/routing.yml*

```
_comprasBundle:
```

```
resource: "@SaludComprasBundle/Controller"
```

Esto importará desde todos los controladores ubicados en la carpeta Controller del bundle, esta es la forma que se utilizará en esta guía. Si desearamos un controlador en específico, se pone el nombre del controlador y se agrega *type: annotation* quedando así (Sí utilizamos el generador interactivo del bundle, podría ya estar creada esta entrada en el archivo):

```
_productoBundle:
```

```
resource: "@SaludComprasBundle/Controller/ProductoController.php"
```

```
type:      annotation
```

El ruteo consiste en dos piezas básicas: el patrón (*pattern*), el cual es la URI que coincidirá con esta ruta, y un arreglo con las opciones por defecto (*defaults*), el cual especifica el controlador que debe ser ejecutado. Ejemplo:

1. Se escribe la dirección: <http://compras.localhost/app.php/producto/list>
2. El controlador frontal (app.php) buscará un patrón que coincida con /producto/list
3. Al encontrarlo, verifica qué controlador procesará la petición en este caso encuentra SaludComprasBundle:Producto:list. La primera parte

(*SaludComprasBundle*) es el nombre del bundle y el namespace, luego el nombre de la clase del controlador (*Producto*) y el método que se ejecutará de dicha clase (*list*)

El sistema de ruteo tiene muchas más funcionalidades para crear flexibles y poderosas URI en la aplicación. Para mas detalles, ver el capítulo [Routing](#) de la documentación oficial de Symfony2.

Crear el controlador

Cuando una URI como */producto/list* es manejada por la aplicación la ruta *_producto_list* coincide y el controlador *SaludComprasBundle:Producto:list* es ejecutado por el framework. El siguiente paso es crear este controlador.

En realidad, un controlador no es más que un método de PHP que Symfony ejecuta. Esto es donde la aplicación usa la información de la petición para construir y preparar el recurso que fue solicitado. Excepto en casos avanzados, el producto final de un controlador es siempre el mismo: un objeto Response de Symfony2:

```
<?php
namespace Salud\ComprasBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class ProductoController extends Controller
{
    /**
     * @Route("/producto/list", name="_producto_list")
     */
}
```

```
public function listAction()
{
    return $this->render('SaludComprasBundle:Producto:list.html.twig');
}
}
```

Observemos la anotación `@Route("/producto/list", name= "_producto_list")` éste es el patrón de la ruta que se importa desde el archivo `app/config/routing.yml` para utilizar esta anotación es necesario la línea

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
```

Si recordamos el controlador que se utiliza es: `SaludComprasBundle:Producto:list`, la segunda parte (Producto) es el nombre de la clase del controlador más el sufijo *Controller* y la acción *listar* coincide con el método *list* más el sufijo *Action*.

Por lo tanto: El nombre de las clases controlador llevan el sufijo *Controller* y extienden a la clase con el mismo nombre. Las acciones llevan el sufijo *Action*.

Las acciones deben devolver un objeto de tipo *Response* que es lo que hace el método *render* si seguimos unas convenciones de Symfony2 podemos acortar la línea

```
return $this->render('SaludComprasBundle:Producto:listar.html.twig');
```

Y dejarla:

```
return array();
```

Para que esto funcione se debe realizar:

- Agregar la anotación `@Template`
- Agregar la línea `use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;`
- El nombre de la plantilla debe ser el mismo de la acción ("*list*" en este caso) y el sufijo `".html.twig"`
- La ubicación debe ser:
`/src/nombreBundle/Resources/views/NombreClaseControlador/`

Crear la plantilla

Si te fijas atentamente en los bocetos gráficos, verás que algunas partes se repiten en todas las páginas. Como ya sabes, duplicar el código nunca es buena idea, ya sea código PHP o etiquetas HTML. Por tanto, tenemos que encontrar alguna forma de evitar la repetición de estos elementos comunes de las páginas.

Una forma sencilla de resolver este problema consiste en definir una cabecera y un pie que se añaden en cada plantilla:

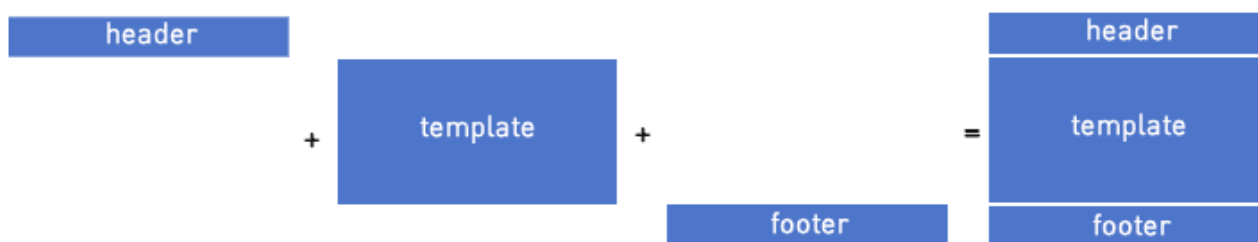


Illustration 13: Cabecera y pie de página

El problema es que los archivos de la cabecera y del pie no contienen código HTML válido, por lo que debemos buscar una alternativa. En vez de perder el tiempo tratando de reinventar la rueda, vamos a utilizar otro patrón de diseño para resolver este problema: el [patrón de diseño decorator](#).

El patrón *decorator* resuelve el problema de otra forma diferente: el contenido se muestra con una plantilla que después se decora con una plantilla global:

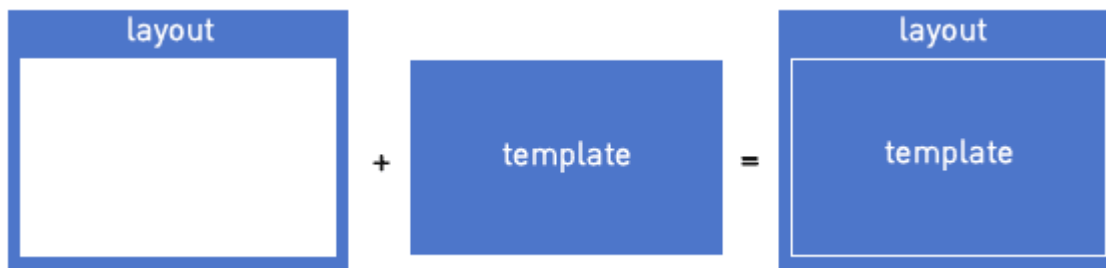


Illustration 14: Funcionamiento del patrón decorator

Las plantillas nos permiten mover todo el código de la presentación en un archivo separado y poder utilizar diferentes porciones de la página del esquema principal. En lugar de escribir el código HTML dentro del controlador (lo cual debemos evitar) se usa una plantilla:

El método `render()` del controlador crea un objeto `Response` el cual llena con el contenido de renderizar la plantilla. Como cualquier otro controlador retornará un objeto `Response`.

Hay dos formas por defecto soportadas por `Symfony2` para renderizar una plantilla: las clásicas plantillas `PHP` y la concisa pero poderosa plantilla [*Twig*](#). Puedes utilizar ambas en el mismo proyecto.

El controlador `renderiza` la plantilla `SaludComprasBundle:Producto:list.html.twig`, la cual utiliza la siguiente convención:

`NombreBundle:NombreControlador:NombrePlantilla`

En este caso, el nombre del bundle es *SaludComprasBundle*, *Producto* el nombre del controlador y *list.html.twig* el nombre de la plantilla:

```
{# src/Salud/ComprasBundle/Resources/views/Producto/list.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}

{% block content%}
    <h1>Listado de productos</h1>
{% endblock %}
```

Veamos la plantilla línea por línea

- Línea 1: Es un comentario con la ruta de la plantilla los comentarios en las

plantillas Twig se hacen con `{# comentario #}`

- Línea 2: La palabra `extends` define una plantilla padre. La plantilla define el layout donde se mostrará el contenido de la plantilla.
- Línea 4: La palabra `block` indica que todo su contenido debe ser colocado en un bloque llamado `content`. Como veremos luego es responsabilidad de la plantilla padre (`layout.html.twig`) de renderizar el bloque llamado `content`.

El contenido del `layout.html.twig` será el siguiente:

```
{#src/Salud/ComprasBundle/Resources/views/layout.html.twig#}
{% extends '::base.html.twig' %}
{% block title %}
    Gesti&oacute;n de Compras Institucional
{% endblock %}
{% block body %}
    <div >
        {% block content %}
        {% endblock %}
    </div>
{% endblock %}
```

Al llamar la plantilla padre `::base.html.twig`, no se ha especificado el nombre del bundle ni del controlador (de ahí los dobles puntos(`::`) del principio). Esto significa que la plantilla está ubicada fuera de los bundles, está en el directorio `app`.

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

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

El layout define el esquema general del sitio y procesa el bloque content que fue definido en la plantilla list.html.twig.

Acceso a los datos

Se modificará el controlador para poder obtener los productos de la base de datos y mandarlos a la plantilla.

```
// src/Salud/ComprasBundle/Controller/ProductoController.php
/**
 * @Route("/producto/list", name="_producto_list")
 * @Template
 */
public function listAction()
{
    //Creamos una instancia del entity manager
    $em = $this->getDoctrine()->getEntityManager();

    // obtenemos todos los productos
    $productos = $em->getRepository('SaludComprasBundle:Item')->findAll();

    //Mandamos el arreglo de productos a la plantilla
    return array('productos' => $productos);
}
```

Para obtener el acceso a la base de datos primero obtenemos una instancia

del *entity manager*:

```
$em = $this->get('doctrine')->getEntityManager();
```

Una de las características de Doctrine2 y otros ORM's que cumplen el patrón de diseño Data Mapper, en el cual en lugar de que cada clase del modelo realice por sí misma consultas a la base de datos (patrón Active Record, el cual el que se utilizaba en versiones anteriores de Doctrine), las consultas a la base de datos se realizan a través de una capa: el Entity Manager.

No podemos realizar consultas a la base de datos sin ir a través del Entity Manager, por ejemplo:

```
$user = $em->find('Entities\User', $id);
```

Es importante saber que el Entity Manager utiliza proxies para llamar al repositorio de objetos. Si no se han creado las clases repositorio se utiliza una implementación por defecto de un repositorio con métodos como: `find()`, `findAll()`, `findBy()`, `findOneBy()`, entre otros. Más adelante veremos cómo crear las clases repositorio.

En la siguiente línea obtenemos el repositorio de la entidad *Item* y utiliza el método *findAll* para obtener todos los objetos de esa entidad.

```
$productos = $em->getRepository('SaludComprasBundle:Item')->findAll();
```

Luego se renderiza la platilla y se le pasa como parámetro el arreglo de objetos productos.

Luego modificaremos la plantilla para mostrar el listado de productos

```
{# src/Salud/ComprasBundle/Resources/views/Producto/list.html.twig #}  
{% extends 'SaludComprasBundle::layout.html.twig' %}
```

```
{% block content%}
    <h1>Listado de productos</h1>
    <table class='listado'>
    {% for producto in productos %}
        <TR>
            <TD>
                <A href='{{ path('_producto_show',{'id': producto.id}) }}' >
                    {{producto.descripcionitem}}
                </A>
            </TD>
            <TD>
                {{producto.preciounitario}}
            </td>
            <td>
                {{producto.idEspecifico.codigoespecifico}}
            </td>
        </TR>
    {% endfor %}
    </table>
{% endblock %}
```

De esto podemos observar que para desplegar el valor de una variable la ponemos entre llaves dobles. Ejemplo `{{producto.preciounitario}}`

Por lo tanto la plantillas Twig tendrá dos tipos de delimitadores `{% ... %}` y `{{ }}`, el primero es utilizado para ejecutar declaraciones como por ejemplos ciclos *for* y el segundo para imprimir a la plantilla el resultado de una evaluación.

Mostrar los datos de un producto

Así como se realizó en un capítulo anterior, en el listado de los productos se podía dar clic en uno de ellos y se carga la información particular de ese producto, haremos eso en estos momentos.

En la plantilla de listar productos hay una nueva etiqueta relacionada con las

rutas

```
{{path ('_producto_show',{'id', producto.id}) }}
```

Esta se encarga de generar la ruta adecuada independientemente de qué entorno se esté ejecutando (app.php o app_dev.php) la etiqueta *path* es una función de las platillas Twig como primer parámetro le pasamos el nombre de la ruta (*_producto_show*) y luego un arreglo de parámetros dentro de la función en las platillas Twig el arreglo se representa entre llaves ({}).

El patrón de la ruta (*/producto/show/{id}*), que se agregará como anotación en el controlador, tiene una parte fija */producto/show* y una variable *{id}* lo que indica que se escribirá un valor y se almacenará como parámetro con nombre *id*

Creamos el controlador:

```
<?php
//.....

class ProductoController extends Controller
{
    //....
    /**
     * @Route("/producto/show/{id}", name="_producto_show")
     * @Template
     */
    public function showAction($id)
    {
        $em = $this->getDoctrine()->getEntityManager();

        $producto = $em->find("SaludComprasBundle:Item", $id);
        //Estas formas son equivalentes
        //$producto = $em->getRepository("SaludComprasBundle:Item")->find($id);
        //$producto = $em->getRepository("Salud\ComprasBundle\Entity\Item")->find($id);
```

```
        return array('producto'=>$producto);
    }
}
```

En el controlador buscamos el producto utilizando el método *find*, este hace una búsqueda por el campo de llave primaria, le indicamos la entidad y el valor que deseamos buscar en este caso el valor del parámetro *\$id*; para poder acceder a las variables pasadas por la URL las ponemos como parámetros del método, para este caso *public function showAction(\$id)*

Veamos la plantilla

```
{# src/Salud/ComprasBundle/Resources/views/Producto/show.html.twig #}
{% extends "SaludComprasBundle::layout.html.twig" %}

{% block content %}
<table>
    <tr>
        <td>DESCRIPCION: </td>
        <td>{{producto.descripcionitem}}</td>
    </tr>
    <tr>
        <td>PRECIO UNITARIO: </td>
        <td>{{producto.preciounitario}}</td>
    </tr>
    <tr>
        <td>UNIDAD DE MEDIDA: </td>
        <td>{{producto.idUnidadMedida.descripcionunidadmedida}}</td>
    </tr>
    <tr>
        <td>ESPECIFICO: </td>
        <td>{{producto.idEspecifico.descripcionespecifico}}</td>
    </tr>
</table>
{% endblock %}
```

Si observamos la barra de depuración, se han ejecutado tres consultas a la base de datos, una para recuperar el producto, otra para la unidad de medida y una más para su específico. La primera se realizó con el método *find* en el controlador y la segunda al momento de colocar en la plantilla `{{producto.idEspecifico.descripcionespecifico}}`. Doctrine sabe las relaciones entre las entidades y al solicitar algún dato por medio del campo de enlace realiza la consulta correspondiente a la base de datos.

6. FORMULARIOS

El manejo de formulario es una de las más comunes – y desafiante – de las tareas de un desarrollador web. Symfony2 integra un componente de formularios que hace el manejo de éstos más fácil. En este capítulo, se construirá un formulario desde el inicio, aprendiendo las más importantes características del componente de formularios.

Creando un simple formulario

Se creará una aplicación que desplegará productos. Dado que los usuarios necesitarán editar y crear productos, será necesario construir un formulario. Pero antes de empezar, enfoquémonos en la clase entidad *Item* la que representa y almacena un simple producto.

Este tipo de clase es comunmente llamada POPO “Plain-Old-PHP-Object” porque, hasta el momento, no se debe hacer nada con Symfony o cualquier otra librería. Es simplemente un objeto normal de PHP que directamente solventa un problema dentro de la aplicación (Ej.: la necesidad de representar un producto). Por supuesto, al final de este capítulo, serás capaz de enviar datos a una instancia de *Item* (por medio de un formulario), validar sus datos, y almacenarlos en la base de datos.

Construyendo el formulario

Ya que se ha creado una clase para la entidad *Item*, el siguiente paso es crear y procesar el formulario HTML. En Symfony2, esto se realiza construyendo un objeto formulario y renderizándolo en una plantilla. Todo esto se puede realizar dentro del controlador, utilizando el método `$this->createFormBuilder($entidad)`. Sin embargo, una mejor práctica es construir el formulario en una clase

separada, la cual puede ser reusada en cualquier lugar de la aplicación. Crearemos una nueva clase que contendrá la lógica para construir el formulario, utilizando el siguiente comando en la consola:

```
$ app/console generate:doctrine:form SaludComprasBundle:Item
```

Observar el código de la clase generada en `src/SaludComprasBundle/Form/ItemType.php`

Y en el controlador haremos lo siguiente:

```
// ...
// src/Salud/ComprasBundle/Controller/ProductoController.php

use Salud\ComprasBundle\Entity\Item;
use \Salud\ComprasBundle\Form\ItemType;
/**
 *
 * @Route("/producto/edit/{id}", name="_producto_edit", requirements={"id"="\d+"}),
 * @Route("/producto/create", name="_producto_create")
 * @Template
 */
public function editAction($id = null)
{
    $producto = new Item();
    $form = $this->createForm(new ItemType(), $producto);

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

Observemos la ruta de esta acción, en este caso vamos a utilizar esta acción tanto para crear nuevos productos como para modificar los ya existentes, para eso se crean dos rutas: la de crear no necesita el parámetro id y en la de editar es obligatorio.

En el archivo `src/SaludComprasBundle/form/ItemType.php`, se han agregado al formulario varios campos por medio del método `add` el cual tiene tres

parámetros:

- El nombre, que corresponde con el nombre de las propiedades de la clase con la cual está relacionada el formulario.
- El tipo de campo, Symfony2 viene con varios tipos de campos que se pueden ver en <http://symfony.com/doc/current/book/forms.html#book-forms-type-reference>
- Un arreglo con opciones propias para cada tipo de campo. Por ejemplo: *label* que especifica qué etiqueta se utilizará al mostrar el campo, *currency* para el tipo *money* que indica el formato de la moneda, y *required* que por defecto se pone con valor *true*.

Cambiamos el formulario para dejarlo de la siguiente forma:

```
// src/Salud/ComprasBundle/form/ItemType.php

// .....

public function buildForm(FormBuilder $builder, array $options)
{
    $builder
        ->add('descripcionitem', 'text', array('attr'=>array('size'=>50), 'label'=>
'Descripción'))
        ->add('autorizado', 'checkbox', array('required' => false))
        ->add('descontinuado', 'checkbox', array('required' => false))
        ->add('preciounitario', 'money', array('label'=>'Precio Unitario',
                                         'currency'=> 'USD'))
        ->add('bloqueado', 'checkbox', array('required'=> false))
        ->add('observaciones', 'textarea', array('required'=>false))
        ->add('idEspecifico')
        ->add('idEspecificoOnu')
        ->add('idUnidadMedida')
    ;
}
```

```
}
```

La configuración de los últimos tres campos la dejaremos para el siguiente capítulo.

PASARLO ANTES DE PERSONALIZAR EL FORMULARIO

Ahora que ya está creado el formulario el siguiente paso es renderizarlo. Esto se hace fácilmente pasando una “vista” del objeto formulario a la plantilla (mirar `$form->createView()` en el controlador) y utilizando un conjunto de funciones especiales para los formularios.

```
{# src/Salud/ComprasBundle/Resources/views/Producto/edit.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}

{% block content %}
<h1>Gestión de productos</h1>
<form action="{{ producto.id? path('_producto_edit', {'id':producto.id}):
    path("_producto_create") }}"
    method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type="submit" value="Enviar"/>
</form>
{% endblock %}
```

Para renderizar el formulario solo se ha utilizado una línea `{{ form_widget(form) }}` aunque generalmente se necesitará tener mayor control sobre la forma en que se renderiza el formulario.

Veamos el resultado, para esto agreguemos un enlace para la acción de edición en el listado de productos, agreguemos antes de la etiqueta `</TR>`

```
<TD>
    <A href="{{ path('_producto_edit', {'id': producto.id}) }}">
        EDITAR
    </A>
```

</td>

Será necesario definir el método [__toString\(\)](#) para las entidades Especifico y UnidadMedida

```
// src/SaludComprasBundle/Entity/UnidadMedida.php
//.....
class UnidadMedida
{
    public function __toString() {
        return $this->descripcionunidadmedida;
    }
}
```

Y para la clase Especifico quedará así

```
// src/SaludComprasBundle/Entity/Especifico.php
//.....
class Especifico
{
    public function __toString(){
        return '('.$this->codigoespecifico .') ' . $this->descripcionespecifico;
    }
}
```

Si ejecutamos el formulario aparecería vacío, recuperaremos el producto para llenar el formulario con los datos correspondientes, en la acción *editAction*, cambiemos la línea: *\$producto = new Item();* reemplazándola por:

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

if (isset($id)) {
    $producto = $em->find('SaludComprasBundle:Item', $id);
}
```

```
        if (!$producto)

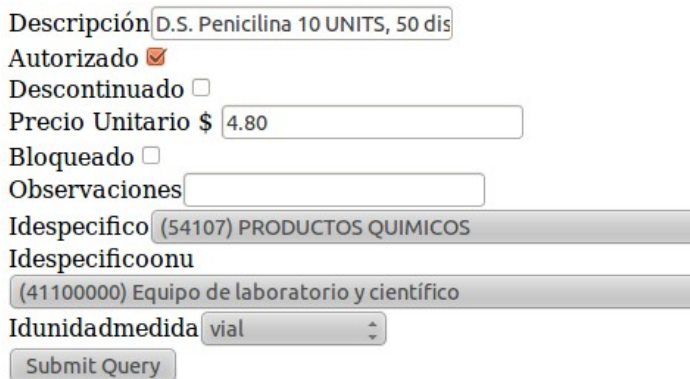
            throw $this->createNotFoundException ("Producto no encontrado");

    }

    else

        $producto = new Item();
```

El formulario resultante



Descripción

Autorizado ☒

Descontinuado ☐

Precio Unitario \$

Bloqueado ☐

Observaciones

Idespecifico

Idespecificoonu

Idunidadmedida

Illustration 15: Salida del formulario

En la plantilla, imprimiendo `form_widget(form)`, cada campo en el formulario es renderizado, con su correspondiente etiqueta y un posible mensaje de error. Aunque esto es fácil, no es muy flexible, posteriormente veremos como personalizar la salida del formulario.

Gestionando el envío del formulario

La segunda tarea a considerar con el manejo de formularios es trasladar los datos ingresados por el usuario y colocarlos en las respectivas propiedades del objeto. Para realizar esto, el formulario de datos enviado debe ser asociado al objeto del formulario:

```
// src/Salud/ComprasBundle/Controller/ProductoController.php
```

```
// .....
public function editAction($id=null) {
    //.....
    $form = $this->createForm(new ItemType(), $producto);

    if ($this->getRequest()->getMethod() == 'POST') {
        //Enlazar el formulario con los datos enviados por el usuario
        $form->bindRequest($this->getRequest());

        //Validar los datos ingresados en el formulario
        if ($form->isValid())
        {
            //Guardar el objeto (aún no en la base de datos)
            $em->persist($producto);

            //Escribir los cambios a la base de datos
            $em->flush();

            //Después de guardar exitosamente es recomendable
            //redirigir al usuario hacia otra página
            // para evitar que los datos se reenvien si refresca la página

            return $this->redirect($this->generateUrl('_producto_show',
                array('id'=>$producto->getId())));
        }
    }

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

Ahora, cuando el formulario es enviado, el controlador enlaza los datos enviados al formulario, el cual traslada los datos a las propiedades del objeto `$producto`. Esto se realiza con el método *`bindRequest()`*.

Tan pronto como el método *`bindRequest()`* es invocado, los datos enviados son transferidos al objeto. Así que si el objeto tiene ya valores en sus propiedades

estos serán sustituidos por los enviados en el formulario.

Una vez enlazados los datos enviados con el objeto se realiza la validación por medio del método *\$form->isValid()*, veremos la validación con may detalle más adelante.

El objeto es marcado para guardarse a la base de datos por medio del método *persist* del Entity Manager y el método *flush* es el que hace la actualización real a la base de datos.

El controlador sigue un patrón para manejar formularios, y son tres posibles caminos:

- Cuando inicialmente carga el formulario en el navegador, el método de solicitud es GET, lo que significa que el formulario es solamente creado y renderizado (pero no enlazado).
- Cuando el usuario envía el formulario (el método es POST), pero los datos son inválidos, el formulario es enlazado y luego renderizado, esta ocasión mostrando todos los mensajes de error.
- Cuando el usuario envía el formulario con datos válidos, el formulario es enlazado y se realiza las acciones necesarias (guardarlo en la base de datos) para luego redirigir al usuario a otra página (para mostrar por ejemplo: "Datos guardados con éxito"). Redirigir al usuario luego de haber guardado los datos, previene que el usuario tenga la posibilidad de refrescar la página y enviar nuevamente los datos.

Validar el formulario

En Symfony2, la validación es aplicada al objeto. En otras palabras, la pregunta no es si el formulario es válido, sino si el objeto producto es o no válido después que el formulario le ha trasladado los datos enviados. Llamando al método *\$form->isValid()* es un atajo para verificar si los datos del objeto *\$producto* son o no válidos.

Symfony2 incorpora un componente [Validator](#) que hace la tarea de validación fácil y transparente. Este componente está basado en [JSR303 Bean Validation specification](#). La cual es una especificación de Java, sí una especificación de Java en PHP.

La validación es realizada agregando un conjunto de reglas (llamadas [constraints](#)) a la clase. Para ver esto en acción, agreguemos las restricciones de validación, teniendo el cuidado de utilizar solo comillas dobles no las comillas simples(No *limit='0'* sino que *limit="0"*)

```
// . . . . .

use Symfony\Component\Validator\Constraints as Assert;

// . . . . .

class Item
{
    /**
     * @var decimal $preciounitario
     *
     * @ORM\Column(name="preciounitario", type="decimal", nullable=true)
     *
     * @Assert\NotBlank(message="Ingrese un valor")
     * @Assert\Min(limit="0", message="Debe ser mayor o igual que 0")
     */
    private $preciounitario;
```

Con esto si se reenvía el formulario con datos inválidos se verá el correspondiente mensaje de error.

Si se da un vistazo al código HTML generado, el componente para Formularios utiliza nuevos campos de HTML5 incluyendo un atributo especial “required” que realiza la validación en el mismo

navegador. Al gunos de los modernos navegadores como Firefox 4, Opera 9.5 o Chrome 3.0 entienden este nuevo atributo “required”.

Debemos distinguir de las validaciones hechas del lado del servidor y las realizadas del lado del cliente (con JavaScript o con HTML5), hemos agregado una validación del lado del servidor, cuando construimos el formulario se ponía *required=true* por defecto y ahora agregamos validación en la clase *@Assert\NotBlank()*. Aunque parezca redundante las dos son necesarias y si tuvieramos que elegir lo correcto sería quedarse con las validaciones del lado del servidor. Las validaciones del lado clientes se pueden desactivar (si están con javascript y se sabe ese lenguaje) o no están totalmente soportadas por todos los navegadores (HTML5) estas se pueden agregar para mejorar la interacción con el usuario pero para garantizar la validación será necesario hacerlas también del lado del servidor.

Usando el servicio validator

Si deseamos validar un objeto que no esté asociado a un formulario, debemos hacer uso del servicio *validator*. El trabajo del validador es fácil: lee las restricciones (es decir, las reglas) de una clase y comprueba si los datos en el objeto satisfacen esas restricciones. Si la validación falla, devuelve un arreglo de errores. Veamos un ejemplo sencillo:

```
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Autor;
// ...
public function indexAction()
{
    $autor = new Autor();
    // ... haz algo con el objeto $autor
    $validador = $this->get('validator');
    $errores = $validador->validate($autor);
    if (count($errores) > 0) {
```

```
        return new Response(print_r($errores, true));
    } else {
        return new Response('¡El autor es válido! ¡Sí!');
    }
}
```

Crear validaciones personalizadas.

Tenemos dos formas para agregar validaciones personalizadas, para una de las opciones Symfony2 nos permite agregar restricciones al valor devuelto por cualquier método público cuyo nombre comience con *get* o *is*.

Vamos a quitar la restricción *Min* colocada a la propiedad *preciounitario* y agregar un método para no permitir valores iguales o menores a cero.

```
/**
 * @Assert\True(message="Precio debe ser mayor que cero")
 */
public function isPrecioUnitarioPositivo() {
    return ($this->preciounitario > 0);
}
```

La segunda alternativa es crea una nueva restricción. Para esto necesitamos crear dos archivos:

```
<?php
// src/Salud/ComprasBundle/Validator/GreaterThan.php
namespace Salud\ComprasBundle\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class GreaterThan extends Constraint
{
    public $message = 'Este valor deber ser un número mayor que {{ limit }} ';
```

```
public $limit ;

public function getDefaultOption()
{
    return 'limit';
}
}
```

La clase es sencilla, sólo debemos definir los parámetros que soportará nuestra restricción, además debe ser obligatorio que la clase extienda de `Symfony\Component\Validator\Constraint` y que lleve la anotación `@Annotation` para que pueda ser ocupada en anotaciones. La clase que realmente hace la validación es:

```
<?php
// src/Salud/ComprasBundle/Validator\GreaterThanValidator.php
namespace Salud\ComprasBundle\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class GreaterThanValidator extends ConstraintValidator
{
    public function isValid($value, Constraint $constraint)
    {
        if (null === $value) {
            return true;
        }

        if (!is_numeric($value) or $value <= $constraint->limit) {
            $this->setMessage($constraint->message, array(
                '{{ value }}' => $value,
                '{{ limit }}' => $constraint->limit,
            ));

            return false;
        }

        return true;
    }
}
```

Esta clase lleva el nombre de la anterior más el sufijo *Validator*, lo más importante aquí es el método *isValid* que es dónde se realizará la validación. Observemos que recibe dos parámetros el dato a validar (*\$value*) y un objeto *Constrain* (*\$constraint*) por medio del cual accedemos a las propiedades de la clase *GreaterThan*.

Ahora tenemos la validación para poder utilizarla en dónde deseemos, vamos a ponerla a la propiedad *preciounitario*:

```
<?php
//src/Salud/ComprasBundle/Entity/Item.php
// . . .
use Salud\ComprasBundle\Validator as Assert2;
// . . .
class Item
{
    // . . . .
    /**
     * @var decimal $preciounitario
     *
     * @ORM\Column(name="preciounitario", type="decimal", nullable=true)
     * @Assert\NotBlank(message="Ingrese un valor")
     * @Assert2\GreaterThan(0)
     */
    private $preciounitario;
```

Renderizando el formulario en la plantilla

Como recordaremos el formulario fue renderizado con una sola línea (`{{ form_widget(form) }}`) pero usualmente necesitaremos tener mayor control sobre cómo se renderiza el formulario:

```
{# src/Salud/ComprasBundle/Resources/views/Producto/edit.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}

{% block content%
```

```
<form action="{{ producto.id? path('_producto_edit', {'id':producto.id}):
    path('_producto_create') }}" method="POST" {{ form enctype(form) }}>
    {{ form_errors(form) }}
    {{ form_row(form.descripcionitem) }}
    {{ form_row(form.autorizado) }}
    {{ form_row(form.preciounitario) }}
    {{ form_row(form.observaciones) }}
    {{ form_rest(form) }}
    <input type="submit" value='Enviar' />
</form>
{% endblock%}
```

Demos un vistazo a cada parte:

- `form enctype(form)` – Si se ha incluido al menos un campo para subir archivos, esto renderiza el obligatorio `enctype="multipart/form-data"`.
- `form_errors(form)` – Rederiza cualquier error global para todo el formulario (errores para un campo específico se muestra a la par de cada campo).
- `form_row(form.descripcionitem)` – Renderiza la etiqueta, algún error, y el código HTML para obtener el campo.
- `form_rest(form)` – Renderiza cualquier campo que no haya sido renderizado todavía. Usualmente es buena idea colocar la llamada a este helper al final de cada formulario (en caso de que se haya olvidado renderizar o para no renderizar manualmente los campos ocultos). Este helper es además muy útil para aprovechar la ventaja de la verificación automática de la protección de [Ataques CSRF](#)

La mayoría del trabajo es realizado por el helper `form_row` el cual renderiza la etiqueta, los errores y el código HTML de cada campo dentro de una etiqueta `<div>` por defecto.

Renderizando manualmente cada campo

El helper `form_row` es muy útil porque se puede muy rápidamente renderizar cada archivo del formulario (y la etiqueta utilizada para cada “fila” puede personalizarse también). Pero dado que en ocasiones necesitaremos mayor control se puede renderizar cada campo totalmente a mano.

```
{{ form_errors(form) }}

<div>
    {{ form_label(form.descripcionitem) }}
    {{ form_errors(form.descripcionitem) }}
    {{ form_widget(form.descripcionitem) }}
</div>

<div>
    {{ form_label(form.autorizado) }}
    {{ form_errors(form.autorizado) }}
    {{ form_widget(form.autorizado) }}
</div>

<div>
    {{ form_label(form.preciounitario) }}
    {{ form_errors(form.preciounitario) }}
    {{ form_widget(form.preciounitario) }}
</div>

<div>
    {{ form_label(form.observaciones) }}
    {{ form_errors(form.observaciones) }}
    {{ form_widget(form.observaciones) }}
</div>

{{ form_rest(form) }}
```

Utilizar temas de formularios

Cada parte del renderizado de un formulario puede ser personalizada. Eres libre de cambiar como se renderiza cada fila del formulario, cambiar las etiquetas usadas para renderizar errores, o incluso personalizar cómo una etiqueta *textarea* es renderizada. No existen limitaciones, y las diferentes configuraciones pueden ser utilizadas en diferentes lugares.

Symfony usa plantillas para renderizar cada una de las partes del formulario. En Twig, esas partes - una fila, una etiqueta *textarea*, errores - son representadas por etiquetas “blocks”. Para personalizar cualquier parte de cómo un formulario es renderizado, solamente es necesario sobrescribir el bloque apropiado.

Para entender cómo funciona, personalizemos la salida de *form_row* para agregar un atributo *class* a la etiqueta *div* que encierra cada fila. Para hacer esto, crearemos una nueva plantilla *fields* que contendrá las nuevas etiquetas.

```
{# src/Salud/ComprasBundle/Resources/views/fields.html.twig #}

{% block field_row %}
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock field_row %}
```

El bloque *field_row* es el nombre del bloque utilizado cuando se renderizan campos por medio de la función *form_row*. Para utilizar el bloque *field_row* que hemos definido, agregaremos la llamada en la plantilla a la cual nos interese

agregar el tema.

```
{#src/Salud/SaludBundle/Resources/views/Producto/edit.html.twig#}  
{% form_theme form 'SaludComprasBundle::fields.html.twig' %}
```

La etiqueta *form_theme* “importa” la plantilla y utiliza todos los bloques relacionados cuando se renderiza el formulario. En otras palabras cuando *form_row* es invocado en la plantilla, se utilizará el bloque desde la plantilla *fields.html.twig*

Además se puede colocar de forma general para todos los bundles, esto se puede realizar agregando en el archivo *app/config/config.yml*:

```
# app/config/config.yml  
twig:  
  form:  
    resources: ['SaludComprasBundle::fields.html.twig']  
  # ...
```

Utilizaremos esta última forma

Para una discusión más exhaustivo, mirar [How to customize Form Rendering](#).

Estilos

Vamos a utilizar un poco de hojas de estilos, pero como no es ese el objetivo de esta guía no entraremos en detalle ni se explicará código css, así que cada uno puede agregar los estilos e imágenes que le parezcan o no agregarlos, para esta guía agregué la hora de estilos que viene con la demo de Symfony2 más unos pequeños ajustes.

Primero dentro de tu bundle debes crear las carpetas *src/Salud/ComprasBundle/Resources/public/images* y

scr/Salud/ComprasBundle/Resources/public/css y colocar ahí las imágenes y hojas de estilos respectivamente.

Luego será necesario instalarlas con la siguiente instrucción en la consola:

```
app/console assets:install web --symlink
```

Donde el parámetro *web* es la carpeta raíz de tu sitio web.

La instrucción recorre todos los bundles y copia los archivos de imágenes y hojas de estilos que encuentra en la carpeta *public* que creamos antes. Esto nos facilita la tarea de copiar manualmente estos archivos cada vez que un bundle lo requiera. Observemos que los ubica dentro del directorio raíz definido para el sitio web y ahí las carpetas *bundles/nombrebundle/css* y *bundles/nombrebundle/images* luego para utilizarlos debemos recordar esa ruta.

Para agregar una hoja de estilos dentro de una platilla Twig lo hacemos con el helper *asset* y en este caso se ha definido dentro de un bloque *stylesheets*, quedando el layout de la siguiente forma:

```
{#src/Salud/ComprasBundle/Resources/views/layout.html.twig#}
{% extends '::base.html.twig' %}
{% block title %}
Gesti&oacute;n de Compras Institucional
{% endblock %}
{% block stylesheets %}
    <link rel="stylesheet" href="{{ asset('bundles/saludcompras/css/main.css') }}"
type="text/css" media="all" ></link>
{% endblock %}
{% block body %}
    <div id="symfony-wrapper">
        <div id="logo-site">
            <a href="">
                
    </a>
</div>
<div id="symfony-header">
</div>

{% if app.session.flash('notice') %}
    <div class="flash-message">
        <em>Aviso</em>: {{ app.session.flash('notice') }}
    </div>
{% endif %}

{% block content_header %}
    <ul id="menu">
        {% block content_header_more %}

        {% endblock %}
    </ul>

    <div style="clear: both"></div>
{% endblock %}

<div class="symfony-content">
    {% block content %}
    {% endblock %}
</div>
</div>
{% endblock %}
```

Agregaremos otra hoja de estilo para cambiar el formato de los listados, en el archivo `src/Salud/ComprasBundle/Resources/views/Producto/list.html.twig`, si se agregara como lo hemos hecho antes quedaría así:

```
{% block stylesheets%}
    <link rel="stylesheet" href="{{ asset('bundles/saludcompras/css/productos.css') }}"
    type="text/css" media="all" />
{% endblock %}
```

```
{% endblock %}
```

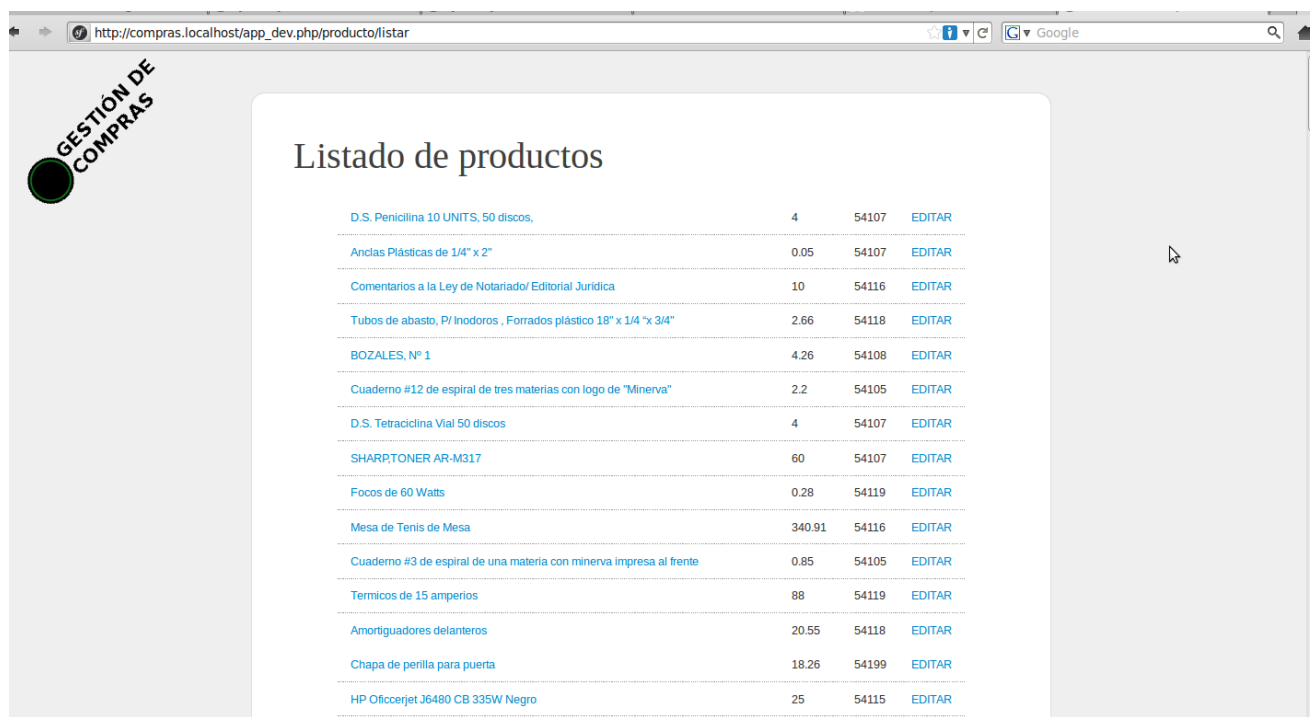
Pero el problema es que por estar en el bloque llamado *stylesheets* sustituiría el que se agregó en el *layout*, lo cual no deseamos que suceda para eso vamos a utilizar el helper *parent*:

```
{% block stylesheets%}
    {{ parent() }}
    <link rel="stylesheet" href="{{ asset('bundles/saludcompras/css/list.css') }}"
type="text/css" media="all" />
{% endblock %}
```

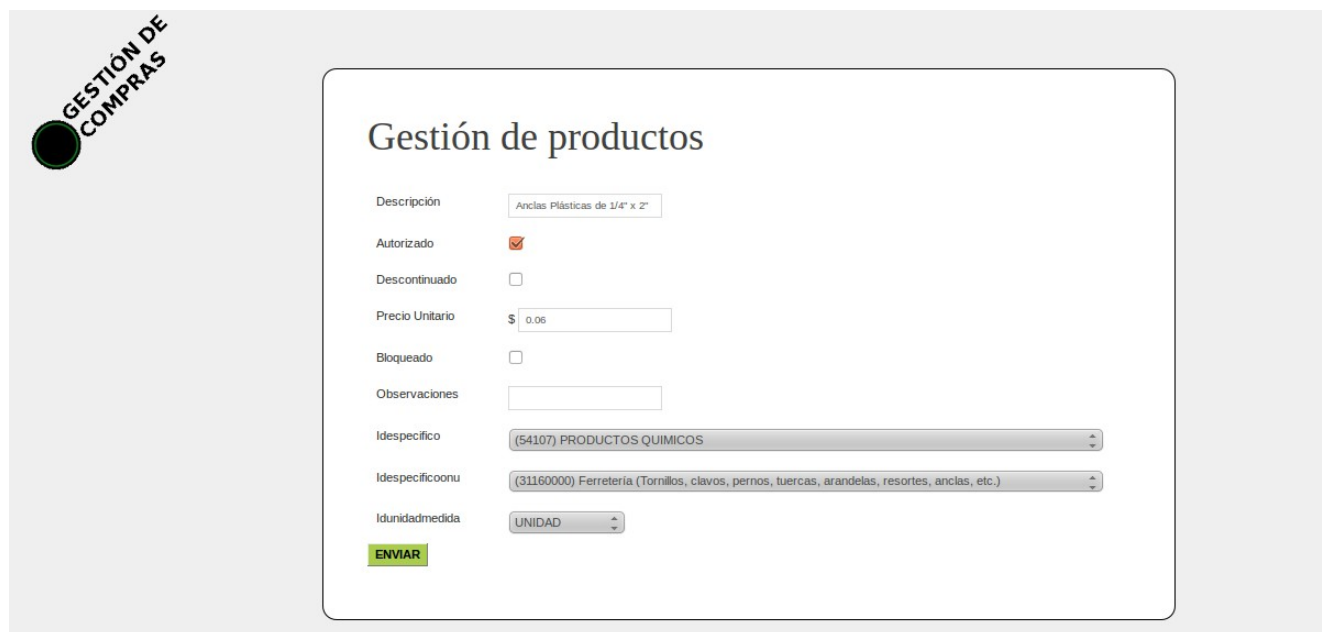
De esta forma primero se agrega el contenido del bloque padre (en este caso el que definimos en el layout) y luego se agrega la línea particular para esta plantilla, así el contenido del bloque no es sobrescrito.

Agregaremos otra hoja en el archivo *src/Salud/ComprasBundle/Resources/views/Producto/edit.html.twig*,

```
{% block stylesheets%}
    {{ parent() }}
    <link rel="stylesheet" href="{{ asset('bundles/saludcompras/css/producto.css') }}"
type="text/css" media="all" />
{% endblock %}
```



Descripción	Cantidad	ID	Acción
D.S. Penicilina 10 UNITS, 50 discos,	4	54107	EDITAR
Anclas Plásticas de 1/4" x 2"	0.05	54107	EDITAR
Comentarios a la Ley de Notariado/ Editorial Jurídica	10	54116	EDITAR
Tubos de abasto, P/ Inodoros , Forrados plástico 18" x 1/4 "x 3/4"	2.66	54118	EDITAR
BOZALES, Nº 1	4.26	54108	EDITAR
Cuaderno #12 de espiral de tres materias con logo de "Minerva"	2.2	54105	EDITAR
D.S. Tetraciclina Vial 50 discos	4	54107	EDITAR
SHARPTONER AR-M317	60	54107	EDITAR
Focos de 60 Watts	0.28	54119	EDITAR
Mesa de Tennis de Mesa	340.91	54116	EDITAR
Cuaderno #3 de espiral de una materia con minerva impresa al frente	0.85	54105	EDITAR
Termicos de 15 amperios	88	54119	EDITAR
Amortiguadores delanteros	20.55	54118	EDITAR
Chapa de perilla para puerta	18.26	54199	EDITAR
HP Officejet J6480 CB 335W Negro	25	54115	EDITAR

Illustration 16: Salida de listados de productos

Gestión de productos

Descripción:

Autorizado: ☒

Descontinuado: ☐

Precio Unitario: \$

Bloqueado: ☐

Observaciones:

Id específico:

Id específico o nu:

Id unidad medida:

ENVIAR

Illustration 17: Formulario de edición de productos

7. MÁS CON EL MODELO

Al formulario que hemos construido le falta la configuración de tres campos, los cuales corresponden a las llaves foráneas. Vamos a modificar la clase del formulario, y sustituir los campos que aparecen a continuación.

```
<?php

// src/Salud/ComprasBundle/Form/ItemType.php

// . . .

->add('idEspecifico', 'entity',
    array('class' => 'SaludComprasBundle:Especifico',
        'query_builder' => function ($repository) {
            return $repository->createQueryBuilder('e')
                ->orderBy('e.descripcionespecifico')
                ->join('e.idCatalogoProducto', 'c')
                ->where('c.codigocatalogo = :codigo_catalogo')
                ->setParameter('codigo_catalogo', '02')
            ;
        }
    ))
->add('idEspecificoOnu', 'entity',
    array('class' => 'SaludComprasBundle:Especifico',
        'query_builder' => function ($repo) {
            return $repo->createQueryBuilder('e')
                ->orderBy('e.descripcionespecifico')
                ->join('e.idCatalogoProducto', 'c')
                ->where('c.codigocatalogo = :codigo_catalogo')
                ->setParameter('codigo_catalogo', '01')
            ;
        }
    )
->add('idUnidadMedida', 'entity',
    array('class' => 'SaludComprasBundle:UnidadMedida',
        'query_builder' => function ($repository){
```

```
        return $repository->createQueryBuilder('um')
            ->orderBy('um.descripcionunidadmedida');
    }
)
// ...
```

Para los tres campos hemos ocupado el tipo [entity](#) este es un campo especial [choice](#) que está diseñado para cargar las opciones desde una entidad de Doctrine. Por el ejemplo para el caso del campo IdUnidadMedida muestras todas las unidades de medida almacenadas en la base de datos.

Las opciones propias para este nuevo campo son:

- class **obligatorio** [tipo: cadena de texto]: Es la clase que representa la entidad
- property [tipo: cadena de texto]: Este es la propiedad que debería utilizarse para desplegar las entidades como texto en el elemento HTML, si se omite se utiliza el método `__toString()` .
- query_builder [tipo: Doctrine\ORM\QueryBuilder]: Si es especificado, este es utilizado para realizar la consulta a la base de datos de las opciones que se utilizarán en el campo del formulario

Este último parámetro es el que revisaremos con mayor detalle, aquí se utiliza un objeto de tipo QueryBuilder.

El objeto QueryBuilder

Este objeto provee la API (*Application Programming Interface*) para la construcción de una consulta DQL (*Doctrine Query Language*) en varios pasos.

Proporciona un conjunto de clases y métodos para contruir consultas a la base

de datos, y además provee una API fluida.

Para trabajar con el QueryBuilder se crea a partir de una instancia del Entity Manager:

```
// $em instancia de EntityManager
$qb = $em->createQueryBuilder();
```

La mayor parte del trabajo con el QueryBuilder se realiza con el método add, es este método el responsable de construir cada pieza del DQL.

```
$qb->add('select', 'u')
->add('from', 'User u')
->add('where', 'u.id = :id_usuario')
->add('orderBy', 'u.name ASC');
```

Para especificar parámetros lo hacemos a través del método setParameter

```
$qb->add('select', 'u')
->add('from', 'User u')
->add('where', 'u.id = :id_user')
->add('orderBy', 'u.name ASC')
->setParameter('id_user', 100);
```

Alternativamente podemos utilizar los métodos helpers del QueryBuilder, con lo cual la construcción de la consulta nos resultará más familiar al SQL:

```
$qb->select('u')
->from('User u')
->where('u.id = :id_user')
->orderBy('u.name ASC')
->setParameter('id_user', 100);
```

El QueryBuilder solamente se encarga de construir el objeto pero no ejecuta la consulta. Es por eso que se debe convertir una instancia del QueryBuilder en un objeto Query:

```
// $qb instancia del QueryBuilder
$query = $qb->getQuery();
```

```
// Ejecutar la consulta, tenemos varias opciones de acuerdo al
// resultado que deseemos
$result = $query->getResult();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

Para mayor información consultar la documentación oficial de Doctrine en el capítulo de [QueryBuilder](#)

Usando el QueryBuilder en el campo Entity

Para nuestro caso particular del campo *entity* el QueryBuilder se realiza a través de las clases de repositorio de las cuales ya hemos hablado un poco en el capítulo del modelo y revisaremos con más detalle más adelante, analicemos el código:

```
->add('idUnidadMedida', 'entity',
    array('class' => 'SaludComprasBundle:UnidadMedida',
        'query_builder' => function ($repository){
            return $repository->createQueryBuilder('um')
                ->orderBy('um.descripcionunidadmedida');
        }
    )
)
```

Específicamente la línea

```
'query_builder' => function ($repository){
    return $repository->createQueryBuilder('um')
        ->orderBy('um.descripcionunidadmedida');
}
```


El parámetro *query_builder* en este caso es una función que recibe como parámetro una instancia de la clase repositorio de la entidad que se haya especificado en el parámetro *class*, la clase repositorio contiene el método para crear objetos *QueryBuilder* el cual podemos contruir con las opciones antes presentadas.

Observemos en los otros campos de tipo *Entity* en el *QueryBuilder* correspondiente se realiza un join para ello solo se especifica la propiedad de la entidad que hace las veces de llave foránea, Doctrine ya tiene registrado (en la definición del modelo) la forma en que se debe establecer la relación (Entidad y propiedad destino)

Clases repositorio

Estas clases son para implementar el patrón de diseño Repository.

Ya hemos mencionado que las consultas a la base de datos se hacen a través del Entity Manager, a su vez éste utiliza las clases repositorio, que en mundo de Java sería equivalente a la capa o clases DAO (Data Access Object). Por ejemplo:

```
$user = $em->getRepository('Entities\User')->find($id);
```

El Entity Manager buscará el repositorio que está asignado a la clase entidad. Si no se ha definido uno se utilizará por defecto un repositorio de la clase (Doctrine\ORM\EntityRepository) el cual contiene los siguientes métodos para el acceso a los datos:

```
public function find()

public function findAll()

public function findBy (array $criteria)

public function findOneBy(array $criteria)
```

Con esos simples métodos se puede acceder a los datos, pero comúnmente éstos no serán suficientes. La clase `EntityRepository` además contiene el método *`createQueryBuilder`*, el cual ya hemos utilizado antes, pero cuando necesitemos escribir nuestras propias consultas ya sea con el `QueryBuilder`, DQL o en SQL nativo, una buena práctica, para respetar el patrón MVC, es no incluir el código en el controlador, para esto será necesario crear nuestras clases repositorio para encapsular las consultas.

Primero debemos especificar el nombre de la clase repositorio en la definición de la entidad por medio de: `@ORM\Entity(repositoryClass=“”)`, veamos con la clase *Específico*:

```
<?php
// src/Salud/ComprasBundle/Entity/Especifico.php
namespace Salud\ComprasBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Salud\ComprasBundle\Entity\Especifico
 *
 * @ORM\Table(name="especifico")
 * @ORM\Entity(repositoryClass="Salud\ComprasBundle\Repository\EspecificoRepository")
 */
class Especifico
{
    // . . . . .
}
```

Luego creamos la clase repositorio en el directorio `Repository` del bundle:

```
app/console doctrine:generate:entities SaludComprasBundle
```

Y obtendremos esta clase vacía (podemos no realizar el comando anterior y crear la clase manualmente):

```
<?php

// src/Salud/ComprasBundle/Repository/EspecificoRepository

namespace Salud\ComprasBundle\Repository;

use Doctrine\ORM\EntityRepository;

/**
 * EspecificoRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class EspecificoRepository extends EntityRepository
{
}
```

Esta clase extiende de la clase EntityRepository, una vez creada la clase podemos agregar los métodos para realizar las consultas personalizadas a la base de datos, en el siguiente ejemplo se ha agregado el método *getByCatalogo(\$catalogo)* el cual utilizaremos para refactorizar el código de la definición del formulario anterior.

```
<?php

namespace Salud\ComprasBundle\Repository;

use Doctrine\ORM\EntityRepository;

class EspecificoRepository extends EntityRepository
{
    public function getByCatalogo($catalogo) {
        $qb = $this->createQueryBuilder('e')
            ->orderBy('e.descripcionespecifico')
            ->join('e.idCatalogoProducto', 'c')
```

```

        ->where('c.codigocatalogo = :catalogo_producto')
        ->setParameter('catalogo_producto', $catalogo);
    return $qb;
}
}

```

Ahora en el formulario anterior los últimos dos campos que agregamos construían una consulta cada uno, la cual si observamos es casi la misma, solo difiere en el parámetro del catálogo a utilizar, podemos simplificar el código para que utilicen el mismo método de la clase repositorio

```

<?php
// src/Salud/ComprasBundle/Form/ItemType.php
// . . . .
class ItemType extends AbstractType {
    public function buildForm(FormBuilder $builder, array $options) {
        $builder->add('descripcionitem', 'text', array('label' => 'Descripción'))
        // . . . .
        ->add('idEspecifico', 'entity',
            array('class' => 'SaludComprasBundle:Especifico',
                'property' => 'descripcionespecifico',
                'query_builder' => function ($repository){
                    return $repository->getByCatalogo('02');
                },
                'label' => 'Específico Catálogo Egresos')
        )

        ->add('idEspecificoOnu', 'entity',
            array('class' => 'SaludComprasBundle:Especifico',
                'property' => 'descripcionespecifico',
                'query_builder' => function ($repository){
                    return $repository->getByCatalogo('01');
                },
                'label' => 'Específico Catálogo ONU')
        );
        // . . . . .
    }
}

```

Agregaremos en la plantilla de edición del producto la siguiente hoja de estilo para

controlar el tamaño del control para el específico de la ONU

```
{# src/Salud/ComprasBundle/Resources/views/Producto/edit.html.twig #}
{# . . . . . #}
{% block stylesheets%}
    {{ parent() }}
    <link rel="stylesheet"
        href="{{ asset("bundles/saludcompras/css/producto.css") }}"
        type="text/css" media="all"
    ></link>
{% endblock %}
{# . . . . . #}
```

El formulario resultante quedará de la siguiente forma:



Formulario para la edición de productos. El formulario contiene los siguientes campos:

- Descripción:** Campo de texto con el valor "Anclas Plásticas de 1/4" x 2".
- Autorizado:** Campo de selección con un icono de checkmark verde.
- Precio Unitario:** Campo de texto con el valor "\$ 0.06".
- Observaciones:** Campo de texto vacío.
- Unidad de Medida:** Selector de lista desplegable con el valor "UNIDAD".
- Específico Catálogo Egresos:** Selector de lista desplegable con el valor "PRODUCTOS QUIMICOS".
- Específico Catálogo Egresos:** Selector de lista desplegable con el valor "Ferretería (Tornillos, clavos, pernos, tuercas, arandelas, resortes, anclas, etc.)".
- ENVIAR:** Botón de envío.

Illustration 18: Formulario para la edición de productos

8. GENERADORES INTERACTIVOS: Interfaces CRUD

Ya hemos utilizado un generador interactivo de Symfony2, cuando creamos el bundle, ahora es el turno del generador para interfaces CRUD (Create Retrieve Update Delete)

Crear la aplicación para la planificación de compras

- Iniciemos el generados de interfaces CRUD

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

- Lo primero que nos solicitará es el nombre completo de la entidad sobre la cual vamos a crear la interface CRUD, escribiremos:
`SaludComprasBundle:PlanCompras`
- Luego nos pregunta si deseamos generar las acciones de escritura (editar, borrar y agregar), respondamos `yes`
- En este paso preguntará sobre el formato que utilizaremos elijamos la opción por defecto (annotation) presionando ENTER.
- Ahora nos pregunta el prefijo que utilizaremos para las rutas, aceptemos la opción por defecto (*plancompras*)
- En el último paso nos pedirá confirmación para realizar las acciones según lo que especificamos en todos los pasos anteriores.

Esto nos ha generado el controlador y las plantillas necesarias para realizar las acciones de listar, mostrar, editar, crear y borrar planes de compras. El código generado es un código que nos servirá de base, generalmente será necesario personalizarlo, por ejemplo para que funcione adecuadamente debemos agregar el método `__toString()` a varias entidades, en lugar de eso vamos a profundizar un poco más y optimizaremos el código.

Modificaremos el controlador que creó el generador interactivo, abramos el archivo: `src/Salud/ComprasBundle/Controller/PlanComprasController`.

Cambiaremos el método *indexAction*, las líneaa:

```
$entities = $em->getRepository('SaludComprasBundle:PlanCompras')->findAll();  
return array('entities' => $entities);
```

La cambiamos para que no utilice el método *findAll()* y quedará así:

```
$planes = $em->getRepository('SaludComprasBundle:PlanCompras')->getPlanes();  
return array('planes' => $planes->getResult());
```

Hemos utilizado el método *getPlanes()* del repositorio PlanCompras, creemos la clase repositorio junto a este método.

```
<?php  
  
namespace Salud\ComprasBundle\Repository;  
  
use Doctrine\ORM\EntityRepository;  
  
class PlanComprasRepository extends EntityRepository  
{  
    /**  
     * Obtiene los planes de compras con todas sus relaciones  
     */  
  
    public function getPlanes() {  
        $qb = $this->createQueryBuilder('plan')  
            ->select('plan, fuente, subfuente, periodo,  
                unidadSolicitante, unidadFinanciadora, origen')  
            ->join('plan.idFuenteFinanciamiento', 'fuente')  
            ->join('plan.idSubfuenteFinanciamiento', 'subfuente')  
            ->join('plan.idPeriodoFiscal', 'periodo')  
            ->join('plan.idUnidadSolicitante', 'unidadSolicitante')  
            ->join('plan.idUnidadFinanciadora', 'unidadFinanciadora')  
            ->join('fuente.idOrigenFinanciamiento', 'origen')  
        ;  
    }  
}
```

```
        return $qb->getQuery();
    }
}
```

Recordemos especificar la clase repositorio en su respectiva entidad

```
<?php
```

```
namespace Salud\ComprasBundle\Entity;
```

```
use Doctrine\ORM\Mapping as ORM;
```

```
/**
 * Salud\ComprasBundle\Entity\PlanCompras
 *
 * @ORM\Table(name="plan_compras")
 * @ORM\Entity(repositoryClass="Salud\ComprasBundle\Repository\PlanComprasRepository")
 */
class PlanCompras
{
    // . . . .
}
```

Ahora veamos cómo quedará la plantilla

```
{# src/Salud/ComprasBundle/Resources/views/PlanCompras/index.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}
{% block stylesheets%}
    {{ parent() }}
    <link rel="stylesheet"
        href="{{ asset('bundles/saludcompras/css/list.css') }}"
        type="text/css" media="all"
    />
{% endblock %}
{% block content %}
<ul>
    <li>
        <a href="{{ path('plancompras_new') }}">
```



```

        Crear nuevo plan
    </a>
</li>
</ul>
<TABLE class='listado'>
    <caption>Planes de compras</caption>
    <thead>
        <TR>
            <Th>Código plan</Th>
            <Th>Unidad solicitante</Th>
            <Th>Número plan por unidad</Th>
            <Th>Fuente</Th>
            <Th>Periodo fiscal</Th>
            <Th>Fecha autorización</th>
            <Th>Fecha envío</th>
            <Th>Acciones</th>
        </tr>
    </thead>
    <TBODY>
        {% for plan in planes %}
        <TR>
            <TD>{{plan.idUnidadSolicitante.codigounidad}}-
                {{plan.idFuenteFinanciamiento.codigofuente}}-
                {{plan.idPeriodoFiscal.aniofiscal}}
            </TD>
            <TD>{{plan.idUnidadSolicitante.nombreunidad}}</TD>
            <TD>{{plan.numeroplan}}</TD>
            <TD>{{plan.idFuenteFinanciamiento.descripcionfuente}}</TD>
            <TD>{{plan.idPeriodoFiscal.aniofiscal}}</TD>
            <td>{{ plan.fechaAutorizacion?plan.fechaAutorizacion|date('d-m-Y H:i:s'): "---"
}}</td>
            <td>{{plan.fechaEnvio ? plan.fechaEnvio|date('d-m-Y H:i:s'): "---" }}</td>
            <TD>
                <ul>
                    <li>
                        <a href="{{ path('plancompras_show', { 'id':
plan.id }) }}">Mostrar</a>
                    </li>
                    <li>
                        <a href="{{ path('plancompras_edit', { 'id':
```

```

plan.id }} }}">Editar</a>
        </li>
    </ul>
</td>
</td>
</tr>
{% endfor %}
</TBODY>
</table>
{% endblock %}

```

La salida será similar a esto:

Código plan	Unidad solicitante	Número plan por unidad	Fuente	Período fiscal	Fecha autorización	Fecha envío	Acciones
01000000-006- 2011	Contabilidad	0001	BID	2011	---	---	Mostrar Editar

Illustration 19: Listado de planes de compras

Ahora haremos lo relacionado a mostrar un plan de compras en particular, vamos a modificar la plantilla.

```

{# src/Salud/ComprasBundle/Resources/views/PlanCompras/show.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}

{% block content %}
<table >
    <tr>

```

```

        <th>Plan:</th>
        <td >
            {{entity.idUnidadSolicitante.codigounidad}} -
            {{entity.idFuenteFinanciamiento.codigofuente}} -
            {{entity.idPeriodoFiscal.aniofiscal}}
        </td>
        <th align=right>Período Fiscal:</th>
        <TD >
            {{entity.idPeriodoFiscal.aniofiscal}}
        </TD>
        <th align=right>Número convenio:</th>
        <TD >
            {{entity.numeroconvenio?entity.numeroconvenio:"----"}}
        </TD>

    </tr>
    <tr>
        <th width='21%'>Unidad / Facultad:</th>
        <td colspan="5" >
            {{entity.idUnidadSolicitante.nombreunidad}}
        </td>
    </tr>
    <tr>
        <th >Línea de trabajo que financia:</th>

        <td colspan="5" >
            {{entity.idUnidadFinanciadora.nombreunidad}}
        </td>
    </tr>
    <tr>
        <th >Origen Financiamiento:</th>
        <td >
            {{entity.idFuenteFinanciamiento.idOrigenFinanciamiento.descripcionorigen}}
        </td>
        <th width=15% align=right>Fuente Financiamiento:</th>
        <td colspan="3" >
            {{entity.idFuenteFinanciamiento.descripcionfuente}}
        </td>
    </tr>

```

```
</table>
</br></br>
<ul class="record_actions">
  <li>
    <a href="{{ path('plancompras') }}">
      Regresar al listado
    </a>
  </li>
  <li>
    <a href="{{ path('plancompras_edit', { 'id': entity.id }) }}">
      Editar
    </a>
  </li>
  <li>
    <form action="{{ path('plancompras_delete', { 'id': entity.id }) }}" method="post">
      {{ form_widget(delete_form) }}
      <button type="submit">Borrar Plan</button>
    </form>
  </li>
</ul>
{% endblock %}
```

La salida será como esta:

Plan:	01000000 - 006 - 2011	Período Fiscal:	2011	Número convenio:	----
Unidad / Facultad:	Contabilidad				
Línea de trabajo que financia:	Contabilidad				
Origen Financiamiento:	Externos	Fuente Financiamiento:	BID		
Regresar al listado					
Editar					
<button>Borrar Plan</button>					

Illustration 20: Mostrar plan de compras

Ahora vamos a desarrollar las acciones para crear y editar un plan de compras. Para esto modificaremos el archivo *src/Salud/ComprasBundle/Form/PlanComprasType.php* para adecuar las etiquetas y el orden de los datos de las tablas relacionadas, sustituir el contenido del método *buildForm*, por lo siguiente:

```
$builder
->add('numeroplan','text', array('label'=>'Número de plan') )
->add('numeroconvenio', 'text', array('label'=> 'Número de convenio',
                                     'required'=>false))
->add('idFuenteFinanciamiento', 'entity',
      array('label'=>'Fuente Financiamiento',
            'class'=>'SaludComprasBundle:FuenteFinanciamiento',
            'query_builder'=>function($repo) {
                return $repo->createQueryBuilder('f')
                    ->orderBy('f.descripcionfuente');
            })
->add('idSubfuenteFinanciamiento', 'entity',
      array('label'=>'Subfuente Financiamiento',
            'class'=>'SaludComprasBundle:SubfuenteFinanciamiento',
            'query_builder'=>function($repo) {
                return $repo->createQueryBuilder('sf')
                    ->orderBy('sf.descripcionsubfuente');
            })
->add('idPeriodoFiscal', 'entity', array('label'=>'Periodo Fiscal',
            'class'=>'SaludComprasBundle:PeriodoFiscal',
            'query_builder'=>function($repo) {
                return $repo->createQueryBuilder('pf')
                    ->orderBy('pf.aniofiscal', 'DESC');
            })
->add('idUnidadSolicitante', 'entity', array('label'=>'Unidad Solicitante',
            'class'=>'SaludComprasBundle:UnidadSolicitante',
            'query_builder'=>function($repo) {
                return $repo->createQueryBuilder('us')
                    ->orderBy('us.nombreunidad');
            })
```

```
->add('idUnidadFinanciadora', 'entity', array('label'=>'Unidad Financiadora',
    'class'=>'SaludComprasBundle:UnidadSolicitante',
    'query_builder'=>function($repo) {
        return $repo->createQueryBuilder('uf')
            ->orderBy('uf.nombreunidad');
    })
);
```

Debemos agregar el método `__toString` a las entidades con las cuales tiene relación la entidad `PlanCompras`

```
<?php
// src/Salud/ComprasBundle/Entity/UnidadSolicitante.php
// . . . .
public function __toString()
{
    return $this->nombreunidad;
}
```

```
<?php
// src/Salud/ComprasBundle/Entity/PeriodoFiscal.php
// . . . .
public function __toString()
{
    return (string)$this->aniofiscal;
}
```

```
<?php
// src/Salud/ComprasBundle/Entity/FuenteFinanciamiento.php
// . . . .
public function __toString()
{
    return $this->descripcionfuente;
}
```

```
<?php
    // src/Salud/ComprasBundle/Entity/SubfuenteFinanciamiento.php
    // . . . .
    public function __toString()
    {
        return $this->descripcionesubfuente;
    }
}
```

Vamos a modificar un poco la plantilla, básicamente es agregar las líneas para incluir el layout, poner el código existente dentro del bloque *content* y adecuación de los mensajes (haremos lo mismo con la plantilla *new.html.twig*), la plantilla queda así:

```
{# src/Salud/ComprasBundle/Resources/views/PlanCompras/show.html.twig #}
{% extends 'SaludComprasBundle::layout.html.twig' %}

{% block content %}
<h1>Edición de plan de compras</h1>

<form action="{{ path('plancompras_update', { 'id': entity.id }) }}" method="post"
{{ form_enctype(edit_form) }}>
    {{ form_widget(edit_form) }}
    <p>
        <button type="submit">Guardar cambios</button>
    </p>
</form>
<ul class="record_actions">
    <li>
        <a href="{{ path('plancompras') }}">
            Regresar al listado
        </a>
    </li>
    <li>
        <form action="{{ path('plancompras_delete', { 'id': entity.id }) }}" method="post">
            {{ form_widget(delete_form) }}
            <button type="submit">Borrar</button>
        </form>
    </li>
</ul>
```

```
</li>
</ul>
{% endblock %}
```

Agregar las validaciones

Solo agregaremos una nueva validación al número de plan, que verificará por medio de una expresión regular que el número de plan tenga cuatro dígitos. Además será necesario crear el constructor en la entidad PlanCompras para establecer los valores por defecto.

```
<?php
// src/Salud/PlanCompras/Entity/PlanCompras.php
// . . .
use Symfony\Component\Validator\Constraints as Assert;
// . . .
class PlanCompras
{
    // . . . .
    /**
     * @var string $numeroplan
     *
     * @ORM\Column(name="numeroplan", type="string", length=4, nullable=false)
     * @Assert\Regex(pattern="/^\d{4}$/", message="Debe tener cuatro dígitos")
     */
    private $numeroplan;

    //...

    public function __construct() {
        $this->montoplan = 0;
        $this->autorizado = false;
        $this->enviado = false;
        $this->consolidado = false;
    }
}
```


9. JAVASCRIPT

Gestión del detalle de plan de compras

Ahora trabajaremos con el detalle del plan de compras, se implementará un funcionamiento con ajax para la gestión de cada línea del plan de compras. Esto se hace con JavaScript el cual es un lenguaje del lado del cliente, y vamos a utilizar específicamente el framework jQuery, aunque se puede utilizar cualquier otro, una de las bondades de jQuery es que se puede utilizar de manera no intrusiva (no se mezcla el código JavaScript y HTML) con lo cual se garantiza una mejor separación y limpieza del código.

Primero crearemos el bundle para contener jQuery y demás plugins de éste que utilizaremos, lo haremos con el generador intereractivo

```
app/console generate:bundle
```

- Ingresar *jQuery/jQueryBundle* como namespace del budle
- *jQueryBundle* como nombre del bundle
- Enter para aceptar la ubicación por defecto del bundle.
- Enter para aceptar el formato de configuración por defecto.
- Enter para aceptar que no genera la estructura completa de directorios
- Confirmamos para que proceda a ejecutar las acciones
- Luego confirmamos la actualización del kernel
- Como último paso le diremos que no deseamos que actualice las rutas ya que no las utilizaremos.

Ahora crearemos la carpeta donde se copiarán jQuery y otros plugins, dentro de *src/jQuery/jQueryBundle/Resources/* creamos la carpeta *public* y dentro de esta otra con el nombre *js*.

Ahora vamos al sitio <http://jquery.com/> y descargamos el framework y lo copiamos al directorio *js* recién creado.

Para el manejo del detalle del plan de compras utilizaremos un plugin para jQuery, para el manejo de grids, descargarlo desde http://www.trirand.com/blog/?page_id=6, a menos que se tenga experiencia con este plugin es recomendable dejar marcadas todas las opciones y dar clic en el botón descargar. Descomprimir el archivo descargado en la carpeta *js* recién creada en nuestro proyecto.

Para incluir los archivos que hemos copiado a la carpeta plugin del bundle jQueryBundle, realizamos el siguiente comando:

```
app/console assets:install web --symlink
```

Además vamos a utilizar jQuery.ui que es la librería oficial de jQuery para crear interfaces de usuario, provee mejoras en la interacción con el usuario, componentes, efectos y temas. La incluiremos primero porque más adelante agregaremos la funcionalidad de autocompletar y porque el plugin anterior, jquery.jqGrid, utiliza los temas de jQuery.ui, la descargaremos desde <http://jqueryui.com/download> podemos elegir que funcionalidades deseamos incluir así como también el tema con la cual la descargaremos, he descargado por defecto con todos los componentes y el tema redmon. Al descargarla descomprimirla en la carpeta de nuestro proyecto *src/JavaScript/jQueryBundle/Resources/public/js*

Para agregar el detalle lo vamos a realizar cuando el plan de compras está en la vista *show*, así que vamos a esa plantilla y primero de las acciones dejemos solo el enlace para regresar al listado (quitar las acciones editar y borrar).

Ahora vamos a incluir jQuery y jqGrid dentro de la plantilla, agregamos el siguiente bloque al principio de la plantilla, después de extender el layout

```
{% block javascripts %}  
    {{ parent() }}
```

```

        <script type="text/javascript" src="{{ asset('bundles/jquery/js/jquery-
1.6.2.min.js') }}" ></script>

        <script type="text/javascript" src="{{ asset('bundles/jquery/js/jquery.jqGrid-
4.1.2/js/i18n/grid.locale-es.js') }}" ></script>
        <script type="text/javascript" src="{{ asset('bundles/jquery/js/jquery.jqGrid-
4.1.2/js/jquery.jqGrid.src.js') }}" ></script>
        <script type="text/javascript" src="{{ asset('bundles/jquery/js/jquery-ui-
1.8.15.custom/js/jquery-ui-1.8.15.custom.min.js') }}" ></script>

{% endblock %}

{% block stylesheets%}
    {{ parent() }}
    <link rel="stylesheet" href="{{ asset('bundles/jquery/js/jquery.jqGrid-
4.1.2/css/ui.jqgrid.css') }}" type="text/css" media="all" ></link>
    <link rel="stylesheet" href="{{ asset('bundles/jquery/js/jquery-ui-
1.8.15.custom/css/redmond/jquery-ui-1.8.15.custom.css') }}" type="text/css" media="all"
></link>
{% endblock %}

```

Y para mostrar el grid con el detalle del plan de compras y los botones para su manipulación, agreguemos al final de la plantilla *show.html.twig* del plan de compras, justo antes de la línea *{* endblock*}*

```

<table id="detalle_plan"><tr><td></td></tr></table>
<div id="toolbar1"></div>
<div id='pager' ></div>
<DIV id="botones" >
    <BUTTON id="add_button">Agregar</button>
    <BUTTON id="edit_button">Editar</button>
    <BUTTON id="delete_button">Borrar</button>
</div>

<div id='data_control'>
    <input type="hidden" value="{{front_controller}}" name="front_controller"
id="front_controller" />
    <input type="hidden" value="{{entity.id}}" name="idPlan" id="id_plan" />
</div>

```

Le hemos pasado una variable a la plantilla (*front_controller*) la cual nos

servirá para determinar qué controlador frontal estamos utilizando y este será utilizado en el código JavaScript. Modifiquemos el método *show* del controlador del plan de compras para agregar esa variable. El final del método debe quedar así:

```
// src/Salud/ComprasBundle/Controller/PlanComprasController.php
// ...

public function showAction($id)
{
    // ...

    return array(
        'entity'          => $entity,
        'front_controller' => $request->getRequest()->getScriptName() );
}
```

Creemos el archivo que gestionará el detalle:

```
//src/Salud/PlanCompras/
var grid_aux, total_antes;
$(document).ready(function(){
    $('#data_control').hide();
    $(function(){
        $("#detalle_plan").jqGrid({
            url: $('#front_controller').val() + "/plancompras/" + $("#id_plan").val() +
"/detalle",
            editurl: $('#front_controller').val() + "/plancompras/" + $("#id_plan").val() +
"/detalle/edit",
            datatype: 'json',
            mtype: 'POST',
            colNames: ['iditem', 'Producto', 'Unidad de medida', 'Precio Unitario', 'Cantidad',
'Total'],
            colModel :[
                {name:'iditem', index:'idItem', width:0, editable: true},
                {name:'descripcionitem', index:'descripcionitem', width:450, editable:
true},
                {name:'descripcionunidadmedida', index:'descripcionunidadmedida',
width:120, editable:true},
                {name:'preciounitario', index:'preciounitario', width:100, align:'right',
editable:true},
                {name:'cantidadpedido', index:'cantidadpedido', width:70, align:'right',
```

```

editable: true },
    {name:'total', index:'total', width:90, align:'right', summaryType: 'sum',
formatter:'currency', editable: true}
],
sortname: 'descripcionitem',
sortorder: 'desc',
toolbar: [true,"bottom"],
caption: 'Detalle Plan de compras',
loadComplete: function() {
    var udata = jQuery(this).jqGrid('getGridParam','userData');
    jQuery("#t_detalle_plan").css('text-align','right').html("<B>Total Plan de
compras:<span id='total_plan'>" + udata.total + "</span>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</B>");
    jQuery(this).jqGrid('hideCol',"iditem");
    grid_aux = this;
}
});
});
});
});

```

Hemos creado el archivo que da la funcionalidad del grid, esto tiene poco que ver con Symfony2 así que no explicaré mucho el código JavaScript, podemos obtener mayor detalle en el sitio de jqGrid tenemos la página de demos <http://trirand.com/blog/jqgrid/jqgrid.html> y la documentación oficial <http://www.trirand.com/jqgridwiki/doku.php?id=wiki:jqgriddocs>.

Lo realmente importante es mostrar la manera en que podemos interactuar con Symfony2 y las interfaces creadas con jQuery, o cualquier otro framework de javascript. En base a eso un aspecto de interés es la línea:

```
url: $('#front_controller').val() + "/plancompras/" + $("#id_plan").val() + "/detalle",
```

La cual está llamando a un controlador de Symfony2, se ha precedido del controlador frontal para que pueda funcionar independiente del entorno en que se esté ejecutando. Vamos a crear la acción que atienda esa petición.

```

// src/Salud/ComprasBundle
// . . .
use Symfony\Component\HttpFoundation\Response;

```

```
//. . .

class PlanComprasController extends Controller
{
    /**
     * @Route("/{id_plan}/detalle", name="_plan_compras_detalle", requirements={"id_plan"=
     "\d+"})
     */
    public function detallePlanAction($id_plan) {
        $em = $this->getDoctrine()->getEntityManager();
        $detalle = $em->getRepository('SaludComprasBundle:LineaPlan')
            ->getDetallePlan($id_plan);

        $detalle_array = $detalle->getArrayResult();
        $total_records = count($detalle_array);

        $output = array(); $i = 0; $total_plan = 0;

        foreach ($detalle_array as $row) {
            $output['rows'][$i]['id'] = $row['id'];
            $output['rows'][$i]['cell'] = array($row['descripcionitem'],
                $row['iditem'],
                $row['descripcionunidadmedida'], $row['preciounitario'],
                $row['cantidadPedido'], $row['total']);
            $total_plan += $row['total'];
            ++$i;
        }

        $output['userdata']['total'] = number_format($total_plan, 2, '.', ',');

        return new Response(json_encode($output), 200, array('Content-Type' =>
        'application/json'));
    }
}
```

Lo nuevo en este controlador es que no estamos renderizando una plantilla sino que devolvemos un objeto Response por lo cual es necesario agregar al principio de la clase la línea

```
use Symfony\Component\HttpFoundation\Response;
```

Además hemos utilizado el método *getDetallePlan* para lo cual crearemos una nueva clase repositorio, esta vez para la clase *LineaPlan*, abramos el archivo *src/Salud/ComprasBundle/Entity/LineaPlan.php*, la línea:

```
* @ORM\Entity
```

Debe quedar así:

```
* @ORM\Entity(repositoryClass="Salud\ComprasBundle\Repository\LineaPlanRepository")
```

Y el archivo de la clase repositorio quedará así:

```
<?php
// src/Salud/ComprasBundle/Repository/LineaPlanRepository.php
namespace Salud\ComprasBundle\Repository;
use Doctrine\ORM\EntityRepository;
class LineaPlanRepository extends EntityRepository {
    public function getDetallePlan($id_plan) {
        $dql = "SELECT lp.id, i.descripcionitem, um.descripcionunidadmedida,
                    lp.preciounitario, lp.cantidadPedido,
                    i.id as iditem, (lp.cantidadPedido * lp.preciounitario) AS total
                FROM SaludComprasBundle:LineaPlan lp
                JOIN lp.idUnidadMedida um
                JOIN lp.idItem i
                WHERE lp.idPlanCompras = :id_plan
                ORDER BY i.descripcionitem";
        $query = $this->_em->createQuery($dql);
        $query->setParameter('id_plan', $id_plan);
        return $query;
    }
}
```

Este método obtiene de la base de datos el detalle del plan de compras seleccionado, mediante una consulta DQL que igual que en ocasiones anteriores se ha especificado para evitar hacer más consultas a la base de datos al recuperar los campos de las tablas relacionadas.

Con esto ya podríamos ejecutarlo y obtener algo similar a esto (ya había ingresado unos datos en el detalle).

Plan:
Unidad / Facultad:
Línea de trabajo que
financia:
Origen Financiamiento:

01000000 - 018 - 2011
Contabilidad
Empresa XXXX
Externos

Período Fiscal:
Fuente
Financiamiento:

2011
Donación

Número convenio: ----

Regresar al listado

Detalle Plan de compras

Producto	Unidad de medida	Precio Unitario	Cantidad	Total
Amortiguadores delanteros	UNIDAD	24.66	1	24,66
Comentarios a la Ley de Notariado/ Editorial Jurídica	vial	12.00	1	12,00
D.S. Penicilina 10 UNITS, 50 discos,	vial	4.80	2	9,60
Mesa circular par 6 personas. 1" de espesor con plástico laminado	UNIDAD	218.40	1	218,40
Mesas de trabajo tipo escritorio	UNIDAD	96.00	7	672,00
SHARP, TONER AR-M317	UNIDAD	72.00	2	144,00
Tinta china negra	Frasco	6.00	3	18,00

AGREGAR
GUARDAR CAMBIOS
BORRAR...

Illustration 21: Detalle del plan de compras

Agregar una nueva línea al detalle.

Vamos a agregar la funcionalidad al botón AGREGAR, en el archivo `src/Salud/ComprasBundle/Resources/public/js/PlanCompras.js`, justo antes de la última línea agregar:

```
function calcular_total(){
    $("#total").val(parseFloat($("#cantidadpedido").val()) * parseFloat(
    $("#preciounitario").val()));
}
$("#add_button").click(function(){
    jQuery("#detalle_plan").jqGrid('editGridRow',"new",{
        height:320,
        width: 400,
        reloadAfterSubmit:false
    });
});
```

The screenshot displays a web interface with a modal window titled 'Agregar registro' (Add record) and a table of purchase details. The modal form contains input fields for 'Producto', 'Unidad de medida', 'Precio Unitario', 'Cantidad', and 'Total', along with 'Guardar' (Save) and 'Cancelar' (Cancel) buttons. The table in the background lists items with columns for 'Unidad de medi', 'Precio Unitar', 'Cantidad', and 'Total'. The total for the plan is 1,098.66.

Unidad de medi	Precio Unitar	Cantidad	Total
UNIDAD	24.66	1	24,66
vial	12.00	1	12,00
vial	4.80	2	9,60
ami UNIDAD	218.40	1	218,40
UNIDAD	96.00	7	672,00
UNIDAD	72.00	2	144,00
Frasco	6.00	3	18,00
Total Plan de compras:			1,098.66

Illustration 22: Formulario para el ingreso del detalle del plan de compras

Vamos a dar mayor funcionalidad al formulario para agregar productos al detalle del plan de compras para ellos vamos a utilizar el componente para autocompleado de jQuery.ui, la idea será que cuando el usuario escriba la descripción del producto la aplicación vaya buscando las coincidencias para eso necesitaremos dos cosas, del lado del cliente la función que agregue el comportamiento de autocompletado en el control de la descripción del producto y del lado servidor la función que realice la búsqueda de los productos que coincidan con el patrón ingresado.

En el archivo `src/Salud/ComprasBundle/Resources/public/js/PlanCompras.js`, dentro de la función que maneja el clic en el botón AGREGAR, EDITAR y BORRAR respectivamente, añadiremos el siguiente código después de `reloadAfterSubmit:false`

```
$("#add_button").click(function(){
    // . .
    reloadAfterSubmit:false,
    beforeSend: function (){
        var cache = {}, lastXhr;
```

```
$("#descripcionitem").autocomplete({
    minLength: 2,
    source: function( request, response ) {
        var term = request.term;
        if ( term in cache ) {
            response( cache[ term ] );
            return;
        }

        lastXhr = $.getJSON( $('#front_controller').val() +
"/plancompras/item/buscar", request, function( data, status, xhr ) {
            cache[ term ] = data;
            if ( xhr === lastXhr ) {
                response( data );
            }
        });
    },
    select: function( event, ui ) {
        $("#iditem").val(ui.item.id);
        $("#preciounitario").val( ui.item.precio );

        $("#descripcionunidadmedida").val( ui.item.unidadmedida ).attr("readonly",true);
        $("#cantidadpedido").focus();
        $("#preciounitario").change(function(){
            calcular_total();
        });
        $("#cantidadpedido").change(function(){
            calcular_total();
        });
    },
    search: function(event, ui) {
        $("#idItem").val("");
        $("#preciounitario").val("");
        $("#descripcionunidadmedida").val("");
    }
});
}
});
```

\$

```

});
$("#edit_button").click(function(){
    var gr = jQuery("#detalle_plan").jqGrid('getGridParam','selrow');
    if( gr != null ) jQuery("#detalle_plan").jqGrid('editGridRow',gr,{
        reloadAfterSubmit:false,
        width: 400,
        beforeShowForm: function (){
            $("#preciounitario").change(function(){
                calcular_total();
            });
            $("#cantidadpedido").change(function(){
                calcular_total();
            });
        }
    }); else alert("Por favor elija una fila");
});

$("#delete_button").click(function(){
    var gr = jQuery("#detalle_plan").jqGrid('getGridParam','selrow');
    if( gr != null ) jQuery("#detalle_plan").jqGrid('delGridRow',gr,{
        reloadAfterSubmit:false
    }); else alert("Por favor elija la fila a borrar");
});
});

```

Y del lado del servidor vamos a implementar la acción /item/buscar/

```

<?php
// src/Salud/ComprasBundle/Controller/PlanComprasController.php
// ..
class PlanComprasController extends Controller
{
    // . . .
    /**
     * @Route("/item/buscar", name="_itemBuscar")
     */
    public function buscarItemAction() {
        $q = strtolower($this->get('request')->get('term'));
    }
}

```

```
        if (!$q)
            return;

        $em = $this->get('doctrine')->getEntityManager();
        $busq = $em->getRepository('SaludComprasBundle:Item')->buscarItem($q);
        $items = $busq->getArrayResult();

        $result = array();
        foreach ($items as $row) {
            array_push($result, array("id" => $row['id'],
                "label" => $row['descripcionitem'] . '-' . $row['preciounitario'] . '-' .
                $row['descripcionunidadmedida'],
                "value" => strip_tags($row['descripcionitem']),
                "precio" => $row['preciounitario'],
                "unidadmedida" => $row['descripcionunidadmedida']
            ));
        }

        return new Response(json_encode($result), 200, array('Content-Type' =>
            'application/json'));
    }
}
```

Para realizar la búsqueda se utiliza el método *buscarItem* de la clase *ItemRepository*

```
<?php
// scr/Salud/ComprasBundle/Repository/ItemRepository.php
namespace Salud\ComprasBundle\Repository;

use Doctrine\ORM\EntityRepository;
class ItemRepository extends EntityRepository
{
    public function buscarItem($parameter)
    {
        $sql = "SELECT item.id, item.descripcionitem, item.preciounitario,
                um.descripcionunidadmedida
                FROM SaludComprasBundle:Item item
                JOIN item.idUnidadMedida um
                WHERE
```

```
        LOWER(item.descripcionitem) LIKE :parameter ";
    $query = $this->_em->createQuery($dql);
    $query->setParameter('parameter', '%' . strtolower($parameter) . '% ');
    return $query;
}
}
```

Recordar modificar la clase entidad correspondiente:

```
<?php
//src/Salud/ComprasBundle/Entity/Item.php
// . . .
/**
 * Salud\ComprasBundle\Entity\Item
 *
 * @ORM\Table(name="item")
 * @ORM\Entity(repositoryClass="Salud\ComprasBundle\Repository\ItemRepository")
 */
class Item
{
```

En la siguiente imagen se ve el resultado del autocompletado

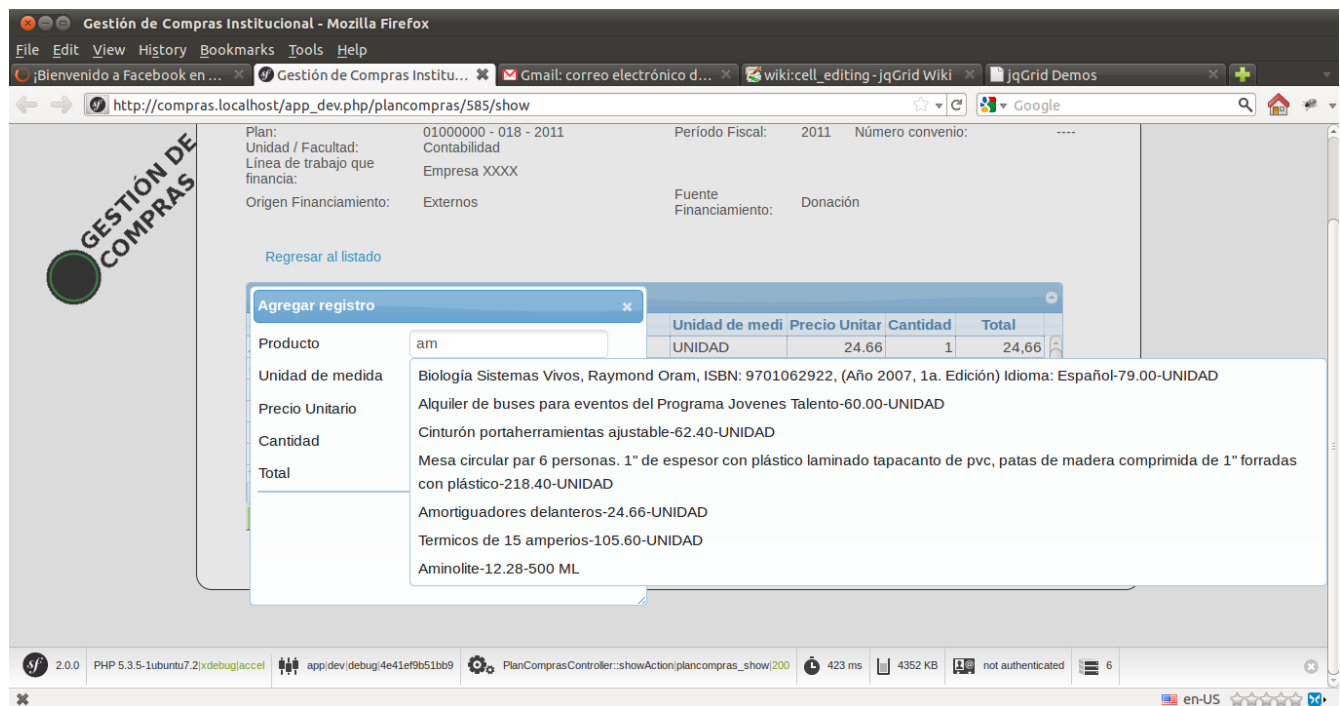


Illustration 23: Fomulario con autocompletado

El formulario ya tiene la funcionalidad requerida del lado cliente ahora vamos a implementar el método para guardar los datos capturados del formulario

```
<?php
// src/Salud/ComprasBundle/Controller/PlanComprasController.php
// ..

use Salud\ComprasBundle\Entity\LineaPlan;

class PlanComprasController extends Controller
{
    // . . .
    /**
     * @Route("/plan/compras/{id_plan}/detalle/edit", name="_planComprasDetalle"),
     */
    public function editDetallePlanAction($id_plan) {
        $em = $this->getDoctrine()->getEntityManager();

        $datos = $this->getRequest();
```

```
$ln_plan = ($datos->get('oper') == 'add') ? new LineaPlan():
    $ln_plan = $em->find("SaludComprasBundle:LineaPlan", $datos->get('id'));

$errorList = array();
if ($datos->get('oper') != 'del') {
    $item = $em->find('SaludComprasBundle:Item', $datos->get('iditem'));
    $plan = $em->find('SaludComprasBundle:PlanCompras', $id_plan);

    $ln_plan->setIdPlanCompras($plan);
    $ln_plan->setIdItem($item);
    $ln_plan->setPreciounitario($datos->get('preciounitario'));
    $ln_plan->setIdUnidadMedida($item->getIdUnidadMedida());
    $ln_plan->setCantidadPedido($datos->get('cantidadpedido'));

    $validator = $this->get('validator');
    $errorList = $validator->validate($ln_plan);
}

if (count($errorList) > 0) {
    $rsp = new Response();
    $rsp->setStatusCode(406, $errorList);
    return $rsp;
} else {
    ($datos->get('oper') == 'del') ? $em->remove($ln_plan) :
        $em->persist($ln_plan);
    $em->flush();
}

return new Response('', 200, array('Content-Type' => 'application/json'));
}
```

Y para agregar la funcionalidad a los botones de editar y borrar agregamos esto

10. SEGURIDAD

La seguridad es un proceso de dos etapas, cuyo objetivo es evitar que un usuario acceda a un recurso al que no debería tener acceso.

En el primer paso del proceso, el sistema de seguridad identifica quién es el usuario obligándolo a presentar algún tipo de identificación. Esto se llama **autenticación**, y significa que el sistema está tratando de averiguar quién eres.

Una vez que el sistema sabe quien eres, el siguiente paso es determinar si deberías tener acceso a un determinado recurso. Esta parte del proceso se llama **autorización**, y significa que el sistema está comprobando para ver si tienes suficientes privilegios para realizar una determinada acción.

Ejemplo básico: Autenticación HTTP

Vamos a configurar la aplicación para que solo los usuarios autenticados puedan ingresar. La siguiente configuración le dice a Symfony que proteja cualquier URL de nuestra aplicación y pida al usuario sus credenciales mediante autenticación HTTP básica (es decir, el cuadro de dialogo a la vieja escuela de nombre de usuario/contraseña):

```
# app/config/security.yml
security:
  firewalls:
    secured_area:
      pattern: ^/
      anonymous: ~
      http_basic:
        realm: "Area Segura"
  access_control:
    - { path: ^/plancompras/*, roles: 'ROLE_USER' }
    - { path: ^/producto/*, roles: 'ROLE_ADMIN' }
```

```
providers:
  in_memory:
    users:
      gestor: { password: universo, roles: 'ROLE_USER' }
      admin: { password: admin, roles: 'ROLE_ADMIN' }

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

El resultado final de esta configuración es un sistema de seguridad totalmente funcional que tiene el siguiente aspecto:

- Hay dos usuarios en el sistema (gestor y admin);
- Los usuarios se autentican a través de la autenticación HTTP básica del sistema;
- Todas la URL están protegidas, y es necesario autenticarse para acceder a ellas;
- Para las rutas /producto/* se necesita el rol de administrador y para rutas /plandecompras/* el rol de usuario;

Veamos brevemente cómo funciona la seguridad y cómo entra en juego cada parte de la configuración.

Cómo funciona un la seguridad: autenticación y autorización

Firewall (autenticación)

Cuando un usuario hace una petición a una URL que está protegida por el cortafuegos, se activa el sistema de seguridad. El trabajo del cortafuegos es determinar si el usuario necesita estar autenticado, y si lo hace, enviar una respuesta al usuario para iniciar el proceso de autenticación.

Un cortafuegos se activa cuando la URL de una petición entrante concuerda con el patrón de la expresión regular configurada en el valor '*config*' del cortafuegos. En este ejemplo el patrón (^/) concordará con cada petición entrante. El hecho de que el cortafuegos esté activado no significa, sin embargo, que el nombre de usuario de autenticación HTTP y el cuadro de diálogo de la contraseña se muestre en cada URL.

Esto funciona en primer lugar porque el cortafuegos permite usuarios anónimos a través del parámetro de configuración `anonymous`. En otras palabras, el cortafuegos no requiere que el usuario se autentique plenamente de inmediato, sólo se hará para la cada URL especificada en la sección `access_control`.

Control de acceso (autorización)

Si un usuario solicita `/producto/list`, el proceso se comporta de manera diferente. Esto se debe a la sección de configuración `access_control` la cual dice que cualquier URL coincidente con el patrón de la expresión regular `^/admin` (es decir, `/admin` o cualquier cosa coincidente con `/admin/*`) requiere el rol `ROLE_ADMIN`.

Los roles son la base para la mayor parte de la autorización: el usuario puede acceder a `/producto/list` sólo si cuenta con el rol `ROLE_ADMIN`.

Vamos a detallar los permisos para los productos permitiendo a los usuario consultar el listado de productos y agregar nuevos, pero que no puedan editarlos.

Primero quitemos la línea general de `app/config/security.yml` - `{ path: ^/producto/*, roles: 'ROLE_ADMIN' }` y los permisos ser harán ahora por controlador de la siguiente forma:

Primero agregamos al inicio del archivo `src/Salud/ComprasBundle/Controller/ProductoController.php`

```
use JMS\SecurityExtraBundle\Annotation\Secure;
```

Ahora en el mismo archivo, en cada acción definimos los permisos necesarios, vamos a habilitar las acciones de listar y mostrar productos para todos los usuario que se hayan autenticado y la edición sólo para el rol *ROLE_ADMIN*:

```
use \Symfony\Component\Security\Core\Exception\AccessDeniedException;

class ProductoController extends Controller {
    // . . .
    /**
     * @Secure(roles="IS_AUTHENTICATED_FULLY")
     */
    public function listAction() {
        // . . . . .

        /**
         * @Secure(roles="IS_AUTHENTICATED_FULLY")
         */
        public function showAction() {
            // . . . . .

            public function editAction($id=null) {
                if (!$this->get('security.context')->isGranted('IS_AUTHENTICATED_FULLY'))
                    throw new AccessDeniedException();
            }
        }
    }
}
```

Observar que para la acción *editAction* la verificación de la autorización se realiza dentro, esto es porque esta acción tiene dos rutas (*/producto/edit/{id}* y */producto/create*) en una el parámetro *id* es opcional y en la otra obligatorio, y la anotación *@Secure* no maneja de manera adecuada las dos rutas.

Si deseamos realizar verificaciones con más detalle, por ejemplo que un usuario pueda ingresar un producto pero que se necesite el ROL_ADMIN para editarlo, podemos acceder al contexto de seguridad como un servicio; cambiemos primero los permisos para la acción editar en general (recordar que esta acción también sirve para agregar productos) y luego dentro del código verificaremos los permisos para cuando quiera editar un producto.

```
public function editAction($id=null) {  
    if (!$this->get('security.context')->isGranted('IS_AUTHENTICATED_FULLY'))  
        throw new AccessDeniedException();  
  
    $em = $this->getDoctrine()->getEntityManager();  
  
    if (isset($id)) {  
        if (!$this->get('security.context')->isGranted('ROLE_ADMIN'))  
            throw new AccessDeniedException();  
    }  
}
```

Lista de control de acceso (ACL): Protegiendo objetos individuales de la base de datos.

En aplicaciones complejas, a menudo te enfrentas al problema de que las decisiones de acceso no se pueden basar únicamente en la persona (Ficha) que está solicitando el acceso, sino también implica un objeto dominio al cual se está solicitando acceso. Aquí es donde entra en juego el sistema ACL.

Vamos a suponer que deseamos que el usuario puede agregar un nuevo producto pero que también pueda editar aquellos que él haya agregado, además el usuario con rol de administrador podrá editar todos los productos.

Primero vamos a configurar los permisos para el patrón */producto/** para que cualquier usuario autenticado pueda acceder, y quitamos las anotaciones *@Secure* de las acciones del controlador *ProductoController*

```
# app/config/security.yml  
# . . .  
access_control:  
    - { path: ^/plancompras/*, roles: 'ROLE_USER' }  
    - { path: ^/producto/*, roles: 'IS_AUTHENTICATED_FULLY' }
```

Configuración inicial de ACL

Ahora, antes de que finalmente puedas entrar en acción, tenemos que hacer algún proceso de arranque. En primer lugar, tenemos que configurar la conexión al sistema ACL que se supone vamos a emplear:

```
# app/config/security.yml
security:
  acl:
    connection: default
```

Después de configurar la conexión, tenemos que importar la estructura de la base de datos. En la consola escribimos:

```
php app/console init:acl
```

Ahora vamos a implementar ACL

Para esto después de guardar un producto asignaremos los permisos adecuados:

```
<?php
// . . .
use \Symfony\Component\Security\Acl\Domain\ObjectIdentity;
use \Symfony\Component\Security\Acl\Domain\UserSecurityIdentity;
use \Symfony\Component\Security\Acl\Permission\MaskBuilder;

class ProductoController extends Controller {
    // . . .
    public function editAction($id=null) {
        // . . .
        if ($form->isValid()) {
            //Guardar el objeto (aún no en la base de datos)
            $em->persist($producto);
            //Escribir los cambios a la base de datos
            $em->flush();

            if ($id == null) {
                // creando la ACL
                $aclProvider = $this->get('security.acl.provider');
                $objectIdentity = ObjectIdentity::fromDomainObject($producto);
```

```
$acl = $aclProvider->createAcl($objectIdentity);

// recupera la identidad de seguridad del usuario registrado actual
$securityContext = $this->get('security.context');
$user = $securityContext->getToken()->getUser();
$securityIdentity = UserSecurityIdentity::fromAccount($user);

// otorga permiso de propietario
$acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
$aclProvider->updateAcl($acl);
}
// . . .
```

En primer lugar, te habrás dado cuenta de que `->createAcl()` no acepta objetos de dominio directamente, sino sólo implementaciones de `ObjectIdentityInterface`. Este paso adicional de indirección te permite trabajar con ACL, incluso cuando no tienes a mano ninguna instancia real del objeto dominio. Esto será muy útil si deseas comprobar los permisos de un gran número de objetos sin tener que hidratar estos objetos.

La otra parte interesante es la llamada a `->insertObjectAce()`. En nuestro ejemplo, estamos otorgando al usuario que ha iniciado sesión acceso de propietario al comentario. La `MaskBuilder::MASK_OWNER` es una máscara predefinida de bits enteros; no te preocupes que el constructor de la máscara debe abstraer la mayoría de los detalles técnicos, pero gracias a esta técnica puedes almacenar muchos permisos diferentes en la fila de la base de datos lo cual nos da un impulso considerable en cuanto a rendimiento.

Comprobando el acceso

```
// src/Salud/ComprasBundle/Controller/ProductoController.php
public function editAction($id=null) {
    $em = $this->getDoctrine()->getEntityManager();
    if (isset($id)) {
```

```
$producto = $em->find('SaludComprasBundle:Item', $id);  
if (!$producto)  
    throw $this->createNotFoundException("Producto no encontrado");  
$securityContext = $this->get('security.context');  
// comprueba el acceso para edición  
if (false === $securityContext->isGranted('EDIT', $producto)) {  
    throw new AccessDeniedException('Acceso denegado');  
}  
}
```


11. BUNDLES

Ya hemos mencionado que en Symfony2 todo es un bundle, hay una parte particular que nos interesará, en symfony 1.x disponíamos de una gran cantidad de plugins que ofrecían variadas y útiles funcionalidades ahora en lugar de plugins tenemos bundles, por el momento se llevan más de 500 bundles creados, <http://symfony2bundles.org/>

FOSUserBundle

Provee funcionalidades para la autenticación en proyectos de Symfony2.

El FOSUserBundle agregar soporte para un sistema de autenticación de usuarios utilizando base de datos. Provee un flexible framework para la gestión de los usuarios, que tiene como objetivo la realización de tareas comunes como el inicio de sesión, recuperar claves y el registro de usuarios.

Agregar instalación alternativa con git

1. descargamos el bundle <https://github.com/FriendsOfSymfony/FOSUserBundle/blob/master/Resources/doc/index.md>
2. Crear la estructura de directorios *FOS/UserBundle* dentro de *src/vendor/bundles*
3. Descomprimir el bundle descargado y su contenido copiarlo a la carpeta *UserBundle* recién creada
4. Agregar el namespace al autoloader.php

```
<?php
// app/autoload.php

$loader->registerNamespaces(array(
    // ...
    'FOS' => __DIR__.'/../vendor/bundles',
```

```
));
```

5. Habilitar el bundle

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new FOS\UserBundle\FOSUserBundle(),
    );
}
```

6. Crear la clase para el manejo de los usuario

El objetivo de este bundle es persistir alguna clase para el manejo de usuario a la base de datos. Tu primer trabajo, es crear dicha clase (en nuestro caso ya la tenemos creada se llama **Usuario**) Esta clase puede tener la estructura y comportamiento que se desee: agregar cualquier propiedad o método que se utilice al fin de cuentas en **tu** clase para el manejo de usuarios.

Esta clase debe tener sólo dos requerimientos:

1. Debe extender la clase **BaseUser**.
2. Debe tener un campo **id**
3. Ver la clase del bundle para no duplicar campos

En nuestro caso, sólo debemos agregar las líneas marcadas en negrita:

```
<?php
//src/Salud/ComprasBundle/Entity/Usuario.php
namespace Salud\ComprasBundle\Entity;

use FOS\UserBundle\Entity\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;

/**
 * Salud\ComprasBundle\Entity\Usuario
```

```
*
* @ORM\Table(name="usuario")
* @ORM\Entity
*/
class Usuario extends BaseUser
{
    // . . .
    protected $id;
```

7. Configurar la seguridad de la aplicación

```
# app/config/security.yml
security:
    providers:
        fos_userbundle:
            id: fos_user.user_manager

    firewalls:
        main:
            pattern: ^/
            form_login:
                provider: fos_userbundle
            logout:      true
            anonymous:    true

    access_control:
        - { path: ^/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/register, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/resetting, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/admin/, role: ROLE_ADMIN }

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: ROLE_ADMIN
```

8. Configurar el FOSUserBundle

```
# app/config/config.yml
fos_user:
    db_driver: orm # other valid values are 'mongodb', 'couchdb'
    firewall_name: main
    user_class: Salud\ComprasBundle\Entity\Usuario
```

9. Importar los archivos de rutas de FOSUserBundle

```
# app/config/routing.yml
fos_user_security:
```

```
resource: "@F0SUserBundle/Resources/config/routing/security.xml"
```

```
fos_user_profile:
```

```
resource: "@F0SUserBundle/Resources/config/routing/profile.xml"  
prefix: /profile
```

```
fos_user_register:
```

```
resource: "@F0SUserBundle/Resources/config/routing/registration.xml"  
prefix: /register
```

```
fos_user_resetting:
```

```
resource: "@F0SUserBundle/Resources/config/routing/resetting.xml"  
prefix: /resetting
```

```
fos_user_change_password:
```

```
resource: "@F0SUserBundle/Resources/config/routing/change_password.xml"  
prefix: /change-password
```

10. Actualizar el esquema de la base de datos

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

Esto es según la documentación del bundle pero debido a restricciones en la base de datos realizaremos en su lugar:

```
app/console doctrine:schema:update --dump-sql
```

y copiaremos las líneas que afectan a la tabla usuario para pegarlas dentro de la base de datos

11. d

Índice de tablas

Tabla 1: Tabla de ejemplo.....	5
--------------------------------	---

Índice de ilustraciones

Illustration 1: Salida en una función mal escrita (Sin XDebug).....	10
Illustration 2: Salida de la función var_dump aplicada sobre un arreglo (Sin XDebug).....	10
Illustration 3: Salida de una función mal escrita (Con XDebug).....	10
Illustration 4: Salida de la función var_dump aplicada sobre un arreglo (Con XDebug).....	11
Illustration 5: Salida del script check.php.....	17
Illustration 6: Pantalla de inicio en entorno de producción.....	30
Illustration 7: Pantalla de inicio en un entorno de depuración.....	32
Illustration 8: Barra de depuración.....	32
Illustration 9: Botón de configuración inicial.....	33
Illustration 10: Inicio de la configuración.....	34
Illustration 11: Cionfigurar la protección contra ataques CSRF.....	35
Illustration 12: Salida del formulario.....	61
Illustration 13: Salida de listados de productos.....	74
Illustration 14: Formulario de edición de productos.....	74