

# Zend Framework - Utiliser le moteur de template de son choix

par JYT ([Les expériences Zend de Sekaijin](#)) ([Blog](#))

Date de publication : 26 octobre 2007

Dernière mise à jour :

Je fais partie de ces gens qui utilisent un moteur de templates. Les raisons en sont multiples. La première est qu'il est particulièrement compliqué dans un moteur de templates de faire du traitement. Ainsi, on n'est pas tenté de faire faire à la **vue** des choses qui ne lui sont pas dévolues.

- I - Introduction
- II - Include ou main page ?
- III - Un exemple simple
- IV - Un exemple de main page
- V - Ajouter la vue au Framework
- VI - D'autres moteurs
- VII - Zend Layout
- VIII - Helper
- IX - Pour tester

## I - Introduction

Je fais partie de ces gens qui utilisent un moteur de templates. Les raisons en sont multiples. La première est qu'il est particulièrement compliqué dans un moteur de templates de faire du traitement. Ainsi, on n'est pas tenté de faire faire à la **vue** des choses qui ne lui sont pas dévolues.

La **vue** présente un point c'est tout. Au **contrôleur** de lui fournir la matière. À chacun son boulot. Il est parfaitement possible de faire de même avec le système que propose Zend Framework. Mais il est tout aussi facile de mettre directement dans sa présentation quelque chose comme :


```
<?php echo Zend_Db::getInstance()->getMyUserName() ?>
```

Voire des choses encore plus complexes. Si la vue a besoin de **myUserName**, c'est au contrôleur de lui donner. L'évolutivité et la maintenabilité de l'application en dépendent. Je n'ai pas été le premier à me pencher sur ce problème. Et force est de constater que les moteurs de templates on leur aficionados.

J'utilise le plus fréquemment, le moteur **ETS** (Easy Template System) c'est le moteur utilisé entres autres par Marcopoly. J'ai aussi usé de Smarty. Les deux sont relativement proches dans leur syntaxe. ETS est un moteur ultra léger (89 521 octets) il n'utilise pas de système de cache. Je développe des applications qui sont très dynamiques et, dans mon cas, le cache est plus un handicap qu'un avantage. Ce n'est pas le cas de toutes les applications et utiliser un cache peut s'avérer efficace.

Comme je l'ai dit, je travaille en équipe et, suivant les projets, un moteur ou un autre est plus opportun, voire aucun. Je me suis donc demandé comment intégrer efficacement des moteurs à ZF. La littérature de ZF sur le sujet est plutôt légère et la solution proposée est totalement inefficace : elle revient à demander au programmeur d'écrire autant de fichiers *phml* comme on le ferait avec ZF sans template, plus tout autant de templates, plus du code pour relier le tout.

On trouve sur Internet quelques implémentations plus probantes. Je dois à un collègue (**Patrick Dubois**) une première intégration d'ETS dans ZF qui avait l'avantage de ne pas obliger à changer une seule ligne dans son contrôleur. Mais pour y parvenir, nous avons fait un méchant hack qui ne me paraissait pas viable. À la suite de cela, nous avons trouvé sur Internet une intégration de Smarty particulièrement bien faite mais qui nécessitait de changer le code du contrôleur.

Je trouvais la solution de Patrick séduisante et celle de Philippe Le Van ( **KitPages**) élégante. Concilier les deux serait une bonne chose. Je dois vous avouer que j'ai déroulé ZF en pas-à-pas pour arriver à comprendre comment il fonctionnait en interne, et particulièrement **Zend\_View**. Après avoir échauffé les neurones (j'espère ne pas en avoir perdu de trop au passage), je suis enfin arrivé à une solution qui a demandé une petite demi journée de travail de mise au point. Dans la foulée, avec l'aide de Patrick (on pense mieux à deux têtes), nous avons intégré à ZF cinq moteurs dans une demi journée.

Notre approche consiste à dériver **Zend\_View** et à remplacer l'instance de cette dernière dans le contrôleur, par la nôtre. Pour y parvenir, il va falloir respecter le travail de Zend. Lors de la phase préparatoire au lancement de l'action de votre contrôleur, ZF va instancier un objet **Zend\_View**. Puis dans votre action vous allez lui indiquer les valeurs à afficher. Éventuellement, vous donnez l'ordre à la vue d'effectuer un rendu, puis vous rendez la main au Front contrôleur qui va passer au rendu final. Donc si comme moi vous ne mettez toujours que le minimum dans votre action, votre code se contente de faire des affectations de valeurs dans la vue,. N'indiquant pas de rendu dans l'action, ZF va demander à la vue de rendre le modèle de vue correspondant au nom du contrôleur et de l'action. La proposition de Philippe Le Van nécessite d'appeler explicitement un rendu dans l'action. Pour ma part, je serais content de conserver le fonctionnement par défaut de ZF.

## II - Include ou main page ?

Une autre chose que je trouve pratique avec les templates, est de pouvoir les imbriquer. Avec phtml tel que proposé par ZF, pour avoir une application qui a toujours la même apparence et dont seul le contenu change, il est nécessaire d'en passer par un include d'un entête et d'un pied de page. C'est une solution simple qui existait bien avant PHP lui-même mais cette approche implique de gérer la cohérence entre ce qui est ouvert dans l'entête et ce qui doit être refermé dans le pied. Avec un modèle principal qui inclut le contenu on a un système de boîtes, qui est homogène. Ainsi, tout ce qui est ouvert dans un fichier est fermé dans ce fichier. On peut résumer ces deux approches ainsi :

```
<?php include 'header.phtml'; ?>
<h2>contenu de ma page</h2>
<?php include 'footer.phtml'; ?>
```

Et :

```
<html>
<body><h1>ma belle application</h1>
<?php include 'content.phtml'; ?>
</body>
</html>
```

Incontestablement, je préfère l'approche *main page*, qui comme on le voit peut très bien être mise en œuvre en phtml. Mais on a vu que ZF allait chercher à rendre le fichier **controller/action.phtml**. Si on adopte l'approche *main page*, on va être en contradiction avec ZF. Il va falloir adapter la vue pour qu'elle accepte de fonctionner ainsi tout laissant croire au contrôleur qu'elle fonctionne exactement comme une **Zend\_View**.

En supposant qu'on y parvienne, on a introduit alors un nouveau problème. Comment faire si on a un rendu à faire qui n'utilise pas la même *main page*, ou qui n'en utilise pas du tout ? De ce côté-là, inutile de fouiller la doc de ZF, il n'y a rien car **Zend\_View** ne connaît pas cette notion. Il sera alors nécessaire d'introduire une nouvelle fonctionnalité à la vue : **setMainTemplate**.

### III - Un exemple simple

En tout premier lieu, je vais essayer de faire une vue qui accepte la notion de *main template* et qui respecte le fonctionnement de ZF, à savoir pas d'intervention dans le contrôleur et pas d'appel sauvage dans un fichier de script comme le propose ZF. On va garder exactement la même architecture et les mêmes fichiers de script que ce que propose ZF mais, au lieu d'avoir à inclure l'entête et le pied dans toutes les pages, on fournira un fichier **main.phtml**.

#### Première étape : dériver de Zend\_View

```
<?php
/**
 *
 * @author Jean-Yves Terrien
 *
 */

Zend_Loader::loadClass('Zend_View');
class Fast_View_Phtml extends Zend_View
```

Prévoyant plusieurs moteurs, il me fallait trouver une façon de les nommer. Ce sera **Fast\_View\_Engine**. Ces vues ne sont pas les moteurs de template mais l'interface entre ZF et le moteur qui, elle, doit rester inchangée. Il n'est pas question ni de modifier le code d'un moteur ni de modifier le code de ZF.

J'ai dit que je voulais pouvoir écrire : **include 'content.phtml'**; or mon **content** va dépendre du script à rendre. J'ai donc besoin d'une variable dans ma vue pour identifier ce script. J'ai aussi dit que je devais pouvoir changer de *main template*. Et au passage, suivant le moteur, mes fichiers ne doivent pas toujours avoir le même type : **phtml** pour PHP, **tpl** pour Smarty et **html** pour ETS. La façon d'écrire un *include* dépend du moteur de template. Je ne me suis pas penché sur tous les moteurs pour mettre en #uvre l'approche *main template*. Je l'ai préparé dans mes moteurs mais il faut pour Smarty et phtml trouver comment on fait un *include* d'un fichier dont le nom est fourni par une variable. Dans le fichier joint, vous trouverez donc des exemples avec ou sans *main page* en fonction du moteur de template.

```
public $_content;
public $_mainTemplate;
protected $_suffix = 'phtml';
```

Arrivé là, il me faut surcharger une seule méthode :

```
/**
 * Includes the view script in a scope with only public $this variables.
 *
 * @param string The view script to execute.
 */
protected function _run()
{
    // récupère le chemin complet du template demandé
    $name = func_get_arg(0);
    // le template principal est considéré à la racine des templates
    // de l'application (ou du module).
    if (!isset($this->_mainTemplate)) $this->setMainTemplate(dirname(dirname($name)) .
'/main.phtml');
    // initialiser ets
    // on indique à la quel est le template à inclure
    $this->_content = $name;
    // rendu de la page
    include $this->_mainTemplate;
}
```

Un copier/coller de la méthode `_run` de `Zend_View`, une petite adaptation et le tour est joué. On va chercher la *main page* à la racine des vues. La méthode `_run` reçoit en argument le chemin complet vers le script à rendre. Il suffit donc de remonter d'un répertoire pour se trouver à la racine des vues, soit de l'application soit du module. Pour fixer la *main page*, j'ai dit que j'allais ajouter une méthode `setMainTemplate`, je l'ai donc utilisée. Comme pour la page principale, il peut être utile de permettre à qui en aurait besoin de connaître le suffixe utilisé. J'ajoute donc une méthode `getSuffix` et, pour faciliter l'écriture des scripts, avoir un membre qui donne le chemin d'inclusion n'est pas mal non plus. Ne nous privons pas.

Voici la classe en entier :

```
<?php
/**
 *
 * @author Jean-Yves Terrien
 *
 */

Zend_Loader::loadClass('Zend_View');
class Fast_View_Phtml extends Zend_View
{
    public $_content;
    public $_templatesDir;
    public $_mainTemplate;
    protected $_suffix = 'phtml';

    public function setMainTemplate($main)
    {
        $this->_mainTemplate = str_replace(chr(92), '/', $main);
    }

    public function getSuffix() {
        return $this->_suffix;
    }

    /**
     * Includes the view script in a scope with only public $this variables.
     *
     * @param string The view script to execute.
     */
    protected function _run()
    {
        // récupère le chemin complet du template demandé
        $name = func_get_arg(0);
        // le template principal est considéré à la racine des templates
        // de l'application (ou du module).
        if (!isset($this->_mainTemplate)) $this->setMainTemplate(dirname(dirname($name)) .
'/main.phtml');
        // initialiser ets
        // on indique à la vue quel est le template à inclure
        $this->_content = $name;
        $this->_templatesDir = dirname(dirname($name)) . '/';

        // rendu de la page
        include $this->_mainTemplate;
    }
}
```

## IV - Un exemple de main page

```
<html>
<body><h1>ma belle application</h1>
<?php include $this->_content; ?>
</body>
</html>
```

## V - Ajouter la vue au Framework

Reste maintenant à indiquer au front contrôleur d'utiliser notre vue. ZF n'a rien prévu pour ce remplacement. Impossible de lui indiquer quelle classe utiliser comme vue. Impossible aussi d'ajouter des plugins comme pour les actions. La seule solution : instancier la vue soi-même et la donner au contrôleur. Cela se passe au démarrage.

```
Zend_Loader::loadClass('Fast_View_Phtml');
self::$_instance->_view = new 'Fast_View_Phtml' ();
$suffix = self::$_instance->_view->getSuffix();
$viewManager = Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$viewManager->setView(self::$_instance->_view)->setViewSuffix($suffix);
```

J'ai justement prévu dans mon front contrôleur la possibilité d'ajouter des éléments de ce type en fonction de la configuration. J'ai donc ajouté une entrée dans mon fichier de paramètre et j'ai ajouté le nécessaire dans mon front contrôleur :

```
public static function setViewEngine ($name = null) {
    $className = ucfirst($name);
    if (null == $className) {
        $className = 'Zend_View';
        $suffix = 'phtml';
    } else {
        if (substr($className, 0, 10) != 'Fast_View_') {
            $className = 'Fast_View_' . $className;
        }
    }
    try {
        Zend_Loader::loadClass($className);
        self::$_instance->_view = new $className();
        self::$_instance->_templateEngine = strtolower($name);
        if (!isset($suffix)) $suffix = self::$_instance->_view->getSuffix();
        $viewManager = Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
        $viewManager->setView(self::$_instance->_view)->setViewSuffix($suffix);
    } catch (Exception $e) {
        Zend_Loader::loadClass('Fast_Exception_View');
        throw new Fast_Exception_View('Invalid View Engine: ' . $className);
    }
}
```

Toujours une exception typée pour mieux gérer les problèmes, et au passage un petit membre privé qui conserve le nom du moteur, ça mange pas de pain.

Et dans la méthode run() :

```
if ($parameters&&$config) {
    $engine = $parameters->fast->get('templateEngine', null);
    $controller = self::getInstance();
    Fast_Controller_Front::setViewEngine($engine);
}
```

C'est fini. Ma **Fast\_View\_Phtml** remplace **Zend\_View** et utilise **main.phtml**. Les fichiers touchés sont donc **parameters.ini** pour le paramètre 'templateEngine', **Fast\_Controller\_Front** pour charger la vue, **Fast\_Exception\_View**, et **Fast\_View\_Phtml**.

## VI - D'autres moteurs

Maintenant, tout est en place.

Pour ajouter ETS par exemple, je crée le script **Fast\_View\_Ets** que je place dans le dossier **library/Fast/View/**. Il me faudra aussi garder quelque part le moteur ETS lui-même. Je les places tous dans **library/TemplatesEngines/**.

En gros, la vue **ETS** va préparer le moteur et faire un appel à **printt** comme **Zend\_View** fait un *include* pour lancer le rendu. Cette classe aura la charge de gérer tout ce qui relève du paramétrage du moteur. S'il y a besoin de paramètre de configuration, elle peut obtenir les valeurs avec **Fast\_Registry**.

ETS a besoin d'un arbre de valeurs. On ajoute donc à la vue un membre pour les stocker : **\_ets**. On va utiliser les *setters* et *getters* de PHP pour affecter les valeurs tout comme le fait **Zend\_View**. On va aussi fournir la méthode **assign** qui a ses adeptes et qui existe aussi dans **Zend\_View**. Et c'est dans la méthode **run** qu'on va faire le gros du travail :

```
/**
 *
 * @param string The view script to execute.
 */
protected function _run()
{
    // récupère le chemin complet du template demandé
    $name = func_get_arg(0);
    // le template principal est considéré à la racine des templates
    // de l'application (ou du module).
    if (!isset($this->_ets->_mainTemplate))
        $this->setMainTemplate(str_replace(chr(92), '/', dirname(dirname($name)) . '/main.html'));
    // initialiser ets
    require_once("TemplatesEngines/Ets/Ets.php");
    // on indique à l'arbre de données d'Ets quel est le template à inclure
    $this->_ets->_content = str_replace(chr(92), '/', $name);
    $this->_ets->_templatesDir = dirname(dirname($name)) . '/';
    // rendu de la page
    printt($this->_ets, $this->_ets->_mainTemplate);
}
```

Elle ressemble à celle de Phtml, qui n'est qu'une copie de celle de Zend légèrement améliorée. La différence : on charge le moteur (require) et on l'invoque (printt).

Vous trouverez dans le fichier joint Phtml (script phtml avec *main page*) ETS, Smarty et Phptal. Ajouter un moteur est devenu quelque chose d'abordable et je vous laisse le loisir d'ajouter le vôtre.

Une petite note à propos de Phptal : TAL est le système de template de Zope. Son moteur a été porté dans plusieurs langages. Il n'est pas très gros (en ko) mais il est très structuré au niveau de son code. Mais surtout, il utilise des fichiers conformes XHTML, la structure XML du document n'est pas altérée. Tous les éléments de Tal sont dans un namespace. Du coup, tous les templates TAL peuvent être édités par un designer Web avec l'outil de son choix. Il est même possible d'embarquer dans le modèle des données d'exemples. Imaginez que vous demandiez à votre designer en chef de vous faire vos écrans et qu'il doit présenter un tableau de données : s'il l'édite sans aucune valeur dedans, il ne pourra se faire une idée du rendu final. Il va donc naturellement mettre des lignes dans sa table. Avec TAL, ces lignes ne sont pas gênantes, il sait en tenir compte pour afficher les valeurs réelles de l'application alors qu'il y a des lignes d'exemples dans la table. Bref, il est très souple, propre et bien structuré. Je vous conseille d'y jeter un #il, même si vous n'envisagez pas l'usage d'un tel système.

## VII - Zend Layout

Pendant que je faisais cette intégration, **Zend Layout** est apparu. Je n'ai pas encore eu le temps de me pencher dessus, mais à première vue cela me paraît intéressant. Je ne peux en dire plus pour le moment.

## VIII - Helper

Une autre approche est d'utiliser les helpers. Je n'ai pas eu le temps de tester la chose. Une chose est sûre : si je le fais, je garderai en vue que l'utilisation d'un moteur ne doit en aucun cas nécessiter un changement quelconque dans mes contrôleurs. Ce n'est pas parce qu'on change de présentation qu'on doit changer la logique de l'application. C'est le principe de fondement de MVC et je ne veux pas le remettre en question.

## IX - Pour tester

Afin de vous permettre de tester, je vous ai fait une [Source](#) copie de la chose. Pour la rendre fonctionnelle, il faut placer la librairie Zend dans le **library**.

Vous pouvez alors jouer avec les paramètres **debug** et **template** dans **parameters.ini**.

A+JYT

