

Zend Framework - BootStrap

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

Date de publication : 21 octobre 2007

Dernière mise à jour :

Zend Framework a besoin pour démarrer de quelques petites choses, comme par exemple le chemin où trouver les contrôleurs ou encore les vues. Et bien d'autres encore.

- I - Démarrage à froid
- II - En passant par une config
- III - Organiser la config.
- IV - Améliorer le FrontController

I - Démarrage à froid

On trouve sur le net nombre d'exemples de Bootstrap pour ZF qui expliquent comment ajouter telle ou telle option au démarrage.

Par exemple : celui de Simon Mundy dans son  [tutoriel sur la mise en #uvre des ACL](#).

```
<?php
// Initialisation de la configuration / environnement
$config = new Zend_Config(new Zend_Config_Ini('../application/config/config.ini', 'live'));

// Création du sitemap à partir du .ini en utilisant la structure de l'exemple
$sitemap = new Zend_Config(new Zend_Config_Ini('../application/config/sitemap.ini', 'live'));

// Création de l'objet de base de données et activation / désactivation du débogage
$db = Zend_Db::factory($config->db->connection, $config->db->asArray());
...etc...

// Création de l'objet Auth
$auth = Zend_Auth::getInstance();

// Création de l'objet Acl
$acl = new MyAcl($auth); // see

// Création du routeur et configuration (Ordre LIFO pour les routes)
$router = new Zend_Controller_RewriteRouter;
...add rules...

// Création des vues et enregistrement des objets
$view = new My_View;
...init view...

$front = Zend_Controller_Front::getInstance();
$front->throwExceptions(true);
$front->setRouter($router)
->setDispatcher(new Zend_Controller_ModuleDispatcher())
->registerPlugin(new My_Plugin_Auth($auth, $acl))
->registerPlugin(new My_Plugin_Agreement($auth))
->registerPlugin(new My_Plugin_View($view))
->setControllerDirectory(array('default' => realpath('../application/controllers/default'),
                                'admin' => realpath('../application/controllers/admin')))
->setParam('auth', $auth)
->setParam('view', $view)
->setParam('config', $config)
->setParam('sitemap', $sitemap)
->dispatch();
```

Si je m'en réfère à la documentation de ZF, le Bootstrap est on ne peut plus simple :

```
<?php
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('/chemin/vers/application/controllers');
```

Cette simplicité est séduisante, mais elle ne permet pas de personnaliser le démarrage.

Et devoir reproduire quelque chose d'aussi complexe que l'exemple ci-dessus est source d'erreurs.

II - En passer par une config

C'est pourquoi j'ai envisagé d'en passer par un fichier de configuration. Le Bootstrap se résumant alors à charger la config, charger le front controller et le démarrer.

```
<?php
/**
 * Bootstrap.
 *
 * Point d'entrée unique de l'application, ce script fixe l'environnement
 * de travail par rapport aux fichiers de configuration disponibles et
 * lance un Front Controller qui est le moteur principal de l'application
 * qui tourne sous le design pattern MVC tel qu'implémenté par
 * Zend Framework.
 */

// chemin des librairies
set_include_path(
    realpath('library')
    . PATH_SEPARATOR . realpath('application')
    . PATH_SEPARATOR . get_include_path()
);
// charge le loader général.
require 'Zend/Loader.php';
Zend_Loader::loadClass('Fast_Config');
Fast_Config::load(); //ou Fast_Config::load("mon fichier d'environnement");

// instancie un front contrôleur
// et lui indique le chemin des contrôleurs d'action
Zend_Loader::loadClass('Zend_Controller_Front');
Zend_Controller_Front::run('application/controllers');
```

Notre Bootstrap reste relativement simple et portable d'une application sur l'autre, les variations des options étant définies par la configuration.

Reste que Zend_Controller_Front ne sait pas utiliser mon Fast_Config. De plus, un fichier de configuration peut facilement devenir aussi compliqué que d'écrire un Bootstrap.

III - Organiser la config.

Dans mon travail, mes développements se font de la façon suivante : le projet est géré avec CVS ou SVN, chaque développeur a une copie de l'application, et l'exécute sur un environnement qui lui est propre. L'application passe ensuite sur une plateforme de développement commune qui est identique à la plateforme cible et qui permet de tester les relations avec les autres applications lorsque cela est nécessaire. Ensuite, l'application est déployée sur une plateforme de recette. Celle-ci est destinée à la maîtrise d'ouvrage qui va valider l'application. Très souvent, l'étape suivante est une mise en pre production dont le but est de tester l'application en situation. Et, au final, elle est enfin placée sur son environnement de production. Il va sans dire qu'immanquablement tous ces environnements ne sont pas parfaitement identiques. La configuration de l'application va donc en dépendre pour partie.

À l'opposé, il existe toujours dans une application des paramètres qui ne sont pas liés à l'environnement. Pour ne pas avoir à systématiquement rééditer un gros fichier de configuration, j'ai pris l'habitude d'en faire plusieurs. En premier lieu, séparer ce qui est du paramétrage de l'application de ce qui est de la configuration due à l'environnement.

J'ai donc un fichier **parameters** qui définit les paramètres de l'application, et un fichier de configuration. Ce dernier existe en autant d'exemplaires que de plateformes (ou presque). Une solution pour sélectionner le bon est de le nommer **config** et de renommer les autres qui pourraient éventuellement traîner par là. L'inconvénient de cette approche est que lorsqu'on voit dans le dossier **config** le fichier **config** on ne sait pas à quel environnement il se rapporte. J'ai donc choisi de les nommer en fonction de l'environnement : **dev**, **recette**, **preprod**, **prod**, etc. Reste donc à indiquer au BootStrap lequel choisir. J'ai pour cela défini un fichier qui ne fait que ça : `environnement.ini`

Si vous regardez la hiérarchie que j'ai choisie dans l'article précédent, il faut déterminer où placer ces fichiers. Pour moi, il s'agit de la configuration de l'application. J'ai donc ajouté un dossier **config** dans le **dossier application**. J'y place les fichiers **environnement.ini**, **parameters.ini** et **dev.ini** etc. Le dossier **application** étant déjà protégé, la configuration n'est normalement pas accessible pour le client.

Comment déterminer ce que l'on place dans **parameters** et ce qui va dans **dev** ou autre ? Il suffit pour cela de se poser la question « l'attribut de configuration doit-il changer si je change de plateforme ? » Par exemple, l'option « utiliser les ACL » ne dépend pas de l'endroit où l'on place l'application. C'est un paramètre. En revanche, « database host » dépend de la plateforme : il va donc dans la config.

Ainsi, le passage d'une plateforme à l'autre se résume à créer un fichier **config** propre à la plateforme et à changer la valeur d'**environnement.ini**.

Cette façon de faire n'est peut-être pas la meilleure, mais elle a fait ses preuves.

Apprendre au `Front_Controller` à utiliser la config.

IV - Améliorer le FrontController

La seule approche que permet par défaut ZF pour configurer le Front Controller est d'en passer par le Bootstrap. Mais ZF est une belle architecture à objets, qui autorise l'héritage et le polymorphisme. Je vais me servir de cette caractéristique pour construire un Front_Controller paramétrable par la configuration.

Fast_Controller_Front va donc dériver de Zend_Controller_Front et fournir les mêmes fonctionnalités mais, au passage, il va utiliser la configuration pour ajouter seul, les options demandées.

Pour cela, il suffit de surcharger la méthode statique **run** :

```
<?php
require_once 'Zend/Loader.php';
Zend_Loader::loadClass('Zend_Controller_Front');

/**
 * Contrôleur principal permettant l'utilisation de la configuration.
 *
 * @author Patrick Dubois
 * @author Jean-Yves Terrien
 *
 * @uses Zend_Controller_Front
 * @package Fast_Controller
 */
class Fast_Controller_Front extends Zend_Controller_Front
{
/**
 * Convenience feature, calls setControllerDirectory()->setRouter()->dispatch()
 *
 * In PHP 5.1.x, a call to a static method never populates $this -- so run()
 * may actually be called after setting up your front controller.
 *
 * @param string|array $controllerDirectory Path to Zend_Controller_Action
 * controller classes or array of such paths
 * @return void
 * @throws Zend_Controller_Exception if called from an object instance
 */
public static function run($controllerDirectory)
{
    $config = Fast_Registry::getConfiguration();
    $parameters = Fast_Registry::getParameters();
    if ($parameters&&$config) {
        #
    } else {
        Zend_Controller_Front::run('application/controllers');
    }
}
}
```

Ainsi dans notre Bootstrap, nous pouvons garder un code simplifié, il suffit d'utiliser Fast_Controller_Front à la place de Zend_Controller_Front :

```
<?php
/**
 * Bootstrap.
 *
 * Point d'entrée unique de l'application, ce script fixe l'environnement
 * de travail par rapport aux fichiers de configuration disponibles et
 * lance un Front Controller qui est le moteur principal de l'application
```

```
* qui tourne sous le design pattern MVC tel qu'implémenté par
* Zend Framework.
*/

// chemin des librairies
set_include_path(
realpath('library')
. PATH_SEPARATOR . realpath('application')
. PATH_SEPARATOR . get_include_path()
);
// charge le loader général.
require 'Zend/Loader.php';
Zend_Loader::loadClass('Fast_Config');
Fast_Config::load(); //ou Fast_Config::load("mon fichier d'environnement");

// instancie un front contrôleur
// et lui indique le chemin des contrôleurs d'action
Zend_Loader::loadClass('Fast_Controller_Front');
Fast_Controller_Front::run('application/controllers');
```

Notez que j'ai pris la précaution de lancer un Zend_Controller_Front si je n'ai pas de config, permettant ainsi de garder le fonctionnement par défaut de ZF.

Je n'ai abordé ici que la structure générale. La mise en #uvre fera l'objet d'une publication future.

