

Zend Framework - Gérer la configuration d'une application

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


Date de publication : 22 octobre 2007

Dernière mise à jour :

Mon but est de mettre en place une structure générique qui me permette de gérer les configurations des diverses applications que je suis amené à écrire. Le contexte dans lequel je me trouve fait que mes applications ont beaucoup de points communs (elles sont toutes destinées à la même entreprise). Je sais très bien que je ne pourrai mettre en place un système de configuration universel mais il doit être suffisamment souple pour accepter la flexibilité nécessaire à la réutilisation.

- I - Configuration et paramétrage
- II - Le chargeur de configuration
- III - Organiser la base de registre
- IV - Une première approche de Fast_Config
- V - Fast_Config
- VI - Les fichiers de config

I - Configuration et paramétrage

Dans  [l'article précédent](#), je débattais de la différence entre le *paramétrage* et la *configuration*. Je vais tenter de mettre en #uvre ici le modèle que j'avais alors proposé.

J'ai donc trois fichiers :

- `environment.ini` ;
- `parameters.ini` ;
- Le fichier de configuration désigné par le fichier environnement.

Si l'on se reporte au code de l'article précédent, dans le Bootstrap le chargement de la config se fait par :

```
<?php  
Fast_Config::load();
```

Ou :

```
<?php  
Fast_Config::load("mon fichier d'environnement");
```

II - Le chargeur de configuration

Au vu de ce code, le chargeur de configuration est donc une classe possédant une méthode statique.

En regardant de plus près, l'application ne doit avoir qu'une seule configuration, au sens opérationnel. C'est-à-dire qu'une instance de l'application tournant à un moment donné ne peut avoir deux configurations différentes. La configuration doit donc au sein de l'application être unique. Une solution est de faire un **singleton**. Mais ici, nous n'avons pas vraiment d'intérêt à créer un objet (même unique) pour charger la configuration.

Charger une configuration est un besoin purement fonctionnel. Notre chargeur doit prendre en compte plusieurs fichiers mais il ne fait qu'un seul chargement. Je dirais même que c'est typiquement une simple fonction.

Que faire alors de la configuration chargée ? Zend Framework nous propose une solution : **Zend_Registry**. La base de registre du framework sert à garder toujours accessibles un ensemble de valeurs structurées. Il n'y a pas de raisons de se priver.

Chargeur de configuration et lecteur de fichier de configuration.

ZF nous fournit la classe **Zend_Config** et quelques dérivées. La classe **Zend_Config** ne ferait-elle pas l'affaire pour notre chargeur ?

Pour répondre à cette question, il faut se pencher sur son fonctionnement. **Zend_Config** est une classe qui permet de lire des fichiers de configuration. Mais elle ne détient pas le mécanisme à trois fichiers que j'ai décrit. Il est parfaitement envisageable de mettre ce mécanisme dans le BootStrap et d'utiliser **Zend_Config** pour cela.

Cette approche simple a l'inconvénient d'alourdir le BootStrap et si on n'y prend pas garde, il va se retrouver follement encombré par tout un tas de choses qu'il est facile de faire ainsi, jusqu'à le rendre illisible.

Fast_Config doit donc porter le mécanisme de chargement de la configuration. De cette manière, s'il évolue, seule cette classe devra changer. Une autre question se pose alors : **Fast_Config** doit-elle dériver de **Zend_Config** ? Nous venons de voir que leur rôle est fondamentalement différent. Une dérivation n'aurait pas de sens. J'aurais peut-être dû choisir un autre nom. Comme **Fast_Config_Loader** par exemple. **Fast_Config** m'est apparu naturel.

III - Organiser la base de registre

On peut se poser la question de l'organisation de la base de registre. Cette base étant structurée, quelle structure adopter ? Je pense que, pour le commun des mortels, retrouver dans une structure la même organisation que celle qu'on a choisi dans les fichiers de configuration, est une facilité appréciable même si d'autres regroupements logiques sont envisageables.

Ma base de registre contiendra trois parties : **environnement**, **parameters** et **configuration**.

Mon algorithme de chargement sera :

- Lecture du fichier environnement ;
- Écriture de l'environnement dans la base ;
- Lecture du fichier **parameters** ;
- Écriture des paramètres dans la base ;
- Lecture du fichier de configuration ;
- Écriture de la configuration dans la base.

Je serais bien tenté par une base de registre qui m'offre les services suivants :

```
<?php
Zend_Loader::loadClass('Zend_Registry');
class Fast_Registry extends Zend_Registry
{
    public static function getEnvironment();
    public static function setEnvironment($env);
    public static function getConfiguration();
    public static function setConfiguration($config);
    public static function getParameters();
    public static function setParameters($config);
}
```

Ainsi, mon chargeur pourrait utiliser les méthodes **set** et l'application les méthodes **get**. De plus **Fast_Registry** dérivant de **Zend_Registry**, j'ai à disposition toutes les fonctionnalités d'une base de registre.

IV - Une première approche de Fast_Config

```
1. <?php
2. Zend_Loader::loadClass('Fast_Registry');
3. class Fast_Config
4. {
5.     /**
6.      * @function load() charge toute la config en fonction du fichier environnement.
7.      * @param $environnement String|null Nom du fichier d'environnement
8.      * @return Fast_Config le contenu du fichier désigné par la clef environnement.
9.      */
10.    public static function load($environnement = null)
11.    {
12.        // initialise une base de registre
13.        Zend_Registry::setClassName('Fast_Registry');
14.        if ($environnement == null) {
15.            $environnement = './application/config/environment.ini';
16.        }
17.        Fast_Registry::setConfigPath(str_replace(chr(92), '/', dirname($environnement).''));
18.        // recherche un fichier de config déterminant l'environnement de travail.
19.        $config = Fast_Registry::setEnvironment(lireLeFichier($environnement));
```

La ligne 13 montre comment utiliser sa propre classe comme classe de registre dans ZF.

Une petite remarque au passage. `str_replace(chr(92), '/', dirname($environnement).'/')` : cet appel permet de remplacer tous les `\` par des `/`, ce qui n'est pas bien grave sauf dans quelques cas. Ainsi `"c:\real\path"` ne sera pas toujours bien compris simplement parce que le `\r` est un caractère qui sera interprété par PHP. Il existe bien des solutions pour se sortir de ce problème. J'utilise pour ma part cette expression et je travaille ainsi toujours avec des chemins à la UNIX.

Revenons à notre code. On le voit, une petite méthode pour lire le fichier serait bienvenue.

J'ai dit que j'allais utiliser `Zend_Config` pour lire le fichier. Un de ses avantages est de pouvoir lire différents types de fichiers de configuration : `.ini` `.xml` etc.

Ma petite procédure de lecture serait bien pratique si en plus elle prenait divers types de fichier.

Comme pour la base de registre, faisons comme si et écrivons notre méthode.

La suite à donner est de regarder dans le fichier environnement si nous trouvons bien les informations sur la configuration. Si ce n'est le cas, inutile de continuer, nous avons là un crash de l'application.

```
<?php
Zend_Loader::loadClass('Fast_Registry');
class Fast_Config
{
    /**
     * @function load() charge toute la config en fonction du fichier environnement.
     * @param $environnement String|null Nom du fichier d'environnement
     * @return Fast_Config le contenu du fichier désigné par la clef environnement.
     */
    public static function load($environnement = null)
    {
        // initialise une base de registre
        Zend_Registry::setClassName('Fast_Registry');
        if ($environnement == null) {
            $environnement = './application/config/environment.ini';
        }
    }
}
```

```
Fast_Registry::setConfigPath(str_replace(chr(92), '/', dirname($environment).'/'));
// recherche un fichier de config déterminant l'environnement de travail.
$config = Fast_Config::getFile('setEnvironment', $environment);
if(!isset($config->main->environment)) {
    Zend_Loader::loadClass('Fast_Exception_Config');
    throw new Fast_Exception_Config('Environment non défini. ');
} else {
    $environmentFile = Fast_Registry::getConfigPath().$config->main->environment;
}
```

Inutile de chercher une échappatoire s'il n'y a pas de config, on plante tout. Une exception est parfaite pour cela.

À l'appelant de se charger de trapper l'exception et de traiter le problème. Le chargeur lui ne peut rien.

Là encore, vous pouvez constater que j'utilise une **Fast_Exception_Config**, mais je pourrais très bien utiliser une **Zend_Exception**. Cependant, utiliser des exceptions typées permet de les traiter plus facilement. Je vous conseille de toujours typer les exceptions. Pour cela, il suffit de fournir une classe qui dérive de **Zend_Exception**.

Voici non pas la mais les miennes :

```
class Fast_Exception extends Exception{}
class Fast_Exception_Config extends Exception{}
```

Un gestionnaire d'exception peut ainsi trapper les exceptions en général, trapper les exceptions fast, trapper les exceptions de configuration fast et apporter un traitement approprié à chaque cas.

V - Fast_Config

À ce stade, j'ai le nom du fichier de configuration qui m'a été fourni par la clef **environnement** dans la section **main** du fichier **environnement.ini**.

Je serais bien tenté de faire pareil avec les paramètres, laissant ainsi un peu de liberté à l'application :

```
<?php
Zend_Loader::loadClass('Fast_Registry');
class Fast_Config
{
    /**
     * @function load() charge toute la config en fonction du fichier environnement.
     * @param $environnement String|null Nom du fichier d'environnement
     * @return Fast_Config le contenu du fichier désigné par la clef environnement.
     */
    public static function load($environnement = null)
    {
        // initialise une base de registre
        Zend_Registry::setClassName('Fast_Registry');

        if ($environnement == null) {
            $environnement = './application/config/environnement.ini';
        }

        Fast_Registry::setConfigPath(str_replace(chr(92), '/', dirname($environnement).'/'));

        // recherche un fichier de config déterminant l'environnement de travail.
        $config = Fast_Config::getFile('setEnvironment', $environnement);

        if(!isset($config->main->environnement)) {
            Zend_Loader::loadClass('Fast_Exception_Config');
            throw new Fast_Exception_Config('Environment non défini.');
```

Me reste à définir la méthode **getFile** qui va déterminer le type de fichier, lire le contenu et le mettre dans la base de registre avec la méthode spécifiée.

Vous trouverez le tout dans le [Source fichier joint](#).

Au passage, il serait intéressant d'activer immédiatement certains éléments de la configuration.

Pour en bénéficier pendant le Bootstrap :

```
<?php
// affichage des erreurs
```

Pour en bénéficier pendant le Bootstrap :

```
$display_errors = $config->debug->get('display_errors', false);
if (ini_get('display_errors') != $display_errors) {
    // pas d'ini_set inutile, c'est gourmand !
    ini_set('display_errors', $display_errors);
}
// niveau d'erreur (ne pas les afficher n'empêche pas de les loguer)
// avec un niveau raisonnable d'alarme par défaut en prod
// le niveau peut être indiqué avec des opérateurs (ex. E_ALL ^ E_STRICT)
$error_reporting = $config->debug->get('error_reporting', E_WARNING);
if (!is_numeric($error_reporting)) {
    eval("error_reporting($error_reporting);");
} else {
    error_reporting($error_reporting);
}
```

Placé juste avant le **return \$config**; cela permet d'avoir les messages d'erreur dès le démarrage lors du développement, et de les retirer en production juste en modifiant le fichier de configuration.

VI - Les fichiers de config

Environnement.ini

```
; Définit le fichier de configuration générale à charger au bootstrap.  
; Ce fichier varie généralement selon que l'on est en développement, en recette ou en production.  
; Exemple :  
; [main]  
; environment = dev.ini  
; parameters = parameters.ini  
  
[main]  
environment = dev.ini  
parameters = parameters.ini
```

Parameters.ini

```
[fast]  
; debug true ou false charge la classe Fast_Debug absent pas de classe  
; auth active le la protection  
debug = true  
db = Pdo_Mysql  
auth = true  
audiance = false  
menu = true  
version = G0-RC1  
  
[login_messages]  
login_need = l'identifiant est obligatoire  
pass_need = le mot de passe est obligatoire  
unknow = Identifiant ou mot de passe inconnu.  
unknow_user = Veuillez vous identifier avant de poursuivre.
```

Dev.ini

```
[debug]  
error_reporting = E_ALL ^ E_STRICT  
display_errors = true  
  
[app]  
baseUrl = /myApp/  
  
[Pdo_Mysql]  
host = 127.0.0.1  
port = 3307  
username = sekaijin  
password =  
dbname = test
```



Vous trouverez dans les fichiers joints des appels à d'autres classes. Je les ai ajoutées au fil du temps et je reviendrai dessus plus tard.

A+JYT

