Atelier Zend Framework : MVC, les plugins et les aides d'action

par Matthew Weier O'Phinney (Phly, boy, phly - the weblog and site of Matthew Weier O'Phinney) (Blog) Julien Pauli (Tutoriels, article et conférences PHP et developpement web) (Blog)

Date de publication : 10/07/2008

Dernière mise à jour :

Le système MVC de Zend Framework propose une architecture souple, matérialisée entres autres par la possibilité de créer des plugins de contrôleur frontal, et des aides d'action.
Nous allons voir en quoi ces 2 entités sont utiles, leur fonctionnement, et leurs différences.
Cet atelier est une traduction des articles de Matthew Weier O'Phinney, ils sont disponibles **ici** et **là**.

I - Les aides d'action (ActionHelpers)....3
I-A - Les bases....3
I-B - La méthode direct()....3
I-C - La gestion des évènements....4
I-D - Enregistrer des aides dans le gestionnaire....4
I-E - Récupérer statiquement des aides du gestionnaire....5
I-F - Créer votre propre aide....5
II - Les plugins de contrôleur frontal (FrontController Plugins)....8
II-A - Définition....8
II-B - Enregistrer et récupérer un plugin avec le contrôleur frontal....8
II-C - Les plugins internes pré-enregistrés....9
II-D - Exemples d'utilisation de plugins....9
II-D-1 - Plugin d'initialisation de MVC....9
II-D-2 - Plugin de gestion de cache....10
II-E - Utiliser plusieurs actions (forward)....12
III - Conclusion....14

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Les aides d'action (ActionHelpers)

Les aides d'action sont souvent considérées comme des "fonctions de pro", et sont ainsi souvent boudées. Cependant, leur but est d'étendre les fonctionnalités du contrôleur d'action (**Zend_Controller_Action**), sans pour autant devoir dériver cette classe importante, par héritage. Nous allons voir comment les aides d'action fonctionnent, comment les créer et les utiliser à notre avantage.

I-A - Les bases

On peut voir dans certains tutoriels, des rajouts de fonctions aux actions, par une méthode d'héritage :

Redéfinition de la classe d'action principale

```
/**
 * Vos contrôleurs d'action étendraient My_Controller_Action
 */
abstract class My_Controller_Action extends Zend_Controller_Action
{
    // Ici, nos méthodes pratiques personnalisées ...
}
```

Ceci est possible, mais est très souvent une solution bancale, dans la mesure où il est très rare que toutes vos actions aient besoin de toutes les fonctionnalités ajoutées par héritage.
Souvent, seule une partie des méthodes ajoutées va servir pour certains contrôleurs, alors qu'une autre partie servira pour d'autres contrôleurs ; cependant, tous les contrôleurs héritent de toutes les méthodes...
La solution : utiliser des aides d'action. Leur rôle est de permettre un ajout de fonctionnalités précises, pour des contrôleurs précis, parmi le lot. Ainsi, les aides d'actions ne sont chargées qu'à leur utilisation, et non en permanence.

Les aides d'actions sont encapsulées dans un gestionnaire d'aide, matérialisé par la classe **Zend_Controller_Action_HelperBroker**. Cette classe sert de registre permanent, et de dico **fabrique**, afin de charger les aides à la demande.
Par défaut, l'attribut *$_helper* de la classe **Zend_Controller_Action** contient une instance du gestionnaire d'aides. Lorsque nous utilisons une aide, le gestionnaire lui passe l'instance du contrôleur d'action actuel, et l'aide peut donc intéragir avec celui-ci. Ainsi, l'aide peut appeler des méthodes publiques du contrôleur d'action (ou accéder à ses attributs publics), et inversement.

En règle générale, nous appelons notre aide en utilisant la dernière partie du nom de la classe la représentant. Par exemple, une aide dont la classe est '**Foo_Helper_Bar**', s'appelle par 'bar'. Pour la récupérer, nous passons soit par une propriété de la classe d'action, soit via sa méthode ***getHelper()*** :

récupération d'une aide d'action

```
$bar = $this->_helper->bar;
$bar = $this->_helper->getHelper('bar');
```

C'est bien, mais il est possible de faire plus que ça.

I-B - La méthode direct()

Les aides d'action utilisent le dico **design pattern** Stratégie (Strategy). Si nous définissons une méthode ***direct()*** dans notre classe d'aide, alors on peut appeller celle-ci comme une méthode de la classe d'action principale.
Par exemple, l'aide d'action "url", qui retourne une URL en se basant sur ses paramètres, s'appelle comme ceci :

pattern strategy et méthode direct()

```
$url = $this->_helper->url('bar', 'foo'); // "/foo/bar"
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Utiliser la méthode ***direct()*** dans l'aide permet donc un appel 'virtuel' direct à celle-ci depuis la classe d'action principale : très pratique.

I-C - La gestion des évènements

Comme si cela n'était pas suffisant, les aides d'action possèdent des méthodes dites évènementielles, qui permettent l'automatisation de certaines tâches. Voici ces 3 évènements :

- ***init()***: appelée lorsque le contrôleur d'action est initialisé (seulement si une instance de l'aide existe dans le gestionnaire)
- ***preDispatch()***: appelée après la routine de preDispatch() des plugins, mais avant la routine preDispatch() de l'action actuelle, et seulement si une instance de l'aide existe dans le gestionnaire.
- ***postDispatch()***: appelée après la routine de postDispatch() des plugins, mais avant la routine postDispatch() de l'action actuelle, et seulement si une instance de l'aide existe dans le gestionnaire.

"Seulement si une instance de l'aide existe dans le gestionnaire" : les appels se font donc à la demande, sur les aides enregistrées dans le gestionnaire.
Prenons à titre d'exemple l'aide **ViewRenderer**, qui est activée par défaut dans Zend Framework, elle utilise les évènements suivants :

1. ***init()***: L'aide initialise un objet de vue, configure le script de vue à rendre, les chemins vers les aides et les filtres de vue, et enregistre l'objet vue comme propriété "view" dans l'action actuelle.
2. ***postDispatch()***: determine si une vue doit être rendue, et si c'est le cas, rend la bonne vue en fonction de l'action appelée, dans le segment de réponse approprié.

Un autre exemple : nous pourrions utiliser l'évènement ***preDispatch()*** dans une aide quelconque pour vérifier un attribut public du contrôleur d'action actuel afin de déterminer quelle action requiert une authentification, et ainsi rediriger vers un formulaire de login approprié le cas échéant. C'est mieux qu'un plugin (voir après), car l'aide agirait uniquement sur l'action actuelle.

I-D - Enregistrer des aides dans le gestionnaire

Si vous voulez utiliser les évènements sur vos aides d'action, il faut pouvoir les enregistrer dans le gestionnaire d'aides, assez tôt, typiquement en chargement (bootstrap), ou dans un plugin (tôt). Pour cela, agissez comme suit :

enregistrement d'une aide d'action dans le gestionnaire

```
Zend_Controller_Action_HelperBroker::addHelper( new Foo_Helper_Bar() );
```

Aussi, vous pouvez spécifier au gestionnaire un préfixe de classe, il saura alors trouver les aides de lui-même :

```
// Par préfixe de classe:
Zend_Controller_Action_HelperBroker::addPrefix('Foo_Helper');

// Si les classes ne sont pas dans l'include_path, spécifiez un chemin en plus du préfixe :
Zend_Controller_Action_HelperBroker::addPath($path, 'Foo_Helper');
```

Ajouter un chemin ou un préfixe ne fait que dire au gestionnaire où se trouvent les classes d'aide, il ne va pas les instancier.
Si vous avez instancié manuellement une aide avant le dispatching, vous devrez alors la passer au gestionnaire pour qu'elle soit enregistrée dans les actions futures, ou alors demander au gestionnaire de vous retourner une aide spécifique en l'instanciant, par la même occasion.

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I-E - Récupérer statiquement des aides du gestionnaire

Il arrivera quelques fois de vouloir récupérer une instance d'une aide d'action, afin de la configurer ou de l'analyser. Plutôt que de l'instancier nous-même, nous pouvons demander au gestionnaire d'aides de créer l'instance et de nous la retourner. ***getStaticHelper()*** est la méthode qu'il nous faut.
Un exemple : l'aide d'action ViewRenderer. Il est souvent utile de la récupérer pour la configurer, par exemple lui passer les chemins des aides de vue par défaut. Procédons ainsi comme ceci :

récupération d'une aide depuis le gestionnaire

```
$viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');

$viewRenderer->initView(); // assurons nous que l'objet de vue est bien initialisé
$viewRenderer->view->addHelperPath($path); // ajoutons un chemin vers les aides de vue.
```

Le gestionnaire d'aide va alors instancier l'objet de l'aide s'il n'existait pas, et le retourner dans le cas contraire. Il s'agit ainsi de la même instance.

I-F - Créer votre propre aide

Maintenant que nous en savons plus sur les aides d'action, passons à la pratique. Imaginons des contrôleurs qui utilisent un ou plusieurs formulaires. De même, un formulaire peut être utilisé dans plusieurs contrôleurs d'action. Nous allons créer une aide d'action qui va permettre de récupérer un formulaire par nom de classe.
Nous allons supposer que les formulaires sont stockés dans un dossier 'forms', juste sous le module en cours. Aussi, les noms de formulaires auront un espace de noms relatif au module en cours, sauf s'il s'agit du module par défaut, ajouté à l'espace de noms 'Form_'. Par exemple dans un module 'news', les formulaires auront l'espace de nom 'News_Form_'. La dernière partie de l'espace de nommage sera simplement le nom du formulaire tel que nous l'appellerons.
Nous utiliserons pour cela la méthode ***direct()***, l'aide d'action n'a qu'une chose à faire : charger le formulaire et le retourner, la méthode ***direct()*** est donc parfaite à cet effet.

notre aide d'action chargeant des formulaires

```
/**
 * Aide d'action de chargement de formulaires
 *
 * @uses Zend_Controller_Action_Helper_Abstract
 */
class My_Helper_FormLoader extends Zend_Controller_Action_Helper_Abstract
{
    /**
     * @var Zend_Loader_PluginLoader
     */
    public $pluginLoader;

    /**
     * Constructeur: initialisee le chargeur de classes d'aides ou plugins
     *
     * @return void
     */
    public function __construct()
    {
        $this->pluginLoader = new Zend_Loader_PluginLoader();
    }

    /**
     * Charge le formulaire avec les options passées
     *
     * @param  string $name
     * @param  array|Zend_Config $options
     * @return Zend_Form
     */
    public function loadForm($name, $options = null)
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

notre aide d'action chargeant des formulaires
```
    {
        $module  = $this->getRequest()->getModuleName();
        $front   = $this->getFrontController();
        $default = $front->getDispatcher()
                         ->getDefaultModule();
        if (empty($module)) {
            $module = $default;
        }
        $moduleDirectory = $front->getControllerDirectory($module);
        $formsDirectory  = dirname($moduleDirectory) . '/forms';

        $prefix = (('default' == $module) ? '' : ucfirst($module) . '_')
                . 'Form_';
        $this->pluginLoader->addPrefixPath($prefix, $formsDirectory);

        $name      = ucfirst((string) $name);
        $formClass = $this->pluginLoader->load($name);
        return new $formClass($options);
    }

    /**
     * Strategy pattern: appelle l'aide comme méthode du contrôleur d'action
     *
     * @param  string $name
     * @param  array|Zend_Config $options
     * @return Zend_Form
     */
    public function direct($name, $options = null)
    {
        return $this->loadForm($name, $options);
    }
}
```

Ce code doit être placé dans un fichier appelé '*FormLoader.php*', situé dans le dossier 'My/Helper/', lui-même dans l'include_path.
Bien, comment l'utiliser maintenant ? Imaginons que nous sommes dans le module par défaut, et dans un contrôleur d'action **LoginController**. Nous voulons charger le formulaire 'login'. Nous le nommerons '*Form_Login*', et placerons sa classe dans 'forms/Login.php', dans le dossier de l'application :

```
application/
    controllers/
        LoginController.php
    forms/
        Login.php - Contient la classe 'Form_Login'
```

Dans notre code d'amorçage (bootstrap), nous nous assureons que le gestionnaire d'aides d'action trouvera l'aide en question :

```
Zend_Controller_Action_HelperBroker::addPrefix('My_Helper');
```

Puis enfin, dans notre contrôleur **LoginController**, nous pouvons appeler le formulaire grâce à l'aide d'action :

```
$loginForm = $this->_helper->formLoader('login');
```

Beaucoup de travail pour si peu ? Du tout ! Aussi longtemps nous suivrons la règle de nommage, nous pourrons appeller l'aide pour autant de formulaires que nécessaire.
Si dans un contrôleur **UserController**, nous voulons appeler un formulaire d'enregistrement, nous procéderons ainsi :

```
$regForm = $this->_helper->formLoader('registration');
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

De plus, une fois que le préfixe est crée et qu'il est enregistré dans le gestionnaire (par exemple 'My_Helper'), nous pouvons ajouter des aides dans le même dossier, elles suivront la même règle de nommage, et le gestionnaire les trouvera et les chargera.

Il est clair que les aides d'action nous aident à suivre le principe 'DRY' (Don't Repeat Yourself : n'écrivez pas deux fois une même implémentation). Tout ce que nous pensons utiliser encore et encore, nous le déléguons à une aide d'action, qui sera disponible pour tous les contrôleurs d'action dispatchés.
Après quelque temps, nous aurons un ensemble de fonctionnalités sous forme de classes, que nous pourrons alors migrer de projet en projet, sans devoir dériver (hériter) la classe d'action principale.
Fléxibilité et extensibilité :-).

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II - Les plugins de contrôleur frontal (FrontController Plugins)

Les plugins de contrôleurs frontal (les plugins) servent à étendre les capacités de l'application dans son ensemble. Comme les aides d'action, ils évitent ainsi de devoir dériver des classes importantes, comme le contrôleur frontal **Zend_Controller_Front**.

II-A - Définition

Dans le Zend Framework, les plugins sont conçus pour s'enregistrer et pour écouter des évènements particuliers. Ces évènements sont remarquables et importants : le routage de la requête, la boucle de dispatching, le dispatching d'une action... Les voici :

1. routeStartup(): Avant le routage de la requête ;
2. routeShutdown(): Après le routage de la requête ;
3. dispatchLoopStartup(): Avant d'entrer dans la boucle de dispatching ;
4. preDispatch(): Avant de dispatcher un contrôleur d'action ;
5. postDispatch(): Après avoir dispatché un contrôleur d'action ;
6. dispatchLoopShutdown(): Après la fin de la boucle de dispatching.

On peut se poser une question lorsqu'on lit cette liste : "Pourquoi y a-t-il distinction entre la fin du routage et l'entrée dans la boucle de dispatching, alors que rien ne se passe entre les deux ?".
En fait, c'est pour une question conceptuelle, sémantique. Nous pourrions vouloir changer le routage juste après celui-ci, ou altérer le dispatcheur juste avant son rôle (sa boucle). La sémantique est différente, et avoir ainsi deux évènements distincts permet d'y voir plus clair.
Dans le même genre : "Pourquoi existe-t-il des méthodes évènementielles ***dispatchLoopStartup/Shutdown()*** et ***pre/postDispatch()*** ?". Tout simplement car nous pouvons mettre en avant la boucle de dispatching.
Le routeur n'est appelé qu'une seule fois, avant la boucle, alors qu'à l'interieur de la boucle, plusieurs actions peuvent s'enchaîner, d'où l'importante distinction.

Un plugin est une classe qui étend **Zend_Controller_Plugin_Abstract**. Cette classe abstraite définit des méthodes vides pour chacun des évènements. Un plugin devra donc redéfinir certaines (ou toutes) de ces méthodes, et le code utile y sera inséré.
Sauf pour ***dispatchLoopShutdown()***, toutes les méthodes du plugin prennent en paramètre une variable $request, de type **Zend_Controller_Request_Abstract** (la classe de requête de base dans le MVC de ZF).

```
public function preDispatch(Zend_Controller_Request_Abstract $request)
{
}
```

Il est donc possible de distinguer les plugins "qui agissent tôt", sur les évènements ***routeStartup()***, ***routeShutdown()***, et ***dispatchLoopStartup()***, soit avant l'entrée en boucle de dispatching, avant toute action.
Ces plugins auront donc un effet sur l'ensemble de l'application. Aussi, nous pouvons à l'inverse distinguer les plugins "qui agissent tard", sur les évènements ***postDispatch()*** et ***dispatchLoopShutdown()***, qui agissent donc après qu'une (ou des) action ait été traitée.

II-B - Enregistrer et récupérer un plugin avec le contrôleur frontal

Les classes de plugins doivent être instanciées, et passées au contrôleur frontal, ceci se fait grâce à la méthode ***Zend_Controller_Front::registerPlugin()*** :

enregistrement d'un plugin

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new FooPlugin());
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

On peut enregistrer un plugin quand on le souhaite, simplement ils n'agiront qu'après leur enregistrement, en général nous les enregistrerons en bootstrap (configuration du système MVC).
En option, nous pouvons passer un numéro pile aux plugins. Ceci permet de spécifier l'ordre dans lequel ils agiront, plus le chiffre est bas, plus tôt le plugin agira (avant ses frères).

```
$front->registerPlugin(new FooPlugin(), 1);   // agira tôt
$front->registerPlugin(new FooPlugin(), 100); // agira tard
```

Il peut quelquefois être nécessaire de pouvoir récupérer un plugin enregistré dans le contrôleur frontal, pour le configurer après son enregistrement, par exemple. La méthode pour ça est ***Zend_Controller_Front::getPlugin()*** :

```
$front     = Zend_Controller_Front::getInstance();
$fooPlugin = $front->getPlugin('FooPlugin');
```

II-C - Les plugins internes pré-enregistrés

Maintenant que nous savons manipuler les plugins, voyons un peu à quoi ils peuvent bien servir. Prenons ainsi comme exemple les plugins pré-existants dans le système MVC de ZF :
Zend_Layout :
Zend_Layout peut être utilisée, de manière optionnelle, avec les composants MVC. Lorsqu'il est utilisé, il ajoute un plugin (entre autres) dans le système MVC. Celui-ci écoute l'évènement ***postDispatch()*** et est enregistré dans la pile avec un numéro très élevé, de manière à ce qu'il intervienne après tous les autres plugins éventuels.
Le plugin Layout permet alors un pattern "Two Step View" : il capture le contenu de la réponse et le passe à l'objet Layout, de manière à ce que ce contenu puisse être réinjecté dans le script de vue layout.
Error Handler :
Ce plugin est enregistré et il écoute l'évènement ***postDispatch()***, lui aussi avec un numéro élevé de pile. Il analyse la réponse afin de vérifier si une exception lui a été enregistrée. Si c'est le cas, il réinjecte alors une autre requête dans le dispatcheur, une requête menant vers un contrôleur et une action d'erreur, permettant ainsi de traiter les exceptions comme des erreurs.

II-D - Exemples d'utilisation de plugins

Pour donner d'autres idées, nous pourrions penser à des plugins dont les rôles pourraient être :

1. Initialisation de l'application ;
2. Système de cache ;
3. Initialisation et personnalisation des routes ;
4. Authentication et gestion d'ACLs ;
5. Filtres de rendu final pour XHTML.

Considérons notre premier exemple : initalisation de l'application. D'habitude, nous initialisons l'application dans le bootstrap, souvent appelé "index.php". Cependant, ceci mène souvent à des fichiers gros, lourds et peu commodes. De même, il n'est alors pas possible de partager du code de ce fichier entre applications. Pourquoi ne pas déléguer une partie de la configuration à un plugin ? Particulièrement sur l'évènement ***routeStartup()*** :

II-D-1 - Plugin d'initialisation de MVC

```
/**
 * Plugin d'initialisation d'application
 *
 * @uses Zend_Controller_Plugin_Abstract
 */
class My_Plugin_Initialization extends Zend_Controller_Plugin_Abstract
{
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    /**
     * Constructeur
     *
     * @param  string $env Execution environment
     * @return void
     */
    public function __construct($env)
    {
        $this->setEnv($env);
    }

    /**
     * Evènement route startup
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @return void
     */
    public function routeStartup(Zend_Controller_Request_Abstract $request)
    {
        $this->loadConfig()
             ->initView()
             ->initDb()
             ->setRoutes()
             ->setPlugins()
             ->setActionHelpers()
             ->setControllerDirectory();
    }

    // ...
}
```

Le fichier bootstrap pourra ressembler à ceci :

```
require_once 'Zend/Loader.php';
Zend_Loader::registerAutoload();
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new My_Plugin_Initialization('production'));
$front->dispatch();
```

Les méthodes décrites dans le plugin sont assez explicites. Nous voyons qu'à présent la boostrap est plus aéré, et la gestion de la configuration sera partageable entre applications.
La maintenabilité en est ainsi améliorée.

II-D-2 - Plugin de gestion de cache

Souvent, les pages d'un site sont relativement statiques. Pourquoi ne pas créer un plugin basé sur **Zend_Cache**, qui va chercher dans le cache une page précédemment intérrogée, et la restaurer, plutôt que de la recalculer ? Voici les critères de notre exemple de cache :

1. Configuration du cache passée au constructeur ;
2. Seules les requêtes GET seront cachées ;
3. Les redirections ne doivent pas être cachées ;
4. N'importe quelle action doit pouvoir dire au cache qu'elle ne souhaite pas être cachée

Ce plugin va se baser sur deux évènements. D'abord, le routage doit être terminé et la boucle de dispatching pas encore entammée, pour vérifier si la requête peut être issue du cache, ou non.
Ensuite, nous devons cacher lorsque nous sommes certains que toutes les actions sont terminées.
Les deux évènements appropriés sont donc ***dispatchLoopStartup()***, et ***dispatchLoopShutdown()***.

```
/**
 * Caching plugin
 *
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
 * @uses Zend_Controller_Plugin_Abstract
 */
class My_Plugin_Caching extends Zend_Controller_Plugin_Abstract
{
    /**
     *  @var bool Désactiver le cache ou non
     */
    public static $doNotCache = false;

    /**
     * @var Zend_Cache_Frontend
     */
    public $cache;

    /**
     * @var string Clé de cache
     */
    public $key;

    /**
     * Constructeur: initialise le cache
     *
     * @param  array|Zend_Config $options
     * @return void
     * @throws Exception
     */
    public function __construct($options)
    {
        if ($options instanceof Zend_Config) {
            $options = $options->toArray();
        }
        if (!is_array($options)) {
            throw new Exception('Invalid cache options; must be array or Zend_Config object');
        }

        if
 (array('frontend', 'backend', 'frontendOptions', 'backendOptions') != array_keys($options)) {
            throw new Exception('Invalid cache options provided');
        }

        $options['frontendOptions']['automatic_serialization'] = true;

        $this->cache = Zend_Cache::factory(
            $options['frontend'],
            $options['backend'],
            $options['frontendOptions'],
            $options['backendOptions']
        );
    }

    /**
     * Démarre le cache
     *
     * Determine si le cache peut être chargé (cache hit). Si c'est le cas, retourne le contenu du cache
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @return void
     */
    public function dispatchLoopStartup(Zend_Controller_Request_Abstract $request)
    {
        if (!$request->isGet()) {
            self::$doNotCache = true;
            return;
        }

        $path = $request->getPathInfo();

        $this->key = md5($path);
        if (false !== ($response = $this->getCache())) {
            $response->sendResponse();
            exit;
        }
```


Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    }

    /**
     * Enregistre le cache
     *
     * @return void
     */
    public function dispatchLoopShutdown()
    {
        if (self::$doNotCache
            || $this->getResponse()->isRedirect()
            || (null === $this->key)
        ) {
            return;
        }

        $this->cache->save($this->getResponse(), $this->key);
    }

    public function getCache()
    {
        if( ($response = $this->cache->load($this->key)) != false) {
            return $response;
        }
        return false;
    }
}
```

Pendant ***dispatchLoopStartup()***, le plugin effecture plusieurs traitements. D'abord, il vérifie les pré-conditions : avons nous une requête GET ? Si c'est le cas, il va créer une clé de cache basée sur l'URI de la requête, et vérifie si cette clé est en cache et si elle peut être chargée (cache hit). Si c'est le cas, il sert alors la réponse depuis le cache. Dans ***dispatchLoopShutdown()***, nous faisons en sorte de regarder si l'action n'a pas indiqué qu'elle ne voulait pas être cachée, si ce n'est pas une redirection ou si nous ne pouvons calculer la clé. Dans tous les autres cas, nous créons un cache de la réponse à cette requête.

Comment dire de ne pas mettre en cache telle ou telle requête ? L'attribut statique *$doNotCache* est là pour cela :

Ne pas cacher

```
My_Plugin_Caching::$doNotCache = true;
```

Ceci supprime le mécanisme de cache pour la requête HTTP en cours.

Pourquoi avoir utilisé ***dispatchLoopStartup()*** plutôt que ***routeStartup()*** ? Et bien parce que le cache joue avec l'objet de requête, or celui-ci est altéré par le processus de routage.
Si plus tard, nous voulions modifier le calcul de la clé de cache, ou ne pas cacher certaines routes mais d'autres, certains modules / actions / contrôleurs ... Nous aurions besoin du processus de routage pour tout ceci.
Cependant, cet exemple est simple et là juste pour montrer comment utiliser les différents évènements des plugins.

II-E - Utiliser plusieurs actions (forward)

Comment dire à une action qu'elle doit être suivie d'une autre ? Dans une action, comment savoir si une action va suivre ?
La réponse se situe dans l'information *isDispatched* de l'objet de requête. Lorsque ce drapeau est mis à false, cela signifie que la requête actuelle n'a pas encore été traitée. Typiquement, c'est ce que fait la méthode ***_forward()*** du contrôleur d'action. En d'autre termes, une requête avec un drapeau *isDispatched* à false, est une nouvelle requête que le contrôleur frontal va injecter dans le processus de dispatching.

Ainsi, pour dispatcher une autre action, il faut simplement changer ce drapeau de valeur.

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Par exemple, pour aiguiller une action sur ***SearchController::formAction()***, il faut procéder comme suit :

```
$request->setModuleName('default')
        ->setControllerName('search'))
        ->setActionName('form')
        ->setDispatched(false);
}
```

Pour vérifier si une action a été dispatchée ou si elle est "fraiche" :

```
if ($request->isDispatched()) {
    // La requête a déja été traitée
} else {
    // Nouvelle requête, pas encore traitée (dispatchée)
}
```

*Vous pouvez regarder le couple plugin / aide d'action **ActionStack**. Son rôle est justement de gérer une pile d'actions. A chaque itération de la boucle de dispatching, le plugin va supprimer une action de la pile et en demander le dispatch.*
Il est donc possible de gérer une pile d'actions qui s'enchaînent.

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III - Conclusion

Question très prisée : dans quel cas utiliser un plugin, et dans quel cas utiliser une aide d'action ?
C'est très simple. Si la fonctionnalité désirée doit intéragir d'une manière ou d'une autre avec l'action en cours de dispatching, il est préférable d'utiliser une aide d'action. Par ailleurs, si la fonctionnalité ne doit être activée que pour certains modules / contrôleurs ou actions, là encore, une aide d'action est plus appropriée qu'un plugin.
En revanche, si la fonctionnalité est relative à l'application dans son ensemble (comme la gestion du cache par exemple), alors un plugin est mieux indiqué.

Les plugins et les aides d'action répondent à des design patterns et à des modèles connus de délagation de responsabilité, ils sont dans la parfaite lignée du Zend Framework. Ils servent à ajouter des fonctionnalités à l'ensemble du modèle MVC, tout en enlevant la tentation de devoir dériver une classe (héritage) importante de ce modèle (par exemple Zend_Controller_Front ou Zend_Controller_Action).
Ainsi, il est très simple d'ajouter une fonctionnalité, mais aussi de la retirer, temporairement, pour des tests par exemple, ce qui n'est par définition pas le cas de l'héritage.

Copyright © 2008 - Julien Pauli. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.