maestros del web

guía Zend

El Framework abierto para el desarrollo de aplicaciones y servicios web

CATEGORIA
PROGRAMACIÓN

NIVEL
INTERMEDIO

Guía Zend:
El Framework abierto para el desarrollo de aplicaciones y servicios web

Versión 1 / mayo 2011

Nivel: Básico / Intermedio

La Guía Zend se encuentra en línea en:

http://www.maestrosdelweb.com/editorial/guia-zend/

Un proyecto de Maestros del Web

- Edición: Stephanie Falla Aroche
- Diseño y diagramación: Iván E. Mendoza
- Autor: Rodrigo Souto

Contacto

http://www.maestrosdelweb.com/sitio/correo/

Redes sociales

Facebook: http://www.facebook.com/maestrosdelweb
Twitter: http://www.twitter.com/maestros

maestros del web

Rodrigo Souto

Argentino, estudiante de Ingeniería en Sistemas en la UTN cuenta con amplia experiencia en lenguajes de programación como PHP, JavaScript, CSS fanático de Zend Framework y MooTools.

Índice

Capítulo 1

Introducción y primera aplicación

Capítulo 1

Introducción y primera aplicación

Zend Framework es un framework[1] open source para PHP desarrollado por Zend la empresa encargada de la mayor parte de las mejoras hechas a PHP. Zend Framework implementa el patron MVC es 100% orientado a objetos, sus componentes tienen un bajo acoplamiento es posible usarlos de forma independiente y brinda un estándar de codificación.

Características:

- Cuenta con soporte para internalización y localización de aplicaciones construir sitios multi-idioma, convertir formatos de fechas, monedas, etc.
- Facilita el setup y brinda herramientas para crear la estructura de directorios y clases por línea de comandos.
- Integración con PHPUnit por medio de Zend_Test para facilitar el testing de la aplicación.
- Tiene adapters para diversos tipos de bases de datos, brinda componentes para la autenticación y autorización de usuarios, envío de mails, cache en varios formatos, creación de web services, etc.

La idea es invertir menos tiempo en el desarrollo y hacer uso de componentes ya testeados.

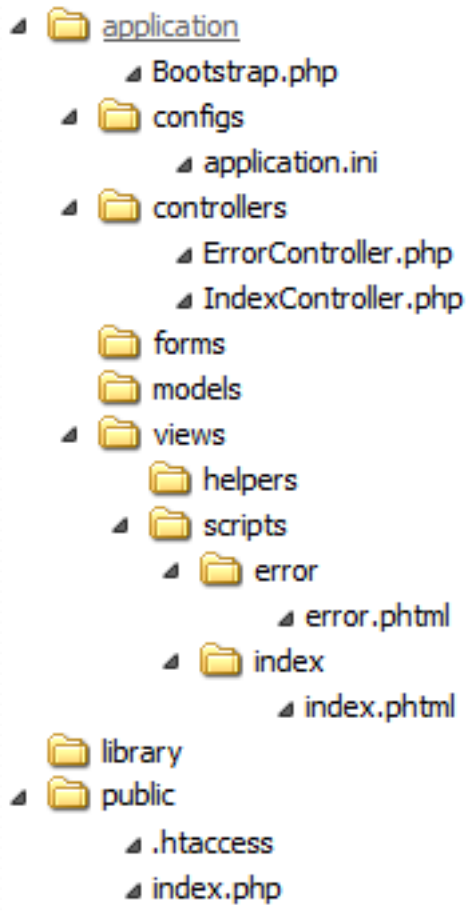
Instalación

1. Descarga[2] la última versión de Framework desde el sitio oficial.
2. Cuando la tengas descargada creamos una estructura de directorios, puedes crearla automáticamente con Zend_Tool_Framework, o puedes hacerlo manualmente.

La estructura inicial debe quedar de la forma que se ve en la imagen de la siguiente página.

1 http://www.maestrosdelweb.com/editorial/los-frameworks-de-php-agilizan-tu-trabajo/
2 http://framework.zend.com/download/latest

Si te fijas en la figura tenemos unos archivos básicos para crear y lo haremos paso a paso.



Action Controllers

Los controladores son clases que extienden de Zend_Controller_Action cada controlador tiene unos métodos especiales cuya nombre tiene el sufijo “Action” y denominados “action methods”.

Por default las URLs en Zend Framework son del tipo */controlador/action* es decir que si en nuestro IndexController tenemos un “pruebaAction” lo podremos ejecutar desde /index/prueba.

Ejemplo: www.maestrosdelweb.com/index/prueba

Código fuente

```
class IndexController extends Zend_Controller_Action
{
public function init()
{
}
public function indexAction()
{
 $this->view->mensaje = 'Primera aplicación con Zend Framework!';
}
}
```

- La estructura es simple en el método *init()* se agregan tareas de inicialización y creamos un action llamado index.
- Al ejecutarlo, Zend Framework automáticamente relaciona el nombre del action con una vista que será renderizada.
- Esta vista tiene la extensión phml en el caso del indexAction la vista asociada será index.phtml dentro de la carpeta correspondiente al controlador index en *views/scripts*.

Ejemplo: /views/scripts/{controlador}/{action}.phtml

En el *indexAction* asignamos un texto a una variable de la vista, y en el archivo index.phtml lo mostramos haciendo *echo $this->mensaje*.

ErrorController

El controlador será ejecutado cada vez que se quiera llamar una página que no existe (error 404) o se produzca algún error en la aplicación (error 500).

Código fuente

```
class ErrorController extends Zend_Controller_Action
{
 public function errorAction()
 {
  $errors = $this->_getParam('error_handler');
  switch ($errors->type) {
   case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
   case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
   case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
    $this->getResponse()->setHttpResponseCode(404);
    $this->view->message = 'Pagina no encontrada';
    break;
   default:
    $this->getResponse()->setHttpResponseCode(500);
    $this->view->message = 'Ocurrio un error inesperado';
    break;
  }
 }
}
```

Configurando el htaccess y el index.php

En *htaccess*: aplicamos el patrón Front Controller y redirigir todas las peticiones al *index.php* luego decidir la página para mostrar.

```
RewriteEngine On
RewriteRule !\.(js|ico|txt|gif|jpg|png|css|pdf)$ index.php
```

El siguiente paso es crear el index:

```
// Definimos la ruta de /application
defined('APPLICATION_PATH')
|| define('APPLICATION_PATH',
realpath(dirname(__FILE__) . '/../application'));
// El entorno de trabajo actual
defined('APPLICATION_ENV')
|| define('APPLICATION_ENV',
(getenv('APPLICATION_ENV') ? getenv('APPLICATION_ENV')
: 'production'));
// Configuramos el include path, es decir los directorios donde estarán nuestros
archivos
$rootPath = dirname(__FILE__)."/..";
set_include_path($rootPath . '/application/config' . PATH_SEPARATOR . $rootPath
. '/library/');
// Zend_Application
require_once 'Zend/Application.php';
// Creamos la aplicacion
$application = new Zend_Application(
APPLICATION_ENV,
APPLICATION_PATH . '/configs/application.ini'
);
$application->bootstrap()->run();
```

1. Creamos dos constantes (APPLICATION_PATH y APPLICATION_ENV)
2. Configuramos el include path
3. Creamos nuestra instancia de Zend_Application y le damos run.

Bootstrap y application.ini

En el index hacemos referencia al archivo *application.ini* que configura el sitio pero que aún no creamos pero haremos algo simple:

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
; bootsrap
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
; frontController
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
; layout
resources.layout.layoutPath = APPLICATION_PATH "/layouts"
; Database
resources.db.adapter = "pdo_mysql"
resources.db.params.host = "localhost"
resources.db.params.username = "user"
resources.db.params.password = "pass"
resources.db.params.dbname = "dbname"
[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
[testing : production]
```

Ahora, creamos el *Bootstrap* registrando en el *autoloader* el *namespace* de nuestra aplicación para instanciar las clases que usemos sin necesidad de hacer antes un include de dicho archivo:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
 protected function _initAutoloader() {
 $autoloader = Zend_Loader_Autoloader::getInstance();
 $autoloader->registerNamespace('App_')->setFallbackAutoloader(true);
 $resourceAutoloader = new Zend_Loader_Autoloader_Resource(
  array(
   'basePath' => APPLICATION_PATH,
   'namespace' => 'App',
   'resourceTypes' => array(
    'form' => array('path' => 'forms/', 'namespace' => 'Form'),
    'model' => array('path' => 'models/', 'namespace' => 'Model')
   )
  )
 );
}
}
```

Seteando un layout

Es momento de crear el layout del sitio dentro de la carpeta "layouts" nombramos el archivo layout.phtml:

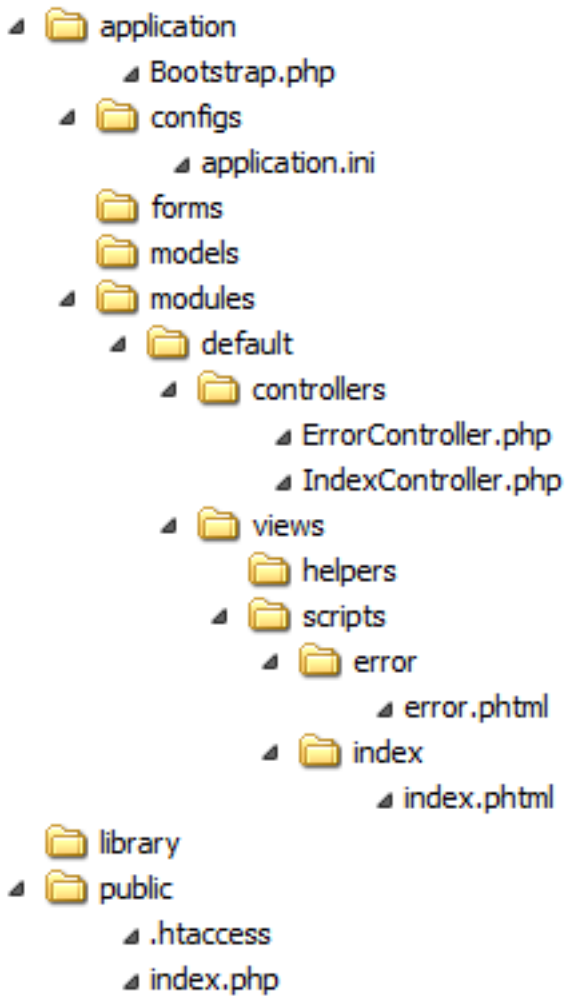
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Primera aplicacion en Zend Framework>/title>
</head>
<body>
   <?php
   echo $this->layout()->content;
   ?>
```

```
</body>
</html>
```

Prueba en: sitio.com, sitio.com/index, sitio.com/index/index en todos ellos se verá el mensaje de bienvenida.

Creando una aplicación modular

Otra organización de directorios muy usada es la siguiente:



Tenemos nuestro sitio dividido en módulos con sus propios controladores y vistas. Para realizar este cambio debemos cambiar la línea:

```
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
```

Por la siguiente línea:

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/modules"
```

- Ahora tenemos el sitio organizado en módulos.
- Default es el módulo por defecto, creamos el directorio modules con un subdirectorio correspondiente al modulo default y copiamos las carpetas views y controller.

- Podemos acceder otra vez, teniendo en cuenta que las urls ahora serán del tipo www.maestrosde-lweb.com/modulo/controlador/action.

Puedes verlo funcionar desde:

- maestrosdelweb.com
- maestrosdelweb.com/default
- maestrosdelweb.com/default/index
- maestrosdelweb.com/default/index/index

Es importante aclarar que en cuanto a controladores usando la estructura de directorios llevan como prefijo el nombre del módulo al que pertenecen. El controlador `Usuarios` del modulo Alta se llamara `Alta_UsuariosController`. Esto no cuenta para el modulo default, los controladores ubicados allí no llevan ningún prefijo especial.

En este punto hemos terminado con la instalación.

Capítulo 2

Modelos y Zend_DB

Capítulo 2

Modelos y Zend_DB

El modelo es el que se encarga de la lógica de negocio y de procesar datos: inserts, updates y deletes. Sin embargo, me arriesgaré a la critica afirmando que todo proceso de datos va en el modelo, hay una regla en inglés que dice “skinny controllers, fat models”.

La idea es que el controlador sólo funcione como intermediario entre la vista y el modelo cuando sea necesario, que no realice ninguna otra actividad, que toda la lógica este contenida en el modelo. De esta manera, tenemos nuestro código mucho más organizado y limpio.

Dentro de un sistema MVC, los modelos contienen la lógica de negocio se encargan de la lectura/escritura de la información, así que estaremos trabajando con ellos de forma frecuente. El controlador puede comunicarse con los modelos para pedir y modificar datos, mientras que la vista puede leer datos pero no puede escribir.

Zend_Db_Select

Nos provee una API orientada a objetos para crear SQL SELECT’s permite setear cada parte del query, no es obligatorio usarlo en nuestros modelos. Si es una consulta simple escribimos el select normalmente en consultas más complejas será de gran ayuda.

La forma más fácil de crear un select object es con el método select() del data base adapter:

```
$select = $dbAdapter->select();
```

Veamos los métodos más usados:

Código fuente

```
// setear el from
$select->from(“miTabla”, “estaColumna”);
// setear el from asignandole un alias a la tabla y especificando que campos
queremos pedir
$select->from(array(“esteAlias” => “estaTabla”), array(“esteCampo”,
“yEsteOtro”));
```

```
// agrega una clausula AND al where
$select->where("nombre = ?", $nombre);
$select->where("activo = 1");
// agrega una clausula OR
$select->orWhere("activo = 2");
$select->group("pais");
$select->joinLeft("otraTabla", "otraTabla.id = tabla.id");
$select->limit($count, $offset);
$select->order("id DESC");
```

Por último, haciendo *echo $select*; obtenemos la consulta SQL correspondiente. Sin embargo, el 99% al armar un select estareos dentro de un modelo, así que obtenemos una instancia con el método *select()* de *Zend_Db_Table_Abstract*:

```
// en un modelo
$select = $this->select();
```

Este select no será un *Zend_Db_Select* sino una versión especial llamada *Zend_Db_Table_Select*, sin embargo todos los métodos nombrados anteriormente funcionan con un par de agregados.

Por un lado, automáticamente setea el *from()* adecuado según la tabla del modelo actual. Por defecto no permite realizar joins a otras tablas si lo hacemos dara un error, para evitar esto deberemos cambiar este comportamiento con el método *setIntegrityCheck()*:

```
$select = $this->select()->setIntegrityCheck(false)->joinLeft(...);
```

Hecho este rápido repaso a *Zend_Db_Select* explicaré un poco de *Zend_Db_Expr* supongamos que hacemos esto:

```
$select->from($tabla, array("COUNT(*)", "nombre", "pais"));
```

A simple vista esta bien, pero cuando Zend arma el select se da cuenta que "*COUNT(*)*" no es un campo correspondiente a la tabla por eso genera un error. Para este caso hacemos uso de *Zend_Db_Expr*:

```
$count = new Zend_Db_Expr("COUNT(*)");
$select->from($tabla, array($count, "nombre", "pais"));
```

Otra forma de hacerlo es escribir la expresión entre paréntesis y Zend automáticamente lo convertirá a una *Zend_Db_Expr*:

```
$select->from($tabla, array("(COUNT(*))", "nombre", "pais"));
```

Cuando usemos un subquery en algún select deberemos pasarlo como una *Zend_Db_Expr*.

Zend_Db_Table

Implementando el patrón Table Data Gateway las diferentes tablas de la base de datos son representadas por *Zend_Db_Table*. Por cada tabla tenemos un modelo que extienda de *Zend_Db_Table_Abstract* que nos brinda diferentes métodos para realizar INSERT's, UPDATE's, y demás operaciones.

Supongamos que tenemos nuestra tabla "mdw_usuarios", cuyo primary key es "id_usuario", nuestro modelo quedaría de la siguiente manera:

Código fuente

```
class Mdw_Model_Usuarios extends Zend_Db_Table_Abstract {
protected $_name = "mdw_usuarios";
protected $_primary = "id_usuario";
}
$usuarios = new Mdw_Model_Usuarios;</pre></div>
<p>Especificando los atributos $_name y $_primary ya podemos trabajar sobre
dicha tabla. Si no completamos $_primary, ZF tratará de buscarlo. Si la tabla
no
tiene una primary key, no puede ser usada con Zend_Db_Table.</p>
<h3>Operaciones básicas</h3>
<div class="codigo"><pre>$data = array("nombre" => $nombre, "apellido" =>
$apellido, "email" => $email);
$usuarios->insert($data);
```

Crear un *insert* es simple: le pasamos como parámetro un array cuyas claves son los nombres de los campos con su correspondiente valor. Realizar un *update* es muy parecido solo que tenemos un segundo parámetro para especificar el where:

Código fuente

```
$data = array("email" => $nuevoEmail);
$where = "id_usuario = 2";
$usuarios->update($data, $where);</pre></div>
<p>Por su parte para realizar un delete debemos especificar un where con los
campos a eliminar:</p>
<div class="codigo"><pre>$where = "id_usuario = 3";
$usuarios->delete($where);
```

Obteniendo datos de la base de datos:

Son varias las formas para leer los datos de la base de datos, la primera es con el método find(), que busca filas por su primary key, ese es el único parámetro que tenemos que enviar.

Por ejemplo: para leer la fila correspondiente a id_usuario = 1:

Código fuente

```
$rows = $usuarios->find(1);</pre></div>
<p>También podemos obtener datos de más de un usuarios, enviando un array:</p>
<div class="codigo"><pre>// id_usuario = 1, 2, 3
$rows = $usuarios->find(array(1, 2, 3));</pre></div>
<p>Find() devuelve un objeto del tipo Zend_Db_Table_Rowset_Abstract sobre el
que
podemos trabajar directamente, por ejemplo lo podemos recorrer con un
foreach:</p>
<div class="codigo"><pre>foreach ($rows as $row) {
echo $row->nombre . "<br />";
}
```

También podemos obtener los datos en un Array PHP:

```
$arrayRows = $rows->toArray();
```

La segunda forma de leer información es con el método *fetchAll ($where, $order, $count, $offset)* ningún parámetro es obligatorio, es decir que haciendo *$model->fetchAll()* obten-

dremos todos los registros de esa tabla.

```
$model->fetchAll("id_pais = 3"); // todos los usuarios cuyo id_pais = 3
$model->fetchAll($select, "apellido ASC", 10); // 10 usuarios cuyo ud_pais = 3,
ordenados por apellido, podemos pasar un Zend_Db_Select en el where
```

El *fetchAll()* también nos devuelve un *rowset*.

Zend_Db_Table_Rowset y Zend_Db_Table_Row

- Un Row corresponde a una fila de la base de datos.
- Un Rowset es un conjunto de Rows, representa un conjunto de filas de la base de datos.

Al hacer una consulta, tanto *find()* como f*etchAll()* devuelven un *Rowset* con los resultados obtenidos, y al recorrer ese *Rowset* cada elemento es un *Row*. Aprendimos que el *Rowset* lo podemos recorrer con un *foreach* convertirlo en array.

En cuanto al *Row* veremos algunas cosas más. Suponiendo que tenemos una columna llamada "id_usuario", podremos acceder a su valor de la siguiente manera:

```
$row->id_usuario;
```

Un *Row* implementa el patrón Row Data Gateway con el que podemos crear, eliminar y actualizar datos directamente:

Código fuente

```
$row->delete(); // elimina el registro de la bdd
$row->nombre_usuario = 'Pepe'; // modificamos el nombre de usuario
$row->save(); // y guardamos los cambios
// para crear un nuevo registro
$newRow = $this->createRow();
$this->nombre = "Usuario nuevo";
$this->edad = 14;
$this->save();
```

Capítulo 3

Controladores, Front Controller Plugins y Action Helpers

Capítulo 3

Controladores, Front Controller Plugins y Action Helpers

Como vimos en el primer artículo el controlador responde a las acciones del usuario se comunica con los modelos y la vista porque en *Zend_Framework* los controladores son clases que extienden de *Zend_Controller_Action* y pueden tener uno o más action methods.

Tareas de inicialización

Zend_Controller_Action tiene un método especial llamado *init()*, que es el último método que se ejecuta en el constructor, en el que podemos poner todo lo que deseemos para personalizar la instanciación del controlador.

Por ejemplo, si queremos setear cierta información sobre el usuario actual:

Código fuente

```
class IndexController extends Zend_Controller_Action {
protected $_idUsuario;
// este método se ejecuta siempre, por lo que en cualquier action podremos
hacer uso de $this->_idUsuario
// para acceder al id del usuario actualmente logueado
public function init() {
$this->_idUsuario = Zend_Auth::getInstance()->getIdentity()->id_usuario;
}
}
```

Métodos preDispatch() y postDispatch()

Los métodos *preDispatch()* y *postDispatch()* se ejecutan antes y después de realizar el dispatch del action actual.

- **El preDispatch()** se suele usar para hacer controles sobre los permisos del usuario logueado.
- **El postDispatch()** para agregar contenido a la vista (por medio del response object), setear headers especiales y muchas otras acciones que se nos pueden ocurrir.

Accediendo al Request y al Response Object

El request *Zend_Controller_Request_Abstract* contiene información como por ejemplo el nombre del action / módulo / controlador actual, la url, parámetros recibidos por *GET* y *POST*, accedemos con *getRequest()* algunos métodos que nos pueden interesar del request object:

Código fuente

```
$this->getRequest()->getActionName(); // devuelve el nombre del action actual
$this->getRequest()->getControllerName(); // el nombre del controller
$this->getRequest()->getModuleName(); // el nombre del módulo
$this->getRequest()->getRequestUri(); // la url actual
$this->getRequest()->isPost(); // si el request fue hecho por el método POST
$this->getRequest()->isXmlHttpRequest() // o fue hecho por ajax?
```

El response *Zend_Controller_Response_Abstract* setea y devuelve los headers del sitio y el contenido HTML del mismo, accedemos a él con el método *getResponse()*:

Código fuente

```
$this->getRequest()->setHttpResponseCode(); // para setear por ejemplo un 404,
301, etc.
$this->getRequest()->getHttpResponseCode(); // devuelve el HTTP response code
actual
$this->getRequest()->appendBody($content); // para agregar contenido al body
$this->getRequest()->clearBody(); // para borrar todo el contenido generado
hasta el momento
```

_forward() y _redirect()

Los métodos conocidos como Utility Methods tienen cierto parecido con _forward ejecuta otro action. Si lo llamamos durante el *preDispatch()*, el action actual será reemplazado por el nuevo de lo contrario será ejecutado su sintaxis es:

```
_forward($action, $controller = null, $module = null, array $params = null)
```

_redirect() recibe una url y realiza un redirect HTTP terminando la ejecución del script actual:

```
_redirect($url, array $options = array())
```

Actions Helpers

Los Action Helpers permiten agregar funcionalidades extras al controlador en tiempo de ejecución, sin tener que modificar al controlador ni hacer una clase propia que extienda *Zend_Controller_Action*. Esto es algo muy bueno, la posibilidad de agregar funcionalidades y comportamientos sin modificar la clase base significa que tenemos menos cosas que testear y nuestro código es más pequeño y reutilizable.

Los view helpers funcionan sobre el magic method_call entonces al momento de llamar un método que no existe sabemos que es un view helper y *Zend_View* se encarga de buscarlo y de cargarlo. En cambio, el método *__call* de *Zend_Controller_Action* tiene otro uso, por lo que no se puede usar para cargar los action helpers. En su lugar, estos deben ser registrados manualmente y son almacenados en el atributo *$_helper* de *Zend_Controller_Action*.

Zend Framework trae por defecto unos cuantos action helpers:

- **ActionStack**: como su nombre lo indica, permite agregar requests al ActionStack Front Controller Plugin, permitiendo crear una pila de actions a ejecutar durante el request. Sin embargo su uso no es recomendado ya que tiene una importante penalización de rendimiento.
- **ContextSwitch y AjaxContext:** estos helpers permiten retornar diferentes tipos de formatos en el request, ya sea json, xml, html, etc. El AjaxContext es una implementación especifica del ContextSwitch que reconoce automáticamente request que fueron hechas por AJAX.
- **Autocomplete, AutoCompleteDojo y AutoCompleteScriptaculous:** la mayoría de los frameworks de JavaScript cuentan con un autocomplete, pero no todos esperan el mismo tipo de datos (algu-

nos trabajan con JSON, otros con texto plano, etc). El Autocomplete Action Helper se encarga de simplificarnos esta tarea, brindándonos una funcionalidad base que luego podremos extender para acomodarla a los requerimientos del framework que usemos. Zend Framework trae por defecto soporte para los autocompletes de dojo y scriptaculous.

- **FlashMessenger**: este es uno de los helpers más usados, permite enviarle al usuario un mensaje que será mostrado en el próximo request, es decir la próxima vez que cargue la página. Por ejemplo, si vamos a hacer un redirect a otra página de nuestro sitio con el flashMessenger podremos generar un mensaje que será mostrado una vez que esta otra página se cargue.
- **JSON**: al trabajar con AJAX es muy común enviar datos en JSON, este helper recibe como parámetros los datos a encodear y se encarga de otras tareas extras como deshabilitar el layout, setear el header content-type a 'text/json', etc.
- **URL**: crea URLs para usar, por ejemplo al hacer un redirect en base al módulo / controlador / action actual, la url base y otros parámetros recibidos.
- **Redirector**: este helper es llamado al hacer un _redirect(), pero tiene muchas más opciones como especificar el código de respuesta HTTP, forzar o no un exit() luego del redirect, etc.
- **ViewRenderer**: este es quizás el helper más usado ya que registra automáticamente la vista en el controlador, setea los paths relacionados a la vista, crea un objeto del tipo Zend_View global para toda la aplicación, etc.

Estos helpers que vienen por defecto pueden ser accesibles en cualquier momento de las siguientes maneras, por ejemplo para llamar al FlashMessenger:

```
$flashMessenger = $this->_helper->getHelper('FlashMessenger');
```

La forma más corta:

```
$flashMessenger = $this->_helper->flashMessenger
```

Pero el flashMessenger implementa el método direct, que al ser llamado trabaja como un alias de addMessage:

```
// Zend/Controller/Action/Helper/FlashMessenger.php
public function direct($message)
{
return $this->addMessage($message);
}
```

Entonces en vez de llamar directamente al *addMessage()* podemos hacer esto:

```
$this->_helper->flashMessenger("123");
```

$this->_helper es el Helper Broker, que por sobrecarga de métodos verifica si lo que se esta llamando es un helper existente, y de ser así ejecuta el método direct con los parámetros que recibe.

El Helper Broker

Zend_Controller_Action_HelperBroker es el encargado de manejar los action helpers: registrarlos, retornarlos, controlar si existe uno determinado, etc.

Para registrar un helper llamamos el método estático *addHelper()*:

```
Zend_Controller_Action_HelperBroker::addHelper(new Mdw_Action_Helper_Ejemplo);
```

Sin embargo hay otras dos opciones que son más eficientes ya que no requieren que el helper sea instanciado en el momento.

addPrefix(): en base a los prefijos que recibe como parámetro, al momento de cargar un helper resolverá el path en el que se encuentra, lo instanciará y lo devolverá:

```
Zend_Controller_Action_HelperBroker::addPrefix('My_Action_Helpers');
```

addPath(): recibe como parámetro un directorio y un prefijo en el que buscar los helpers, permitiendo buscar clases con un determinado prefijo en un directorio establecido:

```
Zend_Controller_Action_HelperBroker::addPath('/Plugins/Helpers', 'Helper');
```

Creando el Helper

Crear un action helper propio es sumamente fácil, nuestra clase debe extender de Zend_Controller_

Action_Helper_Abstract, luego tenemos que registrarla por alguno de los métodos anteriormente vistos.

Front Controller Plugins

El Front Controller atraviesa distintos estados y por cada uno de estos se activan uno o varios eventos (también llamados hooks):

- **routeStartup()**: antes de inicializar el routing del request
- **routeShutdown()**: al finalizar el routing del request
- **dispatchLoopStartup()**: antes de entrar al dispatch loop
- **preDispatch()**: antes de realizar el dispatch de un action
- **postDispatch()**: después de realizar el dispatch de un action
- **dispatchLoopShutdown()**: al terminar el dispatch loop

Un plugin es simplemente una clase que extiende de Zend_Controller_Plugin_Abstract, clase base que define todos estos hooks. Nuestro plugin sobreescribirá los métodos deseados para agregar la funcionalidad deseada. A excepción del dispatchLoopShutdown(), todos los demás métodos reciben al $request como primer parámetro.

Registrando Plugins

Los plugins necesitan ser registrados manualmente en el FrontController:

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Mdw_Plugin(), $index);
```

El segundo parámetro especifica un índice dentro del stack de plugins. Este parámetro es opcional y alterara el orden en que son ejecutados los plugins, si no se completa se van ejecutando en el orden en que fueron agregados. Según las tareas que estemos haciendo, puede llegar a ser importante que un plugin se ejecute antes que otro, así que no hay que perder esto de vista.

¿Qué tareas suelen hacer? Las tareas más comunes son controlar si el usuario esta logueado o tiene acceso a ciertas páginas, cachear, modificar el HTML generado, etc.

Capítulo 4

Vistas, View Helpers y Layout

Capítulo 4

Vistas, View Helpers y Layout

Como vimos en capítulos anteriores es un objeto *Zend_View* al que accedemos desde el controlador con *$this->view*, y desde la vista propiamente dicha con $this recordaremos que asignamos una variable en el controlador con *$this->view->bar = "bar"* y en la vista la accedemos con *$this->bar*.

Cuando empezamos a trabajar con Zend Framework surgen dudas sobre cómo no repetir cosas en distintas vistas, si hay algo que queremos usar en varios lugares cuál es la forma correcta de hacerlo. En nuestra etapa pre-framework era hacer un include y en este capítulo veremos que para lograr esto tenemos a nuestra disposición los View Helpers.

Creando un View Helper

Los view helpers son clases que extienden de *Zend_View_Helper_Abstract* cuyo nombre tiene el formato *Zend_View_Helper_NombreDelHelper*. Los helpers van dentro de la carpeta helpers del módulo correspondiente, y el nombre del archivo es *NombreDelHelper.php*. Invocarlo es tan simple como hacer *$this->nombreDelHelper()* el helper debe implementar un método público llamado *nombreDelHelper()*.

Veamos un ejemplo:

Código fuente

```
// helper que muestra un mensaje de error
// application/views/helpers/ErrorBox.php
class Zend_View_Helper_ErrorBox extends Zend_View_Helper_Abstract {
public function errorBox($msg) {
return "<div class='error'>$msg</div>";
}
}
// en la vista
echo $this->errorBox("Ocurrió un error");
```

Características de los helpers

En los view helpers no hay que hacer *echo*'s, lo correcto es retornar el contenido y hacer el *echo* en la vista.

Código fuente

```
public function errorBox($msg) {
/* esto esta mal */
echo "<div class='error'>$msg</div>";
/* esto esta bien */
return "<div class='error'>$msg</div>";
}
```

Hay una regla práctica que dice: si un view helper se llama sin parámetros, entonces se devuelve una instancia de si mismo. Supongamos el ejemplo anterior, pero ahora no queremos tener un view helper que muestre mensajes de error, otro que muestre mensajes de éxito, etc., sino que queremos un solo helper que se encargue de todo ¿cómo lo hacemos?

Una solución es pasarle al helper dos parámetros, uno el mensaje que queremos mostrar y otro el tipo de mensaje o sino crear un método para cada tipo de mensaje:

Código fuente

```
class Zend_View_Helper_Messages extends Zend_View_Helper_Abstract {
public function messages() {
return $this;
}
public function error($msg) {
return "<div class='error'>$msg</div>";
}
public function success($msg) {
return "<div class='success'>$msg</div>";
}
}
```

Entonces llamamos al helper:

```
echo $this->messages()->error("Ocurrio algun error");
echo $this->messages()->success("Todo salio correctamente!");
```

Desde el view helper podemos acceder a la vista con $this->view el caso más común es para ejecutar un view helper desde otro view helper, por ejemplo:

Código fuente

```
class Zend_View_Helper_Ejemplo extends Zend_View_Helper_Abstract {
public function ejemplo() {
// $html contendrá el html generado a lo largo del helper y que luego
sera retornado a la vista
$html = "";
....
$html .= $this->view->messages()->success("Todo parece ir bien");
....
return $html;
}
}
```

Pero, también hay otro uso que es para acceder a variables que fueron agregadas a la vista:

Código fuente

```
// en el controlador
$this->view->idUsuario = $idUsuario;
// en el helper
public function ejemplo() {
$idUsuario = $this->view->idUsuario;
}
```

Pero esta no es una buena práctica porque cuando deseemos testear nuestras aplicaciones por ejemplo con phpUnit se produce un acoplamiento que nos impide testear nuestra aplicación, así que lo correcto es hacer:

Código fuente

```
// en el controlador
$this->view->idUsuario = $idUsuario;

// en la vista
$this->ejemplo($this->idUsuario);
// en el helper
public function ejemplo($idUsuario) {
}
```

Como vemos queda más claro, limpio y más testeable, supongamos que en un momento el helper falla porque la variable $this->view->idUsuario no existe, ¿Cómo sabemos en que momento de la aplicación se esta generando? Así que para hacerlo bien, la recibimos como parámetro y listo.

View Helpers Iniciales

Zend trae por defecto muchos view helpers, vamos a repasar los más usados y ver para que sirven, aunque les recomiendo leer el manual para conocer todos los demás, probablemente algún día necesitarán hacer algo que ya este hecho:

HeadLink

El headlink helper genera tags a los que le podemos agregar hojas de estilos, favicons, etc. Por ejemplo, en nuestro layout tenemos:

Código fuente

```
$this->headLink()
->appendStylesheet('/css/global.css')
->appendStylesheet('/css/forms.css', 'screen', 'IE'); // también podemos
agregar
css condicionales, por ejemplo para IE
// al hacer el echo, el HeadLink genera el html para agregar los view helpers
echo $this->headLink();
```

En nuestras vistas podemos ir agregando los CSS adicionales que usemos:

Código fuente

```
$this->headLink()->appendStylesheet('/css/alta.css');</pre></div>
<h4>HeadScript</h4>
<p>Similar al headLink, pero para agregar :>/p>
[sourcecode lang="php"]$this->headScript()->appendFile('/js/scripts.js');
echo $this->headScript();
```

HTML Object Helpers:

El helper principal es *htmlObject()* que nos sirve para agregar objetos de diferentes tipos a nuestra página. Luego tenemos otros 3 helpers que son implementaciones particulares del *htmlObject()*:

- **htmlFlash()** para agregar archivos flash
- **htmlPage()** para embeber páginas XHTML
- **htmlQuicktime()** para agregar un vídeo de quicktime

Así que si queremos agregar un archivo flash a nuestra página hacemos directamente:

```
echo $this->htmlFlash('/flash/flash.swf');
```

Action View Helper:

El action view helper permite ejecutar un action de un controlador dado y retornar el contenido a la vista actual. En otras palabra, podemos ejecutar desde la vista un action de cualquier módulo. Por ejemplo, tenemos un action que carga un menú, entonces podemos hacer:

```
echo $this->action('menu', 'miControlador', 'miAction');
```

A primera impresión es una buena idea ya que podemos reutilizar código "widgetizar" contenido, etc. Pero la realidad es otra, desde el lado de rendimiento Zend tiene que clonar la vista, realizar un dispatch adicional y copiar el request object.

Si el action que llamamos esta en el mismo controlador y tenemos código en algún hook (predispatch, postDispatch, etc) lo estaríamos ejecutando dos veces. Por otro lado y no menos importante, estamos

pasando por alto el modelo MVC (la vista comunicándose con el controlador) y agregando una complejidad innecesaria a nuestro diseño. Pero si todo esto no termina de convencer para dejar de lado dicho helper a partir de la versión 2.0 de Zend Framework, va a dejar de existir.

¿Cuál es la solución? Muy fácil, creamos un view helper que genere el menú:

```
echo $this->menuLateral();
```

Así que si estas tentado a usar el action view helper ten la seguridad que estas teniendo un problema de diseño en tu aplicación y que con total seguridad existe una solución alternativa que sea más simple y correcta.

El Layout

Zend_Layout implementa el patrón Two Step View que permite incluir nuestras vistas dentro de otra vista general que funcione como template del sitio. En el primer artículo vimos como inicializar el layout, ahora que ya sabemos lo que es un view helper y los distintos view helpers que trae zend veamos como quedaría nuestro layout:

Código fuente

```
// application/layouts/layout.phtml;
<?php echo $this->doctype();?>
<html>
<head>
<?php
echo $this->headMeta();
echo $this->headTitle();
echo $this->headScript();
echo $this->headStyle();
?>
</head>
<body>
<?php
// view helper propio para crear el header del sitio
echo $this->siteHeader();
```

```
?>
<div id="content">
<?php
// mostrando el contenido de la vista
echo $this->layout()->content;
?>
</div>
<?php
  // view helper propio para crear el footer
echo $this->siteFooter();
?>
</body>
</html>
```

Desde el layout también podemos ejecutar view helpers como si estuviéramos en una vista.

Capítulo 5

Crea y maneja formularios con Zend_Form

Capítulo 5

Crea y maneja formularios con Zend_Form

Zend_Form[1] es el componente de Zend Framework encargado de crear y manejar los formularios que usemos en nuestras aplicaciones su uso será obligatorio en cualquier desarrollo web.

Entre su principales responsabilidades se encuentran:

- Validar los formularios y mostrar los errores
- Filtrado (Filter) de datos (escape / normalización de los datos recibidos)
- Generar el HTML Markup del form y sus elementos

Todo esto convierte a *Zend_Form* en una herramienta realmente simple y potente que nos permitirá ahorrar tiempo al trabajar con formularios y generar un código mucho más pro.

Crear un formulario

Crea un formulario de la siguiente forma:

```
$form = new Zend_Form;
```

Aunque en la práctica lo más limpio es crear una clase propia que extienda de Zend_Form:

Código fuente

```
// application/forms/MyForm.php
class MyForm extends Zend_Form {
/* método usado para inicializar el form */
public function init() {
}
}
// en el controlador
$form = new MyForm;
```

1 http://framework.zend.com/manual/en/zend.form.html

Setear atributos

Lo más básico es setear el nombre, action, class y demás:

Código fuente

```
public function init() {
$this->setName('frmPrueba')
->;setAction('/process_form.php')
->setEnctype('multipart/form-data')
->setMethod('post')
->setAttrib('class', 'frmClass'); // cualquier atributo
html se puede setear con setAttrib()
}
```

Agregando elementos

Zend trae por default los siguientes tipos de elementos:

- Button, Checkbox
- MultiCheckbox, Hidden, Image, Password, Radio, Reset, Select
- Multi-select, Submit, Text, Textarea

Hay dos formas de agregar elementos.

La primera:

```
// instanciar el elemento y pasárselo al form
$form->addElement(new Zend_Form_Element_Text('username'));
```

La segunda:

```
// pasar el tipo del elemento al form y que este lo instancie
$form->addElement('text', 'username');
```

En este artículo usaremos la segunda forma cuya sintaxis es:

```
$form->addElement($elementType, $elementName, array $config);
```

Revisemos el siguiente ejemplo:

Código fuente

```
// application/form/Registro.php
class Form_Registro extends Zend_Form {
public function init() {
$this->addElement(
'text',
'nombre',
array('required' => true, 'label' => 'Nombre')
);
$this->addElement(
'radio',
'sexo',
array(
'required' => true,
'label' => 'Sexo',
'multiOptions' => array('h' => 'Hombre', 'm' => 'Mujer')
)
);
$this->addElement(
'select',
'como_conociste',
array(
'required' => true,
'label' => 'Como nos conociste?',
'multiOptions' => array('1' =&gt; 'Por buscadores', '2'
=> 'Recomendacion')
)
);
$this->addElement('submit', 'enviar', array('label' => 'Enviar'));
}
}
```

Validando el formulario

Al marcar un campo como *required* lo que hacemos es marcarlo como obligatorio. Controlamos que el formulario haya sido llenado correctamente llamando al método *isValid()* se controla que todos lo campos obligatorios estén completos.

Además de otras reglas de validación más avanzadas que se pueden crear fácilmente agregando validators a los distintos elementos (validar email, fecha, solo letras, solo números, etc.) Si el formulario esta incompleto y automáticamente se agregan lo mensajes de error al markup del form.

Código fuente

```
// en el controlador
if ($this->getRequest()->isPost()) {
if ($form->isValid($this->getRequest()->getPost()) {
/*
El formulario esta bien (todos los campos obligatorios fueron completados y los
validators no arrojaron ningún error), si isValid() devuelve false, se cargan
los mensajes de error a los elementos que no estén completados correctamente:
*/
$values = $form->getValues();
$nombre = $form->nombre-&gt;getValue();
$sexo = $form->getElement('sexo')->getValue();
}
}
```

Como vemos en el ejemplo, podemos obtener los valores del formulario con *getValues()*, y obtener el valor de un elemento en particular con getValue() accedemos al elemento por sobrecarga (*magic method __get*) o con *getElement()*.

Filters y Validators

Los filters se encargan de escapar / normalizar los datos recibidos, por ejemplo eliminar todos los números de una cadena de texto, convertir los caracteres extraños a su entidad HTML correspondiente y se pueden agregar todos los filtros que se deseen.

Hay varias formas de agregar un filter a un elemento:

Código fuente

```
// instanciar el filtro y pasarselo al elemento
$element->addFilter(new Zend_Filter_Alnum());
// pasarle el nombre completo
$element->addFilter('Zend_Filter_Alnum');
// pasarle el nombre corto
$element->addFilter('Alnum');
// al momento de crear el elemento
$form->addElement('text', 'nombre', array('filters' =>
array('Alnum')));
```

Zend no crea inmediatamente la instancia del filtro sólo cuando lo utiliza (lazy loading) y ahorramos memoria. Los filtros se aplicarán al momento de hacer el *isValid()* y Zend no modificará la fuente de datos (en este caso *$_POST*), sino que obtendremos los datos filtrados al pedir los valores con *getValues()* / *getValue()*.

El tema de los validators es prácticamente igual:

```
$element->addValidator('Alnum');
$element->addValidator('StringLength', false, array(6, 20));
// al momento de crear el elemento
$form->addElement(
'text',
'nombre',
array(
'validators' => array(
'Alnum',
```

```
array('StringLength', false, array(6, 20))
)
)
);
```

Podemos agregar diversos validators al momento de validar el form se controla que cada validator este bien, si alguno tiene problemas entonces falla la validación completa y se muestran los mensajes de error.

addValidator() acepta tres parámetros:

1. El primero es el nombre del validator.
2. El segundo es un bool que determina que, si falla este validator, no siga con los demás (true) o que siga (false).
3. Array con los valores que recibe el constructor. En nuestro ejemplo, StringLength recibe la cantidad de letras que se aceptan (entre 6 y 20).

Modificando el HTML Markup

Este es un primer acercamiento a Zend_Form y te dejo el archivo que contiene el código completo para que lo descargues y pruebes[1].

1 http://www.maestrosdelweb.com/util/zend-form.rar

Capítulo 6

Sobre decorators en Zend_Form

Capítulo 6

Sobre decorators en Zend_Form

Zend_Form es una solución altamente flexible al problema de los formularios, como vimos en el capítulo anterior se encarga de generarlos, validar, filtrar datos y otra de sus tareas es generar el markup del form y los distintos elementos este suele ser el aspecto que más complicaciones nos genera.

Zend_Form_Decorator_Abstract

Esta es la clase base a partir de la cual extienden todos los *decorators* anteriormente nombrados. Los métodos más importantes que implementa son:

```
setOptions() //setea las diferenes opciones del decorator.
setOptions("placement", APPEND / PREPEND) //si el contenido generado por el
decorator se agrega al principio o al final.
setOptions("separator", string) //separador entre el contenido viejo y el nuevo
```

Standard Decorators

Zend trae por default algunos decorators, veamos los métodos más usados de cada uno, todos los decorator tienen los métodos nombrados anteriormente para *Zend_Form_Decorator_Abstract*:

Callback: Permite generar contenido de acuerdo a un callback en particular.

```
setCallback($callback)
```

Description: Muestra la descripción seteada en un Zend_Form o Zend_Form_Element.

- setTag($tag): El tag del elemento que contiene el texto, por defecto <p>
- setEscape($bool): Si queremos escapar o no los caracteres HTML por defecto es true.
- setClass($class): Class del elemento, por defecto hint.

Errors: Lista los errores que contiene el elemento.

Fieldset: Pone dentro de un fieldset el contenido recibido.

```
setLegend($legend): setea el legend del fieldset.
```

Form: Genera el tag *<form>* del formulario.

FormElements: Forms, display groups y subforms son colecciones de elementos. Este decorator recorre cada uno de estos elementos y genera su contenido, por lo que es un decorator que siempre debe estar presente.

FormErrors: Parecido al Errors, pero muestra todos los mensajes de error en la parte superior del formulario en lugar de hacerlo para cada elemento en particular.

HtmlTag: Permite generar los tag HTML.

```
setTag($tag)
setOption("openOnly", $bool): crea solamente el tag de apertura.
setOption("closeOnly", $bool): crea solamente el tag de cierre
```

Label: En general todos los elementos del form tienen un texto asociado que es generado por este decorator.

ViewHelper: Es el elemento propiamente dicho, por ejemplo un *<input/>*, un *<textarea/>*, etc.

Setear Decorators

Podemos agregar decorators a forms (subforms) y elementos. Cada uno de estos objetos puede tener asociados varios los decorators. El orden de los decorators es importante porque generalmente determina el markup final, no es lo mismo tener *<input /> <label />* que *<label /> <input />*, de un ejemplo a otro lo que cambia es el orden en el que fueron incluidos los decorators *ViewHelper* y *Label*.

Los decorators asociados a un objeto se comportan como una cola, por lo que si queremos agregar un decorator al principio (o en cualquier otro lugar que no sea el final) tendremos que setear todos los decorators otra vez.

Para agregar / setear decorators tenemos dos métodos:

1 | **setDecorators()** setea los decorators del elemento, es decir que si ya hay algún decorator seteado

lo sobreescribe.

2 | **addDecorator()** agrega un decorator al final de los decorators existentes.

Estos métodos son tanto para elementos, como para forms, subforms y display groups.

```
$element->setDecorators($decorators);
$form->addDecorator($decorator);
```

Veamos algunos ejemplos de uso para resolver dudas, suponiendo que nuestro elemento es un text input:

```
$element->setDecorators(array("ViewHelper", "Label", "HtmlTag"));
```

Logramos generar * <label /> <input /> *

```
$element->setDecorators(array("ViewHelper", "Label", "HtmlTag", "HtmlTag"));
```

Puedes pensar que tenemos dos divs, pero no es el caso, porque el último HTMLTag sobreescribe al primero. Para tener dos HtmlTag distintos hay que asignarle un alias a alguno de ellos. Antes de esto, veamos sobre las diferentes formas de pasar un decorator.

Es importante entender este ejemplo porque aquí vemos todas las posibilidades existentes. Obviamente no es del todo válido porque tenemos 3 decorators que se llaman igual "HtmlTag" y otros 2 que se llaman "miHtmlTag", por lo que se estarían sobreescribiendo y solo quedaría uno de cada tipo, pero evaluemos como un ejemplo educativo:

Código fuente

```
$element->setDecorators(array(
// agregamos un decorator del tipo HtmlTag
"HtmlTag",
// idéntico al primer ejemplo, el hecho de que lo pasemos dentro de un array
no cambia nada
array("HtmlTag"),
/* aca vemos un ejemplo mas concreto del caso anterior. Pasamos el decorator
como un array cuando queremos setear alguna opción adicional.
el primer elemento del array es el nombre del decorator, y el segundo un
array de opciones a setearle al decorator */
array("HtmlTag", array("tag" => "span")),
/* este ejemplo es igual al anterior, pero ahora al decorator le damos un
alias. En el futuro no nos referiremos a él como el decorator HtmlTag, sino
como
"miHtmlTag" */
   array(array("miHtmTag" => "HtmlTag"), array("tag" => "span")),
// por último, aqui creamos un HtmlTag dándole otra vez un alias, pero esta
vez sin pasarle las opciones
array(array("miHtmlTag" => "HtmlTag"))
));
```

Teoría de funcionamiento

Ahora que ya sabemos como agregar decorators, darles un alias para usar varios decorators del mismo tipo y setear las opciones veamos como van trabajando los distintos decorators. El contenido generado por un decorator es pasado al siguiente, este agrega (al principio o al final) su propio markup al contenido que recibe y se lo pasa al siguiente decorator que vuelve a hacer lo mismo.

Supongamos que nuestro elemento tiene los siguientes decorators:

```
$element->setDecorators(array("ViewHelper", "Label", "HtmlTag", "Error"));
```

1 | Al ejecutar el método render() se inicia el proceso, al primer decorator se le pasa una cadena vacía (“”)

```
$html = “”;
```

2 | El decorator ViewHelper genera el HTML del elemento asociado, si nuestro elemento es un text input obtendremos como resultado:

```
$html = $html . ‘<input type=”text” name=”…” />’;
```

3 | El siguiente decorator es el Label, por defecto el contenido que genera lo agrega al principio (placement = prepend):

```
$html = ‘<label>...</label>’ . $html = ‘<label>...</label><input type=”text” name=”…” />’;
```

4 | El HtmlTag por defecto funciona como wraper es decir que envuelve el contenido que recibe y el tag por defecto es div:

```
$html = ‘<div>’ . $html . ‘</div>’ = ‘<div><label>...</label><input type=”text” name=”…” /></div>’;
```

5 | El decorator Errors se aplica solamente si hay algún error, supongamos que ocurrió algún error al validar el form, el markup será así:

```
$html = $html . ‘<ul><li>Error 1</li><li>Error N</li></ul>’;
```

Termina el proceso y el markup final es:

```
<div>
<label>...>/label> <input type=”text” name=”…” />
</div>
<ul>
<li>Error 1>/li>
<li>Error N>/li>
</ul></div>
```

Como vimos el markup se fue armando paso a paso y los decorators trabajaron desde adentro hacia afuera, es decir que los decorators que están más afuera son los últimos en ser agregados, ya que los decorators agregan contenido adicional pero generalmente no modifican el contenido que reciben.

Un poco de teoría sobre placement

¿Estos dos decorators son iguales?

```
array('ViewHelper', 'Label') ¿=? array('Label', 'ViewHelper')
```

Quizás la respuesta rápida es decir que no, pero en realidad sí son similares porque el comportamiento por default del Label es agregar el contenido al principio, entonces en el primer caso tenemos:

```
1) <input ... />
2) <label ... /><input ... />
```

En el segundo caso:

```
1) <label ... />
2) <label ... /><input ... />
```

Muchas veces cuando pensamos que algo no se puede hacer o es muy difícil se soluciona manejando adecuadamente el placement de cada decorator.

Algunos ejemplos

Veamos un ejemplo de un markup un poco más complejo:

Código fuente

```
public function init() {
$this->addElement('text', 'uno, array('label' => 'Uno'));
$this->addElement('text', 'dos, array('label' => 'Dos'));
}
```

¿Cómo generamos el siguiente markup?

```
<div class="row">
<div><label>Uno</label><input type='text' name='uno' /></div>
<div><label>Dos</label><input type='text' name='dos' /></div>
</div>
```

Veamos:

Código fuente

```
public function init() {
$this->addElement('text', 'uno', array('label' => 'Uno'));
$this->addElement('text', 'dos', array('label' => 'Dos'));
$this->addElement('text', 'tres', array('label' => 'Tres'));
$this->setElementDecorators(array('Label', 'ViewHelper', 'HtmlTag'));
// nos falta el div que los envuelve...
$this->getElement('uno')->addDecorator('HtmlTag', array('class' => 'row',
'openOnly' => true));
$this->getElement('tres')->addDecorator('HtmlTag', array('closeOnly' =>
false));
}
```

Con *setDecorators()* setemos los decorators de todos los elementos del form agregados hasta el momento. Como los tres elementos tienen los mismos decorators en lugar de setearlos de forma individual se pueden realizar los cambios en un sólo lugar.

Como cada decorator funciona sobre un elemento, no podemos envolver los tres elementos aplicando un decorator a uno solo de ellos. Por lo tanto, al primer decorator le agregamos el tag de apertura y al último el tag de cierre, al terminar el *render()*, el resultado final será el esperado.

Por último, otro ejemplo bastante común es como maquetar una tabla:

Código fuente

```
<form>
<table>
<tr><td><label>Uno</label></td><td><input name='uno' /></td></tr>
<tr><td><label>Dos</label></td><td><input name='dos' /></td></tr>
<tr><td><label>Tres</label></td><td><input name='tres' /></td></tr>
</table>
</form>
```

Los decorators de los elementos los seteamos así:

```
$this->setElementDecorators(
array(
"ViewHelper",
array("HtmlTag", array("tag" => td")),
array("Label" , array("tag" => "td")),
array(array("tr" => "HtmlTag"), array("tag" => "tr"))
)
);
```

- Creamos el input y lo envolvemos en un >td>
- Creamos el label con otro tag td y se ubica al principio (ya tenemos los dos >td> en orden)
- Por último un HtmlTag >tr> que envuelva todo lo anterior

Ahora faltaría el tag table haremos como en el ejemplo anterior usar dos HtmlTag en el primer y último elemento. Pero veamos otra posibilidad, modificando los decorators del form de la siguiente manera:

```
$this->setDecorators(
array(
"FormElements",
array("HtmlTag", array("tag" => "table")),
"Form"
)
);
```

El decorator `FormElements` renderiza todos los elementos, entonces le agregamos el tag table que falta y por último el tag form. Para maquetar cualquier formulario necesitamos conocer a fondo los decorators y todas las opciones de configuración de cada uno.

Capítulo 7

Construir aplicaciones multi-idioma con Zend_Translate

Capítulo 7

Construir aplicaciones multi-idioma con Zend_Translate

Zend_Translate es el componente de Zend encargado de facilitarnos la creación de aplicaciones multilenguaje y que permite leer la información de diversas fuentes de datos (Array PHP, CSV, Gettext) su simplicidad de uso la facilidad para cambiar de un adapter por otro detección automática del lenguaje del usuario.

Adapters disponibles

Cada fuentes de infromación cuenta con un adapter propio que se encarga de procesar el archivo y devolverle a *Zend_Transate* los datos en el formato correcto. En este artículo trabajaremos con archivos de configuración en formato PHP ya que son los más simples y los que usaremos en la mayoría de nuestros proyectos.

Creando los archivos

Las traducciones deben colocarse dentro de *application/configs/languages* donde creamos un subdirectorio nuevo por cada lenguaje o crear directamente los archivos con las traducciones, por ejemplo en.php, es.php, etc.

Usando el adapter *Zend_Translate_Adapter_Array* los archivos de configuración tendrán la siguiente estructura:

```
/* configs/languages/en.php */
return array("uno" => "one", "dos" => "two", "tres" => "three");
/* configs/languages/es.php */
return array("uno" => "uno", "dos" => "dos", "tres" => "tres");
```

Si alguna vez tuvimos que crear algún script propio para que un sitio nuestro sea multi-idioma usamos algún archivo similar.

Obteniendo las traducciones

El siguiente paso es crear una nueva instancia de *Zend_Translate* pasando como parámetro el adapter que usamos, la ubicación del archivo y el idioma correspondiente:

```
$translate = new Zend_Translate('array', APPLICATION_PATH . '/configs/languages/en.php', 'en');
```

Si queremos cargar otro idioma usamos el método addTranslation:

```
$translate->addTranslation(APPLICATION_PATH . '/configs/languages/de.php', 'de', $options);
```

Entre las *$options* más interesantes tenemos:

- **clear (bool)**: Al agregar traducciones para un idioma ya existente determina si se borran todos los datos existentes o no.
- **disableNotices (bool)**: Por defecto cuando se pide una traducción que no existe se genera un error del tipo E_USER_NOTICE en ambientes de producción hay que poner este flag a false.
- **reload (bool)**: Indica que se regeneren los archivos de cache.

Para obtener la traducción tenemos el método *translate($messageId, $locale = null)*, si tenemos varios idiomas cargados podemos especificar que traducción queremos obtener en el segundo parámetro. Este método tiene un alias muy usado que es el método *_()*, por lo que para obtener la traducción hacemos:

```
$translate->_("uno"); // obtenemos la traducción en el idioma actual
```

Otros métodos que nos serán útiles para conocer son:

- **getLocale() / setLocal($locale)**: Para obtener / setear el idioma actual.
- **isAvailable($locale)**: Para saber si un lenguaje esta disponible.
- **isTranslated($messageId, $original, $locale)**: Para saber si una palabra esta traducida.

Cacheando archivos

Supongamos que usamos un formato como por ejemplo XML, si tenemos cientos o miles de palabras

para procesar el XML puede llegar a ser bastante pesado y su procesamiento costoso en términos de tiempo, por lo que podemos setearle a *Zend_Translate* una instancia de *Zend_Cache* para que procese el XML sólo una vez y las veces siguientes lea los datos directamente desde la cache.

Creamos una instancia de *Zend_Cache* con el frontend y el backend luego setearla con el método estático *setCache*:

```
$cache = new Zend_Cache("Core", "File", $frontendOptions, $backendOptions);
Zend_Translate::setCache($cache);
```

Métodos relativos a la cache:

- getCache()
- setCache(Zend_Cache_Core $cache)
- hasCache()
- removeCache()
- clearCache()

Integración con otros componentes

Zend_Translate tiene una gran integración con otros componentes del framework como *Zend_View*, *Zend_Form* podemos guardar nuestra instancia de *Zend_Translate* en *Zend_Registry* y esta será detectada por estos otros componentes para hacer uso de él.

```
Zend_Registry::set('Zend_Translate', $translate);
```

Con esto ya estamos listo para trabajar con otros componentes, veamos los casos más importantes.

Integración con Zend_View: Zend_View_Helper_Translate

Una vez registrada la instancia en Zend_Registry desde las distintas vistas podemos obtener una traducción haciendo:

```
$this->translate("dos");
```

Integración con Zend_Form

Los formularios son una parte importante en todo sitio web, en cuanto a traducción nos interesa especialmente traducir labels, descriptions, legends, mensajes de error, etc. Ya tenemos registrada la instancia de Zend_Translate en el registry y creemos el formulario:

Código fuente

```
class Mdw_Form_Translate extends Zend_Form {
public function init() {
$this->addElement("text", "nombre", array("label" => "label_nombre"));
$this->addElement("submit", "enviar", array("label" => "label_enviar"));
$this->setDescription("description_form");
}
}
```

En nuestro archivo de configuración tendremos lo siguiente:

Código fuente

```
/* es.php */
return array(..., "label_nombre" => "Nombre", "label_enviar" => "Enviar",
"descripton_form" => "Descripcion!");
/* en.php */
return array(..., "label_nombre" => "Name", "label_enviar" => "Send",
"description_form" => "Description");
```

Al momento de renderizar el form automáticamente se buscará una traducción existente para labels, descriptions y si existe se mostrará el texto correcto.

Traducir mensajes de error

Para traducir los mensajes de error de los validators el tema es un poco más complejo:

- Tomando como ejemplo el validator NotEmpty (*Zend_Validate_NotEmpty*)
- Abrimos el archivo y buscamos el atributo *$_messageTemplates*

```
protected $_messageTemplates = array(
self::IS_EMPTY => "Value is required and can't be empty",
self::INVALID => "Invalid type given, value should be float, string, array,
boolean or integer",
);
```

Se hace referencia a constantes y las buscamos:

```
const INVALID = 'notEmptyInvalid';
const IS_EMPTY = 'isEmpty';
```

Con ellas armamos nuestro array con las traducciones:

```
/* es.php */
return array(..., "isEmpty" => "Este valor es obligatorio y no puede estar
vacío", "notEmptyInvalid" => "Tipo de dato incorrecto, debe ser float, string,
array, boolean o integer");
```

Lo mismo con todos los demás validators que vayamos a usar Zend Framework 1.10 viene con dichos mensajes ya traducidos a varios idiomas pero el español no es uno de ellos, por ahora tendremos que hacerlo por nuestra cuenta.

Capítulo 8

Integración con Ajax

Capítulo 8

Integración con Ajax

Trabajar con AJAX[1] es algo cotidiano, por eso como no podía ser de otra manera Zend nos facilita mucho las cosas y hace que del lado del servidor el trabajo sea el mínimo indispensable. Cuando trabajamos con AJAX queremos identificar las peticiones que fueron hechas por `XMLHttpRequest` y retornar los datos en el formato que nos sea más conveniente (HTML, JSON, XML, etc).

Para esto, Zend cuenta con dos action helpers el primero es el `ContextSwitch`, cuyo objetivo es facilitarnos el retornar los datos en diferentes formatos. El otro es el `AjaxContext`, que es una implementación especial del ContextSwich, que como su nombre indica, nos ayuda a identificar los `XmlHttpRequests` y actuar en consecuencia.

Cuando se activa un cambio de contexto, el helper realiza las siguiente acciones:

- Deshabilita el layout.
- Setea un sufijo alternativo para las vistas, de esta manera, cada contexto ejecutará una vista diferente.
- Setea el header content-type según el formato solicitado.

Activar el ContextSwich

Supongamos que tenemos un action llamado list el cual queremos que también retorne contenido en XML, y otro llamado comments y queremos que pueda retornar en XMLy Json. La opción “fea” es crear un action diferente para cada tipo de dato, pero como en realidad todos los actions devuelven la misma información sólo que en diferente formato, lo que hacemos es cambiar solo la vista.

La opción mala es seleccionar manualmente que vista renderizar, por lo cual tendríamos en todos nuestros actions unos horribles if’s o switch’s que rendericen la vista adecuada, y la opción correcta es tener un action helper que se encargue de identificar cuando hay que renderizar otra vista y haga el cambio. Esto es lo que hace el `ContextSwitch` en lugar de crear un nuevo action simplemente le agregamos los contextos adecuados a los ya existentes:

1 http://www.maestrosdelweb.com/editorial/ajaxpaso/

Código fuente

```
class IndexController extends Zend_Controller_Action {
public function init()
{
$contextSwitch = $this->_helper->getHelper('contextSwitch');
$contextSwitch->addActionContext('list', 'xml')
->addActionContext('comments', array('xml', 'json'))
->initContext();
}
}
```

El ejemplo es bastante claro y conciso por lo que solamente hay un par de cosas por explicar. Como vimos en el capítulo sobre action helpers *$this->_helper* es el HelperBroker, al que le pedimos que nos de el helper ContextSwitch y luego a este le agregamos los distintos contextos a los que va a responder cada action.

El *ContextSwitch* lo inicializamos con el método *initContext()* de suele poner en el init para inicializarlo en forma general para todos los actions del controlador. Lo único que faltaría es activar el contexto que deseamos. Para esto tenemos que pasar por *GET* la variable "format" con el formato elegido, por ejemplo XML.

Ahora cuando ejecutemos */index/list*, obtendremos la vista normal, es decir list.phtml, pero cuando entremos a */index/list?format=xml* se ejecutará la vista *list.xml.phml* que creará el XML correspondiente.

Contextos por default

Por defecto Zend incluye los contextos XML y JSON que funcionan de manera similar, con la diferencia que en el contexto JSON la vista (comments.json.phtml) no es obligatoria, ya que automáticamente se encodearán todas las variables de la vista y se responderá el contenido generado.

Especificando contextos existentes

En el ejemplo anterior vimos que para relacionar un contexto a un action usamos el método *addActionContext*, sin embargo hay otra forma de hacerlo especificando los contextos en el atributo *$contexts* del controlador.

Ejemplo similar al anterior:

Código fuente

```
class IndexController extends Zend_Controller_Action {
public $contexts = array(
'list' => array('xml'),
'comments' => array('xml', 'json')
);
public function init()
{
$this->_helper->getHelper('contextSwitch')->initContext();
}
}
```

AjaxContext

Este helper extiende al *ContextSwitch*, por lo que su funcionamiento es prácticamente igual, salvo algunas pequeñas diferencias. Por un lado, los contextos no se especifican en el atributo *$contexts* sino en *$ajaxable*.

Este contexto sólo se activará si el request es un *XmlHttpRequest* aunque pasemos la variable format por *GET*, si el request no fue hecho por AJAX no funcionará. Por último, se agrega el formato HTML, cuyas vistas tendrán el sufijo "ajax.phtml".

Ejemplo: veremos que el uso es prácticamente igual al del *ContextSwitch*

Código fuente

```
class IndexController extends Zend_Controller_Action {
public $ajaxable = array(
'list' => array('html'),
'comments' => array('html')
);
public function init()
{
$this->_helper->getHelper('ajaxContext')->initContext();
}
public function listAction() {
}
public function commentsAction() {
}
}
```

Como vemos es muy simple y el código queda muy limpio el Zend Framework se encarga del trabajo sucio (deshabilitar el layout, setear una vista diferente), nosotros solamente tenemos que inicializar el helper y poco más.

Capítulo 9

Introducción a Zend_Session y Zend_Auth

Capítulo 9

Introducción a Zend_Session y Zend_Auth

Con *Zend_Session* ya no tendremos que trabajar directamente con *$_SESSION* lo haremos por medio de *Zend_Session_Namespace*, que nos provee una interfaz orientada a objetos para manipular las variables de session. Esto no significa que *$_SESSION* ya no exista, sino que al estar trabajando en un entorno 100% orientado a objetos aprovechamos las diferentes ventajas que *Zend_Session* ofrece, tanto en funcionalidades como en calidad de código.

Cada *namespace* es simplemente un elemento del array *$_SESSION*.

Ejemplo: crear y manipular namespaces

Código fuente

```
$namespace = new Zend_Session_Namespace("ejemplo");
// $namespace hace referencia a $_SESSION["ejemplo"];
// seteamos algunas variables
$namespace->string = "valor";
$namespace->int = 7;
$namespace->array = array();
/*
el namespace ahora es asi
$_SESSION["ejemplo"] = array(
"string" => "unString",
"int" => 7,
"array" => array()
);
*/
// verificamos si una variable fue previamente seteada
isset($namespace->string); // true
isset($namespace->inexistente); // false
// eliminamos una variable
unset($namespace->int);
```

Si en otra página queremos acceder a los datos del *namespace* simplemente creamos una nueva instancia y ya podremos trabajar directamente sobre él.

```
$elMismoNamespace = new Zend_Session_Namespace("ejemplo");
// tengo acceso a todas las variables creadas anteriormente
echo $elMismoNamespace->string; // "unString"
```

Una de las principales ventajas de trabajar con namespaces en *$_SESSION* es que evitamos el problema de la colisión de nombres. Teniendo cada cosa en su namespace correspondiente, nunca pisaremos una variable por error.

- Los nombres de namespace no pueden ser una cadena vacía ni empezar con un número o un guión bajo.
- Tampoco pueden empezar con la palabra Zend, ya que dicho prefijo esta reservado para componentes de Zend.

Inicializando Session

Para inicializar session agregamos en nuestro Bootstrap lo siguiente:

```
protected function __initSession() {
Zend_Session::start();
}
```

Sin embargo no es totalmente obligatorio, ya que si al crear un namespace no hicimos el start(), Zend lo hará automáticamente. El problema es que si ya enviamos los headers ocurrirá el famoso error de "Headers already sent" por las dudas puedes hacer el start() manualmente.

Bloqueando un namespace

Zend_Session no sólo nos brinda una linda forma de acceder a las variables de sesión sino que además nos brinda algunas funcionalidades interesantes, como por ejemplo bloquear un namespace para que sea sólo de lectura y que no pueda ser alterado en otro lugar de nuestra aplicación.

Código fuente

```
$namespace = new Zend_Session_Namespace("ejemplo");
// bloqueamos la escritura
$namespace->lock();
// esto lanzaría una excepción, ya que el namespace esta bloqueado
$namespace->bar = "baz";
// antes de escribir preguntamos si podemos hacerlo
if (!$namespace->isLocked()) {
$namespace->bar = "baz";
}
// lo volvemos a habilitar
$namespace->unLock();
```

Podemos consultar si un *namespace* existe con:

- **Consulta**: Zend_Session::namespaceIsset($namespace)
- **Eliminar**: Zend_Session::namespaceUnset($namespace)

Tiempo de vida de un namespace

Otras de las funcionalidades es la de setear la duración de un namespace.

Hay dos posibilidades:

1. La primera es setear un tiempo en segundos ("que el namespace expire en 60 segundos").
2. La segunda es setear una cantidad de hops (cada vez que se carga la página se considera un hop).

Código fuente

```
$namespace = new Zend_Session_Namespace("ejemplo");
$namespace->setExpirationSeconds(30, "string"); // $namespace->string expira en
30 segundos
$namespace->setExpirationSeconds(30); // todo el namespace expira en 30
segundos
$namespace->setExpirationHops(4, "string"); // $namespace->string expira en 4
hops
$namespace->setExpirationHops(4); // todo el namespace expira en 4 hops
```

Si seteamos a la vez un expirationSeconds y un expirationHops el namespace expirará según el primer evento que se cumpla.

Introducción a Zend_Auth

Zend_Auth nos facilita la autenticación de usuarios, es decir el login / logout de usuarios, consultar por la identidad de la persona actualmente logueada, etc. Implementa el patrón singleton para obtener una instancia de la clase usaremos el método estático *getInstance()*:

```
$auth = Zend_Auth::getInstance();
```

Database Adapter

Si bien hay varios adapters disponibles (y podemos crear nuestros propios adapters de ser necesario), en la mayoría de nuestros proyectos trabajaremos contra una base de datos.

- **setTableName($table)**: El nombre de tabla contra la que se verificara que los datos del usuario sean correctos.
- **setIdentityColumn($column)**: El campo de la base de datos correspondiente al nombre de usuario.
- **setCredentialColumn($column)**: El campo de la base de datos correspondiente al password.
- **setIdentity($user)**: El nombre de usuario que queremos autenticar.
- **setCredential($pass)**: El password de dicho usuario.

Supongamos que nuestra tabla se llama 'mdw_usuarios', el campo usuario es 'user', el del password es 'pass' y recibimos los datos del login por $_POST, el ejemplo quedaría así:

Código fuente

```
$dbAdapter = Zend_Db_Table_Abstract::getDefaultAdapter();
$authAdapter = new Zend_Auth_Adapter_DbTable($dbAdapter);
$authAdapter
->setTableName('mdw_usuarios')
->setIdentityColumn('user')
->setCredentialColumn('pass')
->setIdentity($this->getRequest()->getPost('user'))
->setCredential($this->getRequest()->getPost('pass'));
```

Agregando cláusulas al where

Tenemos un campo extra en nuestra base de datos que determina si un usuario esta activado o no, por lo que para loguearse además de tener el user y pass correcto debe verificar que *activado=1*. Este requerimiento lo podemos controlar de la siguiente manera:

Código fuente

```
// obtenemos el query sobre el que trabaja el adapter
$select = $adapter->getDbSelect();
// le agregamos nuestra condición al where
$select->where('activado = 1');
```

Cuando ejecute la consulta para verificar que el user y pass son correctos, también controlará que *activado=1*.

Realizando la autenticación

Teniendo configurado el adapter ya tenemos todo listo para controlar que los datos sean correctos. El método authenticate() de *Zend_Auth* recibe el adapter y devuelve una instancia de *Zend_Auth_Result* a la que le podemos preguntar si el login fue exitoso o no:

```
$result = Zend_Auth::getInstance()->authenticate($authAdapter);
```

Zend_Auth_Result

Contiene toda la información sobre el intento de login:

- Si fue válido o no.
- Mensajes de error en caso de que algo haya fallado.
- Información sobre que fue lo que falló, etc.

Como su única función es devolver información tiene cuatro métodos:

1. **isValid()**: para saber si el login fue correcto o no.
2. **getCode()**: devuelve el código del error
3. **getIdentity()**: el nombre de usuario usado en el intento de autenticación
4. **getMessages()**: los mensajes de error, si hubo alguno

Como dijimos $result contiene los resultados de la autenticación del usuario.

Código fuente

```
if ($result->isValid()) {
// los datos del usuario son correctos
$this->_redirect("/bienvenido");
} else {
// datos incorrectos, podríamos mostrar un mensaje de error
switch ($result->getCode()) {
case Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND:
// usuario inexistente
break;
case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
// password erroneo
break;
default:
/* otro error */
break;
}
```

```
}
```

Persistiendo datos en sesión

Por defecto, Zend_Auth persiste los datos en sesión por medio de Zend_Session. Podemos obtener el storage que representa al namespace de Zend_Auth con:

```
Zend_Auth::getInstance()->getStorage()
```

En muchas ocasiones nos interesa almacenar los datos de la base de datos correspondientes al usuario actual. Esto lo podemos lograr por medio del método getResultRowObject() de Zend_Auth_Adapter_DbTable.

Pedimos estos datos y los guardarmos en sesión para acceder a ellos cuando los necesitemos:

```
$storage = Zend_Auth::getInstance()->getStorage();
$bddResultRow = $authAdapter->getResultRowObject();
$storage->write($bddResultRow);
```

Luego, en otro lugar de nuestra aplicación accedemos a estar información con getIdentity():

```
$infoUsuario = Zend_Auth::getInstance()->getIdentity();
```

Preguntar por el usuario actual y finalizar sesión

Dos cosas que debemos hacer, consultar si hay algún usuario logueado:

```
$auth = Zend_Auth::getInstance();
if ($auth->hasIdentity()) {
echo "Estas logueado";
} else {
echo "No estas logueado";
}
```

La otra es realizar el logout:

```
public function logoutAction() {
Zend_Auth::getInstance()->clearIdentity();
}
```

Capítulo 10

Revisión de componentes

Capítulo 10

Revisión de componentes

En el capítulo final de esta introducción a Zend Framework revisamos componentes útiles, su características y algunos ejemplos.

Zend_Cache

¿Qué podemos cachear?

El contenido del buffer de salida, el resultado de una función, de un método o un objeto propiamente dicho, etc. ¿Dónde podemos guardar la cache? En el sistema de archivos, en una base de datos, en un servidor memcached, etc.

Las distintas cosas que podemos cachear se denominan Frontends y el lugar donde guardamos la cache se denomina Backend. De esta forma, tenemos los frontends:

- Zend_Cache_Frontend_Ouptup, Zend_Cache_Frontend_Function
- Zend_Cache_Frontend_Class, etc., y los backedns Zend_Cache_Backend_File
- Zend_Cache_Backend_Sqlite, Zend_Cache_Backend_Memcached, etc.

Frontend Options

Los distintos frontends tienen las siguientes opciones de configuración:

- **caching**: Habilitar o deshabilitar la cache
- **lifetime**: El tiempo de vida de la cache
- **automatic_cleaning_factor**: Configura cada cuanto se limpia la cache

Backend Options

Por su parte, los distintos backends presentan las siguientes opciones de configuración:

- **cache_dir**: Directorio en el que se guardaran los archivos de la cache.

- **hashed_directory_level**: Permite configurar la cantidad de niveles de directorios, es muy útil cuando tenemos varios miles de archivos.

Pasemos a ver un ejemplo:

Código fuente

```
$frontend = "Core";
$backend = "File";
$frontendOptions = array("lifetime" => 7200);
$backendOptons = array("cacheDir" => "/cacheFiles");
$cache = Zend_Cache::factory($frontend, $backend, $frontendOptions,
$backendOptions);
if (!$result = $cache->load("queryPesado")) {
/* ejecutamos la consulta pesada y la cacheamos para próximos pedidos */
$cache->save($result, 'queryPesado');
}
return $result;
```

Zend_Debug

Se podría decir que *Zend_Debug::dump()* es una versión mejorada del típico *var_dump()* formatea el código, permite agregar a la expresión recibida un label para identificar un dump de otro, etc.

Su sintaxis es:

```
Zend_Debug::dump($var, $label = null, $echo = true);
```

Zend_Paginator

Zend_Paginator nos permite paginar cualquier tipo de información, por ejemplo permitiéndonos trabajar directamente con un *Zend_Db_Select* o un simple array de PHP.

Trabajaremos con Zend_Db_Select's:

Código fuente

```
// en el controlador...
// al factory le pasamos una instancia de Zend_Db_Select, el limit sera seteado
de acuerdo a los valores que luego pasemos a setCurrentPageNumber() y
setItemCountPerPage()
$paginator = Zend_Paginator::factory($select);
// seteamos la página actual y la cantidad de items mostrados por página
$paginator->setCurrentPageNumber($page)->setItemCountPerPage($count);
//lo asignamos a una variable de la vista
$this->view->paginator = $paginator;
// en la vista...
// podemos recorrer $paginator, el cual implementa un iterator que contendrá
los
resultados del query que recibió como parámetro
foreach ($paginator as $value) {
echo $value["campo_bdd"];
}
// paginator.phtml es el partial encargado de armar la paginación
echo $this->view->paginationControl($this->paginator, 'Sliding',
'paginator.phtml')
// paginator.phtml
<?php if ($this->pageCount): ?>
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
<a href="><?php echo $this->url(array(&amp;#39;page&amp;#39; => $this->previous)); ?>">
&amp;lt; Previous
```

```
</a> |
<?php else: ?>
<span class="disabled">&lt; Previous>/span> |
<?php endif; ?>
<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
<?php if ($page != $this->current): ?>
<a href=">?php echo $this->url(array(&#39;page&#39; => $page)); ?>">
<?php echo $page; ?>
</a> |
<?php else: ?>
<?php echo $page; ?> |
<?php endif; ?>
<?php endforeach; ?>
<!-- Next page link -->
<?php if (isset($this->next)): ?>
<a href="<?php echo $this->url(array(&#39;page&#39; => $this->next)); ?>">
Next &gt;
</a>
<?php else: ?>
<span class="disabled">Next &gt;</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

Zend_Http_Client

Otro componente que nos va a servir conocer es *Zend_Http_Client*, el cual nos provee una potente interface para realizar HTTP requests por ejemplo para hacer uso de la API de algún servicio web, enviar datos por POST o GET, setear headers etc., permitiéndonos obtener los headers y el body de la respuesta obtenida. Posee varios adapters (socket, proxy y cURL) En este ejemplo haremos un request muy simple a Twitter para leer el el último update de @maestros

Código fuente

```
$client = new Zend_Http_Client('http://api.twitter.com/1/statuses/user_timeline
/maestros.json');
// seteamos el método y un parámetro GET
$client->setMethod("GET")->setParameterGet('count', '10');
// ejecutamos el request
$client->request();
// obtenemos la respuesta generada en el paso anterior
$response = $client->getLastResponse();
// si salio todo bien...
if ($response->getHeader("Status") == "200 OK") {
$data = Zend_Json::decode($response->getBody());
echo "Ultimo status: ".$data[0]["text"];
} else {
echo "Ocurrió un error al leer los datos.";
}
```

Zend_Mail

Un componente muy completo para el envío de mails. Enviar un mail puede ser tan simple como crear el objeto y setear destinatario, remitente y contenido. Entre sus funciones más avanzadas también nos permite usar para el envía un servidor SMTP (por defecto usa la funcion mail() de PHP), enviar mails con HTML, adjuntar archivos, etc. En el manual hay algunos ejemplos básicos en este artículo veremos uno un poquito más avanzado, usando Gmail como servidor smtp y enviando un archivo adjunto:

Código fuente

```
// configuración del smtp y datos para realizar la autenticación en el mismo
$config = array(
'auth' => 'login',
'username' => 'user@gmail.com',
'password' => 'pass',
'ssl' => 'ssl',
'port' => 465);
$transport = new Zend_Mail_Transport_Smtp('smtp.gmail.com',
$config);
$mail = new Zend_Mail();
$mail->setBodyText("Hola te envio este mail");
$mail->setFrom('zend_mail@prueba.com', 'Yo');
$mail->addTo('to@to.com');
$mail->setSubject('Probando envío');
// adjuntamos un archivo, $file es un archivo local
$at = $mail->createAttachment(file_get_contents("prueba.txt"));
// nombre del archivo adjunto
$at->filename = "archivo_adjunto.txt";
$mail->send($transport);
```

Finalizamos con la Guía Zend que tiene como objetivo introducirte al Framework abierto para el desarrollo de aplicaciones y servicios web. Espero que con estos 10 capítulos tus dudas hayan sido aclaradas y si aún tienes preguntas te invitamos a participar en los comentarios en línea en www.maestrosdelweb.com.

Otras guías

iPhone

Crea tu propia aplicación para dispositivos móviles basados en iOS, y descubre las posibilidades de trabajo que este mercado ofrece a los desarrolladores.

Visita la Guía iPhone

ASP.NET

ASP.NET es un modelo de desarrollo Web unificado creado por Microsoft para el desarrollo de sitios y aplicaciones web dinámicas con un mínimo de código. ASP.NET

Visita la Guía ASP.NET

Producción de Video

Tendencias en la creación de videos, herramientas, consejos y referencias en la producción de videos para la web.

Visita la Guía Producción de vídeo

Curso Android

Actualiza tus conocimientos con el curso sobre Android para el desarrollo de aplicaciones móviles.

Visita el curso Android

Mapas

Aprende a utilizar el API de Google Maps para el desarrollo de tus proyectos con mapas.

Visita la Guía Mapas

Startup

Aprende las oportunidades, retos y estrategias que toda persona debe conocer al momento de emprender.

Visita la Guía Startup

Adictos a la comunicación

Utiliza las herramientas sociales en Internet para crear proyectos de comunicación independientes.

Visita Adictos a la comunicación