ZF - Session 2

1. Models

Será aquí donde nuestra lógica de negocio será implementada. Un framework poco o nada puede ayudarnos en este punto ya que es imposible que conozca nuestra lógica de negocio. Es por ello que los modelos no suelen heredar de ninguna clase del framework. Tampoco estamos obligados a seguir ninguna nomenclatura especial en los nombres de clases y/o métodos ni a mantener una estructura de carpetas concreta.

Siguiendo con nuestro módulo de ejemplo, vamos ahora a crear el modelo.

Dentro de 'src/Calculator' creamos un directorio llamado 'Model' bajo el que crearemos nuestro modelo:

```php
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
 * @link        http://github.com/xenframework for the canonical source repository
 * @copyright   Copyright (c) xenFramework. (http://xenframework.com)
 * @license     MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Calculator\Model;

class CalculatorModel
{
    /**
     * @var int
     */
    private $op1;

    /**
     * @var int
     */
    private $op2;

    /**
     * @var int
     */
    private $result;

    public function __construct()
```

```
{
    $this->result = 0;
}

public function add()
{
    $this->result = $this->op1 + $this->op2;
}

public function subtract()
{
    $this->result = $this->op1 - $this->op2;
}

public function multiply()
{
    $this->result = $this->op1 * $this->op2;
}

public function divide()
{
    $this->result = (int) ($this->op1 / $this->op2);
}

public function getResult()
{
    return $this->result;
}

/**
 * @param int $op1
 */
public function setOp1($op1)
{
    $this->op1 = $op1;
}

/**
 * @return int
 */
public function getOp1()
{
    return $this->op1;
}

/**
 * @param int $op2
 */
public function setOp2($op2)
{
```

```
        $this->op2 = $op2;
    }

    /**
     * @return int
     */
    public function getOp2()
    {
        return $this->op2;
    }
}
```

Hemos elegido el nombre CalculatorModel para la clase por mantener la línea de los nombres de controllers pero no es obligatorio.

Nuestro modelo ya está listo para ser usado por un controller.

2. Controllers

Los 'controllers' contienen 'actions' que son los encargados comunicar vistas con modelos.

Una clase 'controller' no tiene por qué heredar de ninguna otra clase. Pero si vamos a hacer uso del 'Layout', las vistas, el 'router', el 'ServiceManager', ..., y otros muchos componentes, debemos heredar de la clase 'AbstractActionController'. Esta clase nos permitirá tener acceso a todos esos componentes y a un gran número de métodos que nos serán de utilidad en un 'action'.

Es importante mantener el código de un 'action' lo más ligero posible. Hemos de pensar que el 'controller' no tiene que conocer nada de nuestra lógica de negocio, puesto que ésta se encuentra en nuestros modelos. Supongamos el caso de tener que comprobar que la edad del usuario debe estar dentro de un rango. Si esta comprobación se realiza en un 'action', estaremos repitiendo código en cada 'action' que haga uso de ese valor. En cambio, si esta comprobación se hace en un modelo, únicamente se realizará ahí.

Crearemos un controller en el módulo Calculator.

Bajo la carpeta src/Controller creamos la clase CalculatorController:

```
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
```

```
 * @link       http://github.com/xenframework for the canonical source repository
 * @copyright  Copyright (c) xenFramework. (http://xenframework.com)
 * @license    MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Calculator\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class CalculatorController extends AbstractActionController
{

}
```

Hemos heredado de la clase 'AbstracActionController'.

Ahora vamos a añadir cada uno de los actions que tiene la app Calculator:

```
public function indexAction()
{
    return [];
}

public function addAction()
{
    return [];
}

public function addDoAction()
{
    return [];
}
```

Los actions deben tener la terminación Action.

Utilizaremos el action addAction para mostrar un formulario que permita al usuario introducir los operandos.

En el action addDoAction será donde llamemos al modelo pasándole los datos del formulario para que el modelo nos haga la operación y nos devuelva el resultado.

Ahora es cuando necesitamos al modelo. El modelo CalculatorModel es una dependencia de nuestro controller. Si creásemos la instancia dentro del controller estaríamos infrautilizando el framework.

Pensemos en que dos controllers tuvieran la misma dependencia. En ambos tendríamos el mismo código para crearla. Además pensemos en que esa dependencia puede tener a su vez más dependencias.

Para evitar esa repetición de código y hacer nuestro controller más independiente, facilitando así la realización de tests unitarios, hacemos uso de la Inyección de Dependencias (Dependency Injection).

La Inyección de Dependencias es una forma de IoC (Inversion of Control) en la que antes de crear nuestro controller le indicamos al framework qué dependencias vamos a necesitar y él las creará por nosotros.

Otra opción sería hacer uso del 'ServiceLocator' dentro del 'controller'. Aunque este patrón está considerado como un 'anti pattern', ya que hace al 'controller' dependiente del 'ServiceLocator' en sí mismo.

A continuación vamos a ver las dos maneras que tenemos para decirle al 'framework' que nos cree la dependencia que necesitamos, el modelo 'CalculatorModel'.

2.1. Constructor Injection

Crearemos una propiedad en el controller para almacenar el modelo. Le diremos al framework que nos inyecte el modelo por el constructor en el momento de crear la instancia del controller.

En este punto tenemos la clase CalculatorModel creada pero el framework aun no conoce su existencia. Vamos a darla de alta en el fichero module.config.php como un nuevo servicio.

```
'service_manager' => array(
    'invokables' => array(
        'Calculator\Model\Calculator' => 'Calculator\Model\CalculatorModel',
    ),
),
```

Como nuestro modelo no tiene ninguna dependencia puede ser un invokable. Ahora ya podemos recuperar nuestro modelo cuando lo necesitemos del contenedor de servicios mediante el ServiceLocator. La key que usaremos será 'Calculator\Model\Calculator' tal y como hemos indicado.

Ahora en el controller vamos a crear la propiedad model y el método __construct.

```
/**
 * @var CalculatorModel
 */
private $model;

/**
```

```
     * @param CalculatorModel $model
     */
    public function __construct(CalculatorModel $model)
    {
        $this->model = $model;
    }
```

Lo siguiente es crear un factory para que el framework cree las instancia de este controller a través de él.

```
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
 * @link        http://github.com/xenframework for the canonical source repository
 * @copyright   Copyright (c) xenFramework. (http://xenframework.com)
 * @license     MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Calculator\Controller\Factory;


use Calculator\Controller\CalculatorController;
use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class CalculatorFactory implements FactoryInterface
{

    /**
     * Create service
     *
     * @param ServiceLocatorInterface $serviceLocator
     *
     * @return mixed
     */
    public function createService(ServiceLocatorInterface $serviceLocator)
    {
        $sm = $serviceLocator->getServiceLocator();
        $model = $sm->get('Calculator\Model\Calculator');

        return new CalculatorController($model);
```

```
    }
}
```

Es importante que nuestro Factory implemente la interfaz FactoryInterface. Esta interfaz nos obliga a implementar el método createService, que es el que el framework buscará a la hora de crear el controller.

Dentro de este método obtenemos una instancia del ServiceManager para así poder recuperar el modelo del contenedor de servicios.

Por último devolvemos una nueva instancia del controller pasándole el modelo por el constructor.

El framework aun no sabe que queremos usar este factory para crear nuestras instancias del controller. Vamos a indicárselo ahora.

```
'controllers' => array(
    'invokables' => array(
        'Calculator\Controller\Index' => 'Calculator\Controller\IndexController',
    ),
    'factories' => array(
        'Calculator\Controller\Calculator' =>
'Calculator\Controller\Factory\CalculatorFactory',
    ),
),
```

Vemos que nuestro controller está ahora bajo la key factories.

Ya que estamos en el fichero module.config.php vamos a aprovechar para crear las rutas de nuestros actions.

```
'router' => array(
    'routes' => array(
        'calculator_home' => array(
            'type' => 'Literal',
            'options' => array(
                'route'    => '/calculator/',
                'defaults' => array(
                    'controller' => 'Calculator\Controller\Calculator',
                    'action'     => 'index',
                ),
            ),
        ),
        'calculator_add' => array(
            'type' => 'Literal',
            'options' => array(
                'route'    => '/calculator/add/',
                'defaults' => array(
                    'controller' => 'Calculator\Controller\Calculator',
                    'action'     => 'add',
```

```
            ),
        ),
    ),
    'calculator_addDo' => array(
        'type' => 'Literal',
        'options' => array(
            'route'    => '/calculator/add-do/',
            'defaults' => array(
                'controller' => 'Calculator\Controller\Calculator',
                'action'     => 'addDo',
            ),
        ),
    ),
  ),
),
```

Y el código del controller quedaría así:

```
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
 * @link        http://github.com/xenframework for the canonical source repository
 * @copyright   Copyright (c) xenFramework. (http://xenframework.com)
 * @license     MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Calculator\Controller;


use Zend\Mvc\Controller\AbstractActionController;
use Calculator\Model\CalculatorModel;

class CalculatorController extends AbstractActionController
{
    /**
     * @var CalculatorModel
     */
    private $model;

    /**
     * @param CalculatorModel $model
     */
```

```
    public function __construct(CalculatorModel $model)
    {
        $this->model = $model;
    }

    public function indexAction()
    {
        $title = $this->layout()->getVariable('title');
        $newTitle = $title . ' - Calculator Home';
        $this->layout()->setVariable('title', $newTitle);

        return ['title' => $newTitle];
    }

    public function addAction()
    {
        $title = $this->layout()->getVariable('title');
        $newTitle = $title . ' - Calculator Add';
        $this->layout()->setVariable('title', $newTitle);

        return ['title' => $newTitle];
    }

    public function addDoAction()
    {
        $this->model->setOp1($this->params()->fromPost('op1'));
        $this->model->setOp2($this->params()->fromPost('op2'));
        $this->model->sum();

        $title = $this->layout()->getVariable('title');
        $newTitle = $title . ' - Calculator Add Result';
        $this->layout()->setVariable('title', $newTitle);

        return ['result' => $this->model->getResult(), 'title' => $newTitle];
    }
}
```

En este ejemplo vemos cómo utilizar el modelo y cómo recuperar variables del layout y pasar variables a la vista.

3. Views

Las vistas son el componente MVC que se encarga de mostrar o recoger información por parte del usuario.

Si en el controller hemos dicho que no debe haber código que haga referencia a la lógica de nuestra aplicación, en las vistas aún menos.

Dentro de un módulo hay una carpeta llamada view destinada a alojar las vistas del módulo. Debemos crear una carpeta que se llame igual que el módulo. Los nombres de carpetas bajo view son en minúsculas ya que no tendremos clases en esta carpeta, y por lo tanto, no crearemos namespaces para ellas.

Dentro de la carpeta con el nombre del módulo debemos crear otras con el nombre de cada controller. Dentro de ella tendremos un fichero .phtml para cada action del controller.

Si el nombre del controller es compuesto, entonces en el nombre de la carpeta aparecerán las palabras separadas por un guión '-'. Lo mismo sucede con los actions.

En el fichero module.config.php tenemos una sección para configurar las vistas.

Se trata de la key 'view_manager'. Aquí podemos crear alias para nuestras vistas bajo la key 'template_map'. El contenido mínimo para esta key es el siguiente:

```
'view_manager' => array(
    'template_path_stack' => array(
        __DIR__ . '/../view',
    ),
),
```

Donde le estamos indicando el directorio bajo el cual encontrar las vistas para este módulo.

Esta es la configuración del view_manager que viene por defecto en el módulo Application:

```
'view_manager' => array(
    'display_not_found_reason' => true,
    'display_exceptions'       => true,
    'doctype'                  => 'HTML5',
    'not_found_template'       => 'error/404',
    'exception_template'       => 'error/index',
    'template_map' => array(
        'layout/layout'           => __DIR__ . '/../view/layout/layout.phtml',
        'application/index/index' => __DIR__ . '/../view/application/index/index.phtml',
        'error/404'               =>    DIR    . '/../view/error/404.phtml',
        'error/index'             => __DIR__ . '/../view/error/index.phtml',
    ),
    'template_path_stack' => array(
        __DIR__ . '/../view',
    ),
),
```

Donde aparte de la key 'template_path_stack' podemos crear por ejemplo, alias a algunas vistas o partials bajo la key 'template_map'. Además podemos configurar otros parámetros de la vista como el 'doctype'.

Por defecto ZF2, cuando vaya a renderizar buscará una vista que se llame igual que el action en la ruta que hemos indicado antes. Pero podemos cambiar la vista de cualquier action y entonces es cuando nos vienen bien los alias, ya que no tendremos que escribir toda la ruta.

```
public function updateAction()
{
    $id = $this->params()->fromRoute('id');
    $user = $this->model->getById($id);

    $view = new ViewModel();
    $view->setTemplate('user/admin/add.phtml');
    $view->title = 'Upate User';
    $view->action =  $this->url()->fromRoute('user\admin\updateDo');
    $view->user = $user;

    return $view;
}
```

4. Layouts

Un layout es la parte común a todas las vistas. Por ejemplo, observaremos que en ninguna vista se incluyen las etiquetas <html><head> ... Ya que esto es común a todas las vistas lo sacamos fuera y lo colocamos en un único fichero, el layout.

Cada módulo puede definir su propio layout.

Por defecto ZF2 utilizará el alias 'layout/layout' que buscará en el 'view_manager' del fichero module.config.php.

Los layouts en realidad son vistas y como tales, pueden tener variables y vistas hijas.

El layout aplicable a una respuesta será el definido en el último módulo cargado. A no ser, que en algún método onBootstrap de algún módulo hayamos dicho lo contrario.

Para recuperar el layout en el método onBootstrap haríamos lo siguiente:

```
public function onBootstrap(MvcEvent $e)
{
    $sm = $e->getApplication()->getServiceManager();
    $config = $sm->get('config');
    $layout = $e->getViewModel();
    $layout->setVariable('title', $config['application']['name']);
}
```

En cambio en un action haríamos:

```
public function indexAction()
{
    $title = $this->layout()->getVariable('title');
    $newTitle = $title . ' - Calculator Home';
    $this->layout()->setVariable('title', $newTitle);

    return ['title' => $newTitle];
}
```

Para poder utilizar esa variable de layout en una vista tendríamos que usar el view helper layout():

```
<?php echo $this->layout()->title; ?>
```

Si quisiéramos tener un layout diferente para cada módulo podríamos crearnos un módulo únicamente para esta tarea. Utilizaríamos el método onBootstrap de este nuevo módulo con el siguiente código:

```
public function onBootstrap(MvcEvent $e)
{
    $e->getApplication()->getEventManager()->getSharedManager()-
>attach('Zend\Mvc\Controller\AbstractController', 'dispatch', function($e) {
        $controller = $e->getTarget();
        $controllerClass = get_class($controller);
        $moduleNamespace = substr($controllerClass, 0, strpos($controllerClass, '\\'));
        $controller->layout($moduleNamespace . '/layout');
    }, 100);
}
```

Donde cogemos el controller que va a responder y le cambiamos el layout dependiendo del módulo en el que se encuentre.

5. Routing

El Routing es el mapeo de la url de la petición actual a un action de un controller. De ello se encarga el router. Este componente también puede ser utilizado en el sentido inverso, podemos generar una url a partir de un identificador de ruta o de un controller y un action.

5.1. Creando rutas

El mapeo de las rutas con los controllers y actions tiene lugar en el fichero module.config.php del módulo. Bajo la key 'router' creamos un array 'routes' donde registraremos las rutas que tienen respuesta en controllers del módulo.

```
return array(
    'router' => array(
```

```
    'routes' => array(
        'home' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/',
                'defaults' => array(
                    'controller' => 'Application\Controller\Index',
                    'action'     => 'index',
                ),
            ),
        ),
```

'home' es el identificador de la ruta que posteriormente puede ser utilizado en el View Helper Url() para generar la url correspondiente. Este identificador debe ser único dentro de todos los módulos, ya que en caso de no ser así, prevalecerá el último módulo en cargar.

Para cada ruta debemos especificar las keys 'type' y 'options'.

En 'type' indicamos el tipo de ruta, en el ejemplo 'Literal'. Podemos poner el Namespace completo del tipo de ruta o también podemos poner solamente 'Literal'. Una ruta 'Literal' es una ruta sin parámetros, estática.

Bajo 'options' indicamos el path de la ruta, en este caso '/'. En 'defaults' indicamos el controller y el action que se llamará cuando la petición se corresponda con esta ruta. En controller ponemos el identificador de controller que hemos dado de alta en la sección 'controllers' de este mismo fichero de configuración:

```
'controllers' => array(
    'invokables' => array(
        'Application\Controller\Index' => 'Application\Controller\IndexController'
    ),
),
```

5.1.1. Child routes

Podemos crear rutas hijas a partir de una ruta.

```
return array(
    'router' => array(
        'routes' => array(
            'account' => array(
                'type' => 'Literal',
                'options' => array(
                    'route'    => '/account/',
                    'defaults' => array(
                        'controller' => 'account',
                        'action'     => 'index',
                    ),
```

```
            ),
            'may_terminate' => true, // no other segments will follow it
            'child_routes' => array(
                'paginator' => array(
                    'type' => 'Segment',
                    'options' => array(
                        'route' => 'page/:page/',
                        'constraints' => array(
                            'page' => '[0-9]+',
                        ),
                        'defaults' => array(
                            'controller' => 'account',
                            'action'     => 'index',
                        ),
                    ),
                ),
            ),
        ),
    ),
```

Aquí estamos indicando que la ruta 'account' con path '/account/' puede darse así tal cual (de ahí la key 'may_terminate' a true) o bien puede estar seguida de las rutas hijas. En este caso tenemos una ruta hija llamada 'paginator' que es de tipo 'Segment'. Las rutas de tipo 'Segment' tienen una parte variable. En este caso la parte variable es el número de página y comienza con ':'. Si además tiene está entre corchetes [], estamos indicando que es opcional. Vemos también el uso de la key 'constraints' para indicar mediante expresiones regulares los valores válidos para la parte variable de la ruta.

Hemos visto los tipos de ruta 'Literal' y 'Segment' que son los más comúnmente usados. Para más información sobre el resto de tipos de rutas debemos consultar el manual oficial:

http://framework.zend.com/manual/current/en/modules/zend.mvc.routing.html

5.2. Usando rutas en Controllers

El utilizar rutas generadas por el Router nos permite no tener que actualizar nuestro código en caso de que cambiemos la definición de nuestras rutas.

5.2.1. Obteniendo los parámetros de una ruta

En una ruta de tipo 'Segment' podemos acceder a sus parámetros desde un controller de la siguiente manera:

```
$page = $this->params()->fromRoute('page');
```

5.3. Usando rutas en Views

Para generar una ruta en una vista usamos el View Helper Url():

```
<div>
   <ul>
      <li><a href="<?php echo $this->url('calculator_home'); ?>">Home</a></li>
      <li><a href="<?php echo $this->url('calculator_add'); ?>">Sum</a></li>
   </ul>
</div>
```

Para cuando tengamos rutas Child usamos la barra '/' en el identificador:

```
$this->url('account/paginator', array('page' => 1))
```

En el segundo argumento podemos pasar un array con los parámetros de la ruta.