

# Atelier Zend Framework : Gestions des exceptions intégrées dans MVC : le plugin ErrorHandler

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

Date de publication : 29/05/2007

Dernière mise à jour : 27/01/2008

La gestion des erreurs dans une application web basée sur MVC est un passage obligatoire. Zend Framework propose un plugin agissant dans le contrôleur frontal : le 'ErrorHandler'.

Voyons ensemble son fonctionnement, et son utilité.

- I - Utilisation du plugin
- II - Notions subtiles avancées

## I - Utilisation du plugin

Le modèle MVC de ZF regorge de designs patterns et parmi eux, on note de nombreux proxies, qui je le rappelle permettent de déléguer le traitement entier d'un processus à une autre classe.

Le contrôleur frontal de Zend Framework utilise le motif proxy via des plugins. Un plugin est une classe que l'on peut injecter dans le frontController, et qui va avoir un but précis et agir au sein même du processus de dispatching.

ZF est livré de base avec plusieurs plugins autoenregistrés : l'un d'entre eux est le 'ErrorHandler'

Le ErrorHandler a pour but de traiter les exceptions levées dans la boucle de dispatching.

Il analyse en postDispatch si la réponse contient des Exceptions. Si c'est le cas, alors il renvoie la requête vers le trio module/contrôleur/action d'erreur, qu'on lui aura spécifié.

Par défaut il s'agit du module 'default', errorController et errorAction, tout ceci est bien entendu modifiable.

Si une exception est levée pendant le processus de dispatching du contrôleur d'erreur, alors l'exception est interceptée, elle est passée au frontController qui se voit donner l'ordre de l'envoyer.

Il faut donc implémenter en dernier recours une méthode de gestion d'erreurs "graves", en sortie du contrôleur frontal

Aussi, si vous n'utilisez pas ce plugin, désactivez le par un `$frontController->setParam('noErrorHandler', true)`. De cette manière vous éviterez tout ce processus inutile, et sauvegarderez des ressources ;-)

On attaque le code ? Vous allez voir, c'est vraiment pas méchant !

```
<?php
// ... plein de config ici, je vous laisse faire ...

/* Voila par exemple la personnalisation du contrôleur d'erreur
$eh = new Zend_Controller_Plugin_ErrorHandler();
$eh->setErrorHandlerModule('monmodule')
    ->setErrorHandlerController('error')
    ->setErrorHandlerAction('erreurs');
$frontController->registerPlugin($plugin);

*/
// *****
// démarrage de la boucle de dispatching
// *****

try {
    $frontController->dispatch(); // dispatche !
} catch (Exception $exception) { // attrape toute exception
    exit($exception->getMessage());
}
```

Le contrôleur d'erreur lui, va gérer les exceptions au cas par cas :

```
<?php
class ErrorController extends Zend_Controller_Action
{
```

```

private $_exception;
private static $errorMessage;
private static $httpCode;

public function preDispatch()
{
    $this->_helper->viewRenderer->setNoRender(true); // ne rend aucune vue automatiquement
    $this->_exception = $this->_getParam('error_handler');
    $this->_response->clearBody(); // on vide le contenu de la réponse
    $this->_response->append('error',null); // on ajoute un segment 'error' dans la réponse

    switch ($this->_exception->type) {
        case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
        case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
            self::$httpCode = 404;
            self::$errorMessage = 'Page introuvable';
            break;
        case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_OTHER:
            switch (get_class($this->_exception->exception)) {
                case 'Zend_View_Exception' :
                    self::$httpCode = 500;
                    self::$errorMessage = 'Erreur de traitement d\'une vue';
                    break;
                case 'Zend_Db_Exception' :
                    self::$httpCode = 503;
                    self::$errorMessage = 'Erreur de traitement dans la base de données';
                    break;
                case 'Metier_Exception' :
                    self::$httpCode = 200;
                    self::$errorMessage = $this->_exception->exception->getMessage();
                    break;
                case 'Autre_Exception' :
                    self::$httpCode = 500;
                    self::$errorMessage = 'Exemple avec une "autre exception"';
                    break;
                default:
                    self::$httpCode = 500;
                    self::$errorMessage = 'Erreur inconnue';
                    break;
            }
            break;
    }
}

public function errorAction()
{
    $this->getResponse()->setHttpResponseCode(self::$httpCode);
    $this->_errorMessage .= sprintf("<p>%s</p>",self::$errorMessage);
}

public function postDispatch()
{
    $this->getResponse()->appendbody($this->_errorMessage,'error');
    $this->getResponse()->appendbody('<a href="javascript:history.back()">retour</a>', 'error');
    if (Zend_Registry::get('config')->debug == 'true') {
        $message = sprintf('<hr>DEBUG INFOS :<br /><strong>Exception de type <em>%s</em> <u>%s</u>
    envoyée dans %s à la ligne %d </strong> <p>Stack Trace : %s </p><hr>',
            get_class($this->_exception->exception),
            $this->_exception->exception->getMessage(),
            $this->_exception->exception->getFile(),
            $this->_exception->exception->getLine(),
            Zend_Debug::dump($this->_exception->exception,null,false)
        );
        $this->getResponse()->append('debug',$message);
    }
}
}

```

Attention ici il y a des choix personnels : je n'utilise pas de vue pour le contrôleur d'erreur.

Je vide la réponse de tout segment et j'y ajoute des segments personnalisés, si un **Layout** intervient, il comprendra ceci

`$this->_getParam('error_handler')` retourne un `ArrayObject`, qui contient au rang 'type', le type d'exception retournée, et au rang 'exception', la dernière exception attrapée par le plugin.

Le type peut être de 3 sortes, matérialisées par 3 constantes de la classe `Zend_Controller_Plugin_ErrorHandler`, dont la lecture coulera de source.

Selon le type d'exception, on va renvoyer un code HTTP approprié, avec un message approprié. Si le mode debug est activé ( enregistrer dans le registre auparavant ), on affiche des infos de débogage.

## II - Notions subtiles avancées

C'est simple, mais il y a des notions à subtiles à assimiler si on ne veut pas tomber dans certains pièges.

Déjà, la pile de plugin. J'en parle plus amplement [ici](#). Le plugin errorHandler est activé lors de l'entrée en boucle de dispatching, par le contrôleur frontal (sauf si on le désactive, ça va de soit). Il est enregistré dans la pile des plugins au rang 100.

Les plugins pouvant créer des effets de bord entre eux, il est important de noter que le ErrorHandler agira en dernier, sauf si vous enregistrez des plugins avec un rang supérieur à 100.

Par défaut, **registerPlugin()** enregistre en partant de 1, puis incrémente la pile de 1 à chaque fois; donc si vous ne vous occupez de rien lors de l'enregistrement de vos divers plugins (si vous en avez ;-), en théorie vous ne risquez rien.

Surtout rappelez vous un fait très important : si la boucle de dispatching venait à être dispatchée 2 fois avec agissement du plugin ErrorHandler, celui-ci organise une sortie d'urgence pour éviter une boucle infinie de dispatching.

Pensez à ce cas simple : une exception est levée, ErrorHandler appelle alors le contrôleur d'erreur et renvoie un jeton ( il redemande une boucle de dispatching complète ). Le dispatcheur essaye de dispatcher le contrôleur d'erreur, qui est introuvable pour une raison X. Le plugin ErrorHandler capture alors cette exception et demande le dispatching du contrôleur d'erreur, et tout recommence, à l'infini, puisque ce contrôleur d'erreur n'est pas dispatchable.

Le plugin ErrorHandler est donc capable de vérifier si il agit 2 fois de suite. Si c'est le cas, alors c'est que probablement une boucle infinie se fait sentir, il opte donc pour la "sortie d'urgence".

Il demande au contrôleur frontal de renvoyer les exceptions (**throwExceptions(true)**), puis lui envoie la dernière exception qu'il a capturée.

Il faut donc impérativement entourer son **\$front->dispatch()** d'un try catch, même si vous ne demandez pas un rendu d'exceptions (cas par défaut), car en cas de pépin, le plugin ErrorHandler lui, demande au contrôleur frontal d'envoyer ses exceptions.

Aussi, il est idiot de demander au contrôleur frontal d'envoyer les exceptions explicitement, car alors on court-circuite le plugin qui n'agit plus correctement.

Le plugin ErrorHandler est étudié pour gérer les exceptions dans le dispatching complet, mais lui agit dans la boucle (prédispatch - dispatch - postdispatch). C'est à dire qu'il interagit avec tous les plugins enregistrés. Si ceux-ci emettent une exception en dispatchLoopStartup, par exemple, alors l'exception est traitée par ErrorHandler.

Mais en revanche, une exception lancée dans le même plugin, en prédispatch, sera alors levée 2 fois, et la sortie d'urgence intervient alors.

