

Atelier Zend Framework : Utilisation du helper ViewRenderer

par Julien Pauli ([Tutoriels](#), [article et conférences PHP et développement web](#)) ([Blog](#))

Date de publication : 28/06/2007

Dernière mise à jour : 21/01/2008

Depuis la version 1.0RC1 est apparu un nouveau ActionHelper : le ViewRenderer. Son but est tout simple : s'occuper de la vue à votre place, et ainsi automatiser sa configuration, et son rendu.

Ainsi, toute action rend une vue sans qu'on s'en occupe, évidemment c'est très flexible : désactivable et personnalisable :

- I - Introduction
- II - Dans le vif du sujet
- III - Le fonctionnement de VR, cas par défaut
- IV - La personnalisation du ViewRenderer
- V - Conclusions

I - Introduction

La classe `Zend_Controller_Action` est munie d'un système de helpers. Ce système permet de créer des objets qui vont exécuter des traitements dans toutes les actions, qui vont les "aider", d'où le terme de "helper".

Zend Framework possède dans sa distribution de base quelques helper, nous allons étudier l'un d'entre eux : le `ViewRenderer` (abrégé par la suite "VR").

Parti du constat que quasiment toutes les actions nécessitent l'affichage d'une vue, le VR sert à automatiser ce processus. Et c'est tant mieux, car même s'il paraît complexe (vous allez voir, il n'en est rien), il permet d'écrire encore moins de code qu'auparavant, et rend de précieux services.

II - Dans le vif du sujet

Ce helper est automatiquement activé dans les actions, à moins de spécifier le contraire. Il est créé et activé lors de l'appel à `ZC_Front::dispatch()`.

Une des caractéristiques essentielles du VR est qu'il automatise totalement le processus de rendu de la vue, ainsi, toute action mène systematiquement à l'affichage d'une vue, qui doit être stockée dans un dossier précis.

Vous allez voir que comme à son habitude, ZF nous offre une vraie possibilité de personnalisation. Alors que le VR impose que l'on respecte un certain schéma dans son design applicatif, notamment au regard de l'arborescence du site; il est complètement personnalisable. Premièrement, il existe une méthode qui désactive totalement le VR :

```
Zend_Controller_Front::getInstance()->setParam('noViewRenderer', true);
```

Ici on dit au frontController de ne pas utiliser VR. Il va propager cet état dans tout le système MVC, et votre application fonctionnera donc sans rendu de vue, à moins de tout faire manuellement (VR est apparu avec ZF 1.0RC1)

Mais il est tout de même idiot de se passer de cette aide précieuse, que je vais tout de suite vous détailler.

III - Le fonctionnement de VR, cas par défaut

Nous allons étudier le fonctionnement de VR dans son état initial, c'est à dire tel que fourni dans ZF, sans aucune manipulation.

- VR rend automatiquement une vue dans l'action actuelle, sauf si celle-ci est redirigée (forward ou redirect)
- VR ne rend qu'un seul script de vue
- VR cherche ce script dans :moduleDir/views
- VR rend le script :controller/:action.:suffix

Ainsi, si j'appelle `http://monsite.com/monapp/membres/liste` , la vue qui va être rendue automatiquement sera représentée par le fichier `monapp/membres/views/scripts/liste/index.phtml`.

Ceci en accord avec la structure conseillée pour utiliser ZF, qui je vous rafraichis la mémoire, ressemble à ceci :

```
docroot/  
  index.php  
application/  
  default/  
    controllers/  
      IndexController.php  
      FooController.php  
  blog/  
    controllers/  
      IndexController.php  
    models/  
    views/  
      helpers/  
      filters/  
      scripts/  
  news/  
    controllers/  
      IndexController.php  
      ListController.php  
    models/  
    views/  
      helpers/  
      filters/  
      scripts/
```

Il est impératif, pour comprendre le fonctionnement du VR, de se rendre compte que l'action pilote le helper VR, qui lui-même va piloter la vue.

Du coté de l'objet vue en question, rien ne change, sa structure reste la suivante, par défaut :

```
views/  
  helpers/  
  filters/  
  scripts/
```

On va y aller doucement. Le VR initialise la vue dans le `dispatch`, il la configure, en lui passant les bons dossiers de fonctionnement basés sur l'action en cours de dispatching.

Il crée ainsi de lui-même un objet Vue, il lui ajoute un basePath en correspondance avec le modèle de VR : :moduleDir/views.

L'action dispatchée, le postDispatch demande alors à VR d'afficher la vue par défaut. Le basePath ayant été spécifié, VR demande à son tour à l'objet Vue de se rendre.

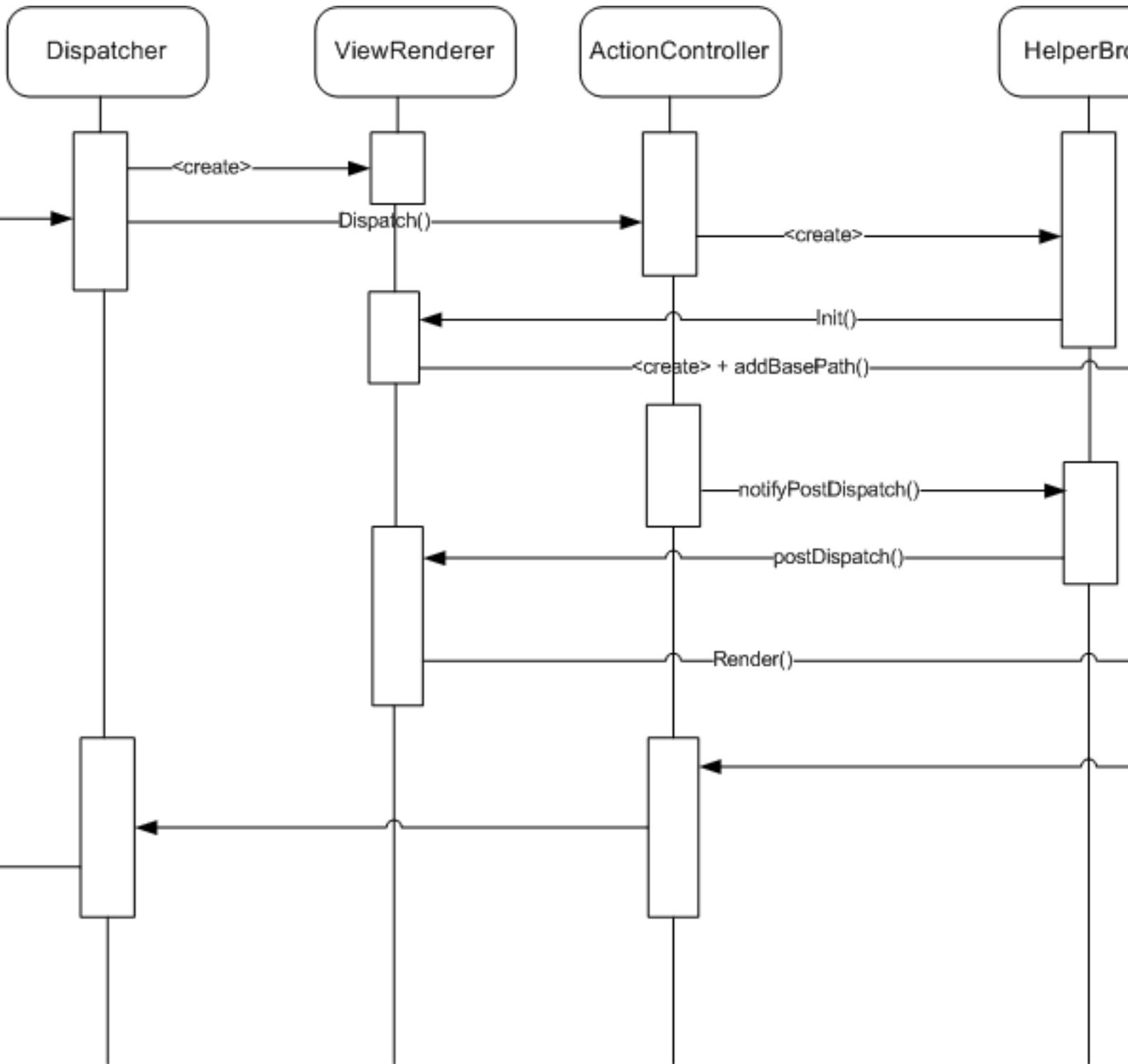


Diagramme de séquence simplifié

Sur le diagramme de séquence (dont le processus MVC a été simplifié), on se rend compte que le processus de dispatching rend automatiquement la vue.

Il est important de noter que le VR s'occupe de la vue, l'ActionController n'a en théorie pas à appeler **render()**, s'il devait le faire, la méthode est proxifiée vers le VR.

Maintenant que nous avons vu le fonctionnement du VR, voyons un peu comment on va pouvoir l'adapter à nos besoins, car nous n'avons pas forcément envie de suivre tout ce processus par défaut.

IV - La personnalisation du ViewRenderer

Dans un premier temps, on veut pouvoir ne pas rendre de vue. Dans ce cas, dans une action, ça s'effectue comme ceci :

```
<?php
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

Cette action ne sera pas rendue, si vous voulez ne rendre aucune action dans ce contrôleur, utilisez la même syntaxe, mais dans le `preDispatch()`.

Si vous souhaitez à ce stade là désactiver VR pour tout le reste du processus (éventuellement les actions suivant celle-ci), utilisez la méthode **`setNeverRender(true)`**.

Voyons un peu la personnalisation du reste, avec un bootstrap complet, pour être plus "réaliste" :

Structure des répertoires utilisée

```
www/
  index.php
app/
  config/
    config.ini
  default/
    controllers/
      IndexController.php
    views/
      helpers/
        foo.php
      filters/
      scripts/
        vueA.phtml
        vueB.phtml
  exemple/
    controllers/
      IndexController.php
      AdminController.php
      WebserviceController.php
    models/
      ExempleTable.php
    views/
      helpers/
        bar.php
      filters/
      scripts/
        indexfoo.phtml
```

index.php

```
<?php
$ds = DIRECTORY_SEPARATOR; // raccourci
$appDir = realpath(dirname(dirname(__FILE__))) . $ds . 'app' . $ds;
set_include_path($appDir . PATH_SEPARATOR . get_include_path());
```

index.php

```
require ('Zend/Loader.php');
Zend_Loader::registerAutoload(); // enregistrement d'autoload sur la pile

$config = new Zend_Config_Ini($app . 'config' . 'config.ini', 'exemple', true);

$db = Zend_Db::factory('PDO_MYSQL', $config->db->toArray()); // création de la connexion
Zend_Db_Table::setDefaultAdapter($db); // envoie l'adapter à toutes les classes de l'ORM qui vont
l'utiliser

$frontController = Zend_Controller_Front::getInstance(); // récupère le FrontController

$frontController->setControllerDirectory(array(
    'default' => $appDir . 'default' . $ds . 'controllers', // default module
    'exemple' => $appDir . 'exemple' . $ds . 'controllers', // exemple module
));

$view = new Zend_View($config->view->toArray()); // Initialisation de la vue principale
$view->strictVars(); // active le tracking des variables non utilisées ou mal écrites
$view->setBasePath($appDir . $ds . 'default' . $ds . 'views'); // repertoire par défaut des vues

$viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');
$viewRenderer->setView($view)
    ->setViewBasePathSpec(':moduleDir/views') // facultatif => comportement par défaut
    ->setViewScriptPathSpec(':controller:action:suffix') // par défaut :
:controller/:action:suffix
    ->setViewScriptPathNoControllerSpec(':action:suffix') // facultatif => comportement
par défaut
    ->setResponseSegment('body');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);*/

$frontController->setParam('noErrorHandler', true); // on ne veut pas du plugin de gestion des
erreurs
$frontController->throwExceptions(true); // renvoie les exceptions au lieu de les ajouter à la
réponse

try { // boucle de dispatching
    $frontController->dispatch(); // dispatche !
} catch (Exception $exception) { // attrape toute exception
    exit($e->getMessage()); //sortie pour le développement
}
```

config.ini

```
[exemple]
view.encoding      = "UTF-8"
view.escape        = "htmlentities"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

db.username        = myname
db.password        = mypassword
db.dbname          = exemple
db.host            = myhost
```

J'ai ici montré une application qui non seulement utilise sa vue personnelle (et ne se repose pas sur VR pour la créer), mais en plus utilise des dossiers personnalisés.

default est le module par défaut, c'est réellement lui que le frontController utilise lorsqu'aucun module n'est appelé dans l'URL (c'est changeable). Moi, j'ai décidé qu'il ne comporte pas de dossiers métiers (/models), il ne sert qu'à héberger la page d'accueil (qui sera indexAction(), dans default/indexController), ainsi que quelques vues dont, en revanche, je veux pouvoir me servir depuis n'importe quel contrôleur (on va voir comment).

Ces vues contiendront par exemple des listes qu'elles fetcherons directement de certains modèles, ou alors de la présentation de données...

Nous configurons donc le basePath de la vue à `app/default/views/`. Je n'ai spécifié aucune option de configuration à la vue (`$view`) concernant ses dossiers internes. Elle garde donc par défaut `chemindunvue/helpers - /filters` et `/scripts`.

Je lui dis que la fonction d'échappement à utiliser est `htmlentities`, et que l'encodage de sortie est l'UTF8. Je lui dis tout ça avant de la passer au VR, qui je vous le rappelle va l'utiliser dans les actions dispatchées.

`setView()` enregistre l'objet vue dans le VR pour qu'il l'utilise dans les actions, ici ma vue utilise l'encodage UTF-8

`setViewBasePathSpec()` enregistre le chemin du dossier des vues, je n'ai fait aucun changement moi, juste pour vous montrer

`setViewScriptPathSpec()` enregistre le chemin utilisé après le BasePath, pour trouver le fichier de la vue

`setViewScriptPathNoControllerSpec()` enregistre le chemin utilisé après le BasePath, pour trouver le fichier de la vue, lorsque la variable `noController` est à `true` (on va voir cela)

`setViewSuffix()` enregistre l'extension des fichiers de vues, par défaut il s'agit de `'.phtml'`

`setResponseSegment()` spécifie au VR que la vue qu'il va utiliser doit se rendre dans un segment particulier de la réponse, plutôt que `'default'` (par défaut). J'ai choisis de l'appeler `'body'`.

On peut utiliser comme patterns pour les chemins : `:moduleDir` , `:module` , `:controller` , `:action` , `:suffix`. Avec ceci on peut dessiner les chemins que l'on désire utiliser.

Avec les paramètres que je viens d'insérer, les contrôleurs se comportent comme suit :

```
<?php
class Exemple_IndexController extends Zend_Controller_Action
{
    public function fooAction()
    {
        $table = new ExempleTable();
        $this->view->nbreVisites = $table->count('visites');
    }
}
```

ExempleTable provient du métier, des données y sont sélectionnées. Elles sont passées à la vue, dont on a pas besoin de s'occuper.

Le script `app/exemple/views/scripts/indexfoo.phtml` sera rendu, avec `nbreVisites` comme variable. VR va utiliser la vue que je lui ai passée en `index`.

```
<?php
class Exemple_IndexController extends Zend_Controller_Action
{
    public function fooAction()
    {
        $table = new ExempleTable();
```

```
$this->view->nbreVisites = $table->count('visites');
}

public function barAction()
{
    $this->view->error = true;
    $this->render('vueB', null, true);
    return;
}

public function foobarAction()
{
    $this->_helper->viewRenderer->setNoRender();
    $table = new ExempleTable();
    $visites = $ExempleTable->getVisites();
    $visites->traitement();
    $visites->save();
    return;
}
}
```

Comme le `basePath` de ma vue est déjà spécifié, `barAction` demande de rendre `vueB` directement. `$this->render()` dans une action est proxifiée vers le VR. Ainsi `$this->render(...)` est exactement équivalent à `$this->_helper->viewRenderer->render(...)`.

Je rappelle aussi que lorsque le VR entre dans une action, il exécute sur l'objet vue un **`addScriptPath()`** en correspondance avec les patterns que vous lui avez spécifié, et le contrôleur actuellement parsé. Ainsi dans une action, non seulement `app/default/views` est accessible, mais aussi `app/{monmodule}/views/`.

Ceci peut être démontré très rapidement :

```
<?php
class Exemple_IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        Zend_Debug::Dump($this->view);
        return;
    }
}
/* affiche :
object(Zend_View)#11 (17) {
    ["_path:private"] => array(3) {
        ["script"] => array(2) {
            [0] => string(55) "D:\sites\test\zf\viewrenderer\app\exemple\views\scripts\"
            [1] => string(52) "D:\sites\test\zf\viewrenderer\app\default\views\scripts\"
        }
        ["helper"] => array(3) {
            [0] => array(2) {
                ["prefix"] => string(23) "Exemple_View_Helper_"
                ["dir"] => string(55) "D:\sites\test\zf\viewrenderer\app\exemple\views\helpers\"
            }
            [1] => array(2) {
                ["prefix"] => string(17) "Zend_View_Helper_"
                ["dir"] => string(52) "D:\sites\test\zf\viewrenderer\app\default\views\helpers\"
            }
            [2] => array(2) {
                ["prefix"] => string(17) "Zend_View_Helper_"
                ["dir"] => string(37) "D:\sites\include\zf\Zend\View\Helper\"
            }
        }
    }
}
... ..*/
```

Remarquez comme les chemins ont été rajoutés, et comme les view helpers ont été namespacés.

Remarquez aussi que les chemins sont empilés FILO (First In Last Out). Si dans mon module default j'ai une vue 'Mavue', et que je l'ai aussi dans exemple, ca sera celle de exemple qui sera utilisée, car le dossier du module exemple est au dessus du dossier du module default.

Dans foobarAction, je décide de ne rien rendre automatiquement, je désactive donc temporairement le rendu dans cette action.

Maintenant, l'histoire du "noController" : regardez mon render() dans barAction. J'ai fait un rendu en spécifiant noController à true (je rappelle la syntaxe de render() : render('script', 'segment_de_reponse', 'no_controller?').

Je dis donc à mon VR (car \$this->render() est proxifié vers VR->render()) d'utiliser le `ViewScriptPathNoControllerSpec` pour chercher le chemin et le nom de la vue. Ici, il s'agit bien de 'vueB', et non pas de 'indexfoobar', qui aurait été utilisé si j'avais fait un simple \$this->render().

En fait, c'est exactement ce à quoi sert ce fameux mode de rendu "noController", c'est pour les vues qui sont accessibles partout, quelque soit le contrôleur invoqué, on ne veut pas que son nom fasse parti du chemin ou du nom de la vue à rendre. Pour plus de commodités, je redéfinit ma méthode render() de mes classes ActionController de manière à ce que celles-ci rendent systématiquement avec un noController à true :

ma classe d'action

```
<?php
class mesActions extends Zend_Controller_Action{
    public function render($action = null, $name = null, $noController = true)
    {
        return parent::render($action, $name, $noController);
    }
}
?>
```

un controleur qui en hérite

```
<?php
class Exemple_IndexController extends mesActions
{
    public function foo2Action()
    {
        $this->render('vueA');
        $this->render('vueB');
    }
}
```

vueA.phtml

```
<p>coucou, voici la vueA, et elle appelle un petit helper :-)</p>
<?php echo $this->foo(); ?>
```

Tout simple. Notez aussi qu'un appel à render() dans une action, va désactiver le rendu automatique. Dans ma foo2Action, je rend vueA, puis vueB, et VR ne vas pas rendre ensuite indexfoo2 comme il l'aurait fait si je n'avais pas appelé de render().

Pour les helpers, c'est tout aussi simple. Les helpers dans default/views/helpers sont accessibles de n'importe où, mon helper 'foo' peut ressembler à ca :

foo.php


```
<?php
class Zend_View_Helper_Foo{
    public function foo(){
        return 'foohelper appelé';
    }
}
```

Souvenez vous que le "prefix" est celui par défaut, je ne l'ai pas modifié lorsque j'ai créé ma vue dans mon index.php.

Mon objet vue, piloté par VR, s'attend donc à trouver une classe Zend_View_Helper_nomduHelper. Pour mettre en évidence le namespacing, voyons le helper bar :

bar.php

```
<?php
class Exemple_View_Helper_Bar{
    public function bar(){
        return 'barhelper appelé';
    }
}
```

La classe doit avoir un nom qui débute par Exemple_, car le VR, lorsqu'il ajoute les chemins, utilise des  **namespaces** pour les helpers, ce qui est très pratique pour bien les séparer. Et ici il est clair que notre "bar" helper ne peut être appelé que depuis une vue exécutée dans le module Exemple. Ailleurs que dans le module Exemple, le chemin du helper n'est tout simplement pas ajouté à notre instance de vue, qui ne le connaîtra donc pas.

V - Conclusions

En conclusion, il faut bien se rappeler du schéma du VR. L'ActionController est peuplé du ViewRenderer, et c'est celui-ci qui va piloter les objets Vue. Toute manipulation sur la Vue dans l'ActionController est automatiquement déléguée au VR, qui va manipuler l'objet Vue à la place de l'action.

Le ViewRenderer rend donc de gros services, car il permet d'automatiser une partie non négligeable du processus de rendu. La vue, ainsi que le VR lui-même pouvant être totalement personnalisés (utilisation de Smarty, de répertoires peu orthodoxes ...), il est alors difficile de s'en passer.

Ceci sans compter son excellent tandem avec **Zend_Layout**, la solution de templating de Zend Framework.

Je vous quitte en vous soulignant que si vous avez quelques trous de mémoire, la documentation officielle est toujours présente, et toujours aussi bien faite :-)

Manipulation de l'objet Vue : Zend_View

Manipulation du helper ViewRenderer (descendre un peu)

Le modèle MVC

