

Atelier Zend Framework : Créer une simple authentification HTTP, avec rôles

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

Date de publication : 12/11/2007

Dernière mise à jour : 27/01/2008

Le but ici est de créer un espace réservé, par authentification basique HTTP.

Dans un premier temps nous allons identifier des rôles, grâce aux ACL

Puis, nous prendrons connaissance des objets de requête, réponses, et authentification

Enfin, nous tâcherons d'utiliser ceci dans un modèle MVC à base de frontcontroller (contrôleur frontal)

- I - Zend_ACL : Liste de contrôle et de gestion d'utilisateurs
- II - Zend_Auth : authentification via Zend Framework
- III - Zend_Auth lié à Zend_Acl
- IV - Liaison avec un modèle MVC
 - IV-A - Rappel sur le Contrôleur Frontal (FC) de ZendFramework
 - IV-B - Intégrer la vérification d'identité sous forme de plugin FC
 - IV-C - Attention aux conflits de plugins
- V - Conclusion

I - Zend_ACL : Liste de contrôle et de gestion d'utilisateurs

Zend_ACL va nous permettre de tenir sur le site une liste de contrôle des utilisateurs. Zend_ACL fait la liaison entre des ressources, et des rôles.

Dans notre exemple nous aurons 2 rôles : "administrator", et "guest". Administrator a le droit d'accéder à la ressource "My private space". Guest lui, a le droit d'accéder tout simplement au site.

En revanche si l'on ne fournit pas d'identifiants, alors on se voit interdit d'accès au site.

```
<?php
$acl = new Zend_Acl();
$acl->add(new Zend_Acl_Resource('My private space'));
$acl->addRole(new Zend_Acl_Role('administrator'));

/**
 * autorise uniquement si l'assertion renvoie true
 */
$acl->allow('administrator', 'My private space', null, new AclAssertion($result =
    $adapter->authenticate()->getIdentity()));
```

Sans parler ici de la partie qui contient authenticate() , **allow()** accepte 4 paramètres, dont 2 obligatoires.

Le premier est le nom du rôle (ou un objet le représentant), le second est le nom de la ressource (ou un objet la représentant).

De la même manière, nous possédons une méthode **deny()**, qui elle, définit explicitement une interdiction. Par défaut, si on ne spécifie aucune liaison entre les ressources, et les rôles, alors ZF considère que tout est interdit.

Nous autorisons (**allow()**) donc l'administrateur, à accéder à la ressource 'My Private Space', mais en 4ème paramètre, nous passons une assertion. Une assertion est une vérification, qui renvoie true si la règle est vérifiée, et false sinon.

Les assertions sont utilisées aussi dans le code afin de le tester. Ainsi, si cette assertion répond true, alors oui, **allow()** va fonctionner, et notre administrator se verra offrir l'accès à My private space.

En revanche, si l'assertion n'est pas vérifiée, et qu'elle renvoie false, la règle **allow()** ne sera pas suivie, et donc par défaut administrator ne pourra accéder à My private space.

L'assertion prend la forme d'une classe, ici elle a même un constructeur. Toutes les assertions pour Zend_Acl doivent implémenter l'interface **Zend_Acl_Assert_Interface**, et donc définir toutes ses méthodes. Ici, l'interface ne nous définit qu'une seule méthode :**assert()**

II - Zend_Auth : authentification via Zend Framework

L'authentification est une étape que l'on rencontre souvent dans le web. C'est tout simplement donner une identité à un client. Pour ceci, il a besoin de fournir des informations.

Dans cet atelier, il les fournira via le processus d'authentification du protocole HTTP. Les RFCs **2616** et **2617** vous renseigneront d'avantage. L'objet qui va nous y aider est un `Zend_Auth_Adapter_Http`

Il faut ensuite comparer ces données avec une source, ce sera le rôle d'un objet `Zend_Auth_Adapter_Http_Resolver_File`, les identifiants seront donc stockés dans un fichier. C'est pour HTTP actuellement le seul 'conteneur'.

Dans la logique des choses, `Zend_Auth_Adapter_HTTP` va recevoir une requête HTTP, il va, selon ses paramètres, remplir une réponse HTTP. Toujours selon les RFC, si le client ne fournit pas les bons identifiants, ou ne fournit pas d'identifiants du tout, alors une réponse 401 lui est envoyée. Cet en-tête demande des identifiants au client, les navigateurs savent alors qu'il faut afficher une fenêtre, pour les demander. Notez que certains d'entre eux vous empêcheront de faire plus de X essais à la suite, de même, si vous appuyez sur "annuler" à ce moment là, ils envoient des identifiants vides.

Selon donc la requête que `Zend_Auth_Adapter_HTTP` va recevoir, il va remplir les en-têtes de la réponse HTTP.

Nous utiliserons la méthode HTTP dite "Basic", il existe aussi une méthode "Digest", un peu plus complexe, mais plus sécurisée. La méthode Basic possède l'inconvénient de faire transiter en presque clair (base64 encodé), les identifiants à chaque requête, avec le risque qu'ils soient interceptés. Je vous renvoie une fois de plus à la [RFC2617](#) pour plus d'infos

```
<?php
$config = array(
    'accept_schemes' => 'basic',
    'realm'           => 'dvp',
);

// identification HTTP
$adapter = new Zend_Auth_Adapter_Http($config);

// resolveur d'identification : fichier pour HTTP
$basicResolver = new Zend_Auth_Adapter_Http_Resolver_File();
$basicResolver->setFile('auth.txt');

$adapter->setBasicResolver($basicResolver);

// passage de la requête HTTP actuelle
$adapter->setRequest(new Zend_Controller_Request_Http());
// puis passage d'un objet de réponse HTTP
$adapter->setResponse($response = new Zend_Controller_Response_Http());
```

Remarquez comme chaque objet a son rôle, comme l'application est correctement découplée, il est très simple ici de changer d'authentification, par exemple utiliser une base de données; sans pour autant devoir changer la logique générale de notre mini application.

C'est ce que j'aime au sein de Zend Framework, je trouve sa conception très bonne, certes ça n'est pas parfait, car rien ne l'est, mais on peut clairement utiliser les objets que l'on souhaite, ils ont été prévus pour s'assembler entre eux, ceci, grâce à la puissance du modèle objet de PHP5.

III - Zend_Auth lié à Zend_Acl

Maintenant que nous avons presque tout défini, (il manque l'assertion tout de même), il est très simple d'effectuer un contrôle d'accès :

```
<?php
// ... à la suite
if ($acl->isAllowed('administrator', 'My private space') ) // si administrateur
{
    $response->setHttpResponseCode(200);
    $response->appendBody('welcome administrator ');
}elseif(!empty($result)){ // sinon si guest : il y a un résultat suite à authenticate()
    $response->setHttpResponseCode(200);
    $response->appendBody('welcome to the guest part of the website');
}else{ // enfin, si l'authentification a échoué
    $response->appendBody('Please, authenticate');
}
// pour finir, renvoie simplement la réponse HTTP au client.
$response->sendResponse();
```

Remarquez comment je manipule l'objet réponse, une réponse HTTP, qui fait partie du modèle complet MVC de Zend Framework. Ici, point de frontController, mon MVC n'est basé que sur une seule page, que je n'ai pas eu envie de surcharger avec trop d'objets.

Il n'empêche que l'objet **Zend_Controller_Response_Http** représente réellement une réponse HTTP. Cet objet contient une partie en-tête, et une partie corps (je vous renvoie à la RFC2616 sur le protocole HTTP, si vous n'y êtes pas un minimum familier).

Cet objet de réponse est manipulé par **Zend_Auth** dans un premier temps, puis je peux prendre la main dessus. Je modifie le code de réponse HTTP (**setHttpResponseCode()**) dans le cas où 'tout va bien'. Un code 200 est renvoyé et le corps de la réponse est rempli avec un message (**appendBody()**).

Dans le cas où l'authentification n'a pas encore été effectuée, ou si le client a décidé d'annuler la requête, alors je lui envoie le message 'please authenticate', le code de retour HTTP est 401 dans ce cas là, mais c'est Zend_Auth qui l'a rempli automatiquement, je n'ai pas besoin de m'en soucier.

Enfin, **sendResponse()** "flush" la réponse, tous les en-têtes qu'elle contient sont envoyés au navigateur, et son corps est lu.

L'assertion, elle, ressemble à ceci :

```
class AclAssertion implements Zend_Acl_Assert_Interface
{
    const ADMINKEY = "admin";

    private $_aclResult;

    public function __construct(array $result)
    {
        $this->_aclResult = $result;
    }

    /**
     * Zend_Acl_Assert_Interface
     */
    public function assert(Zend_Acl $acl, Zend_Acl_Role_Interface $role = null,
```

```
        Zend_Acl_Resource_Interface $resource = null, $privilege = null)
    {
        return (array_key_exists("username", $this->_aclResult) && $this->_aclResult['username'] ==
self::ADMINKEY) ? true : false;
    }
}
```

Je passe au constructeur de cette classe, le résultat de l'authentification. L'appel de la méthode **authenticate()** sur l'objet **Zend_Auth** va lui faire lire la requête (**Zend_Controller_Request_Http**)

Si l'en-tête de cette requête contient un champ 'authorization' (c'est le protocole HTTP qui veut ça), alors c'est que le client a fourni des identifiants (valides, ou pas).

le résultat de **authenticate()** est un objet **Zend_Auth_Result**. Sa méthode **getIdentity()** retourne un tableau, si la clé 'username' y existe, alors c'est que le processus d'identification a fonctionné : le client a fourni un couple login/pass valide.

Dans le cas contraire, le tableau retourné par **getIdentity()** est vide.

Ainsi, le constructeur de mon assertion prend en paramètre ce tableau, vérifie si la clé "username" y existe, et si elle vaut la valeur "admin". Si c'est le cas, alors c'est que l'administrateur s'est identifié : l'assertion doit retourner true.

Voyez aussi, pour repérer un 'guest', il suffit de voir si ca n'est pas un administrateur, et si le tableau retourné par **getIdentity()** est non vide. Alors, c'est quelqu'un qui a fourni un couple login/pass valide, mais qui n'est pas administrateur : c'est un 'guest'.

Quant au fichier stockant les mots de passe, il prend la forme login:realm:pass. Le 'realm' est un identifiant de site, ici nous avons décidé qu'il avait la valeur 'dvp'. De cette manière, un seul fichier peut stocker des infos pour plusieurs sites, plusieurs domaines, plusieurs parties distinctes d'une même application.

le fichier auth.txt

```
admin:dvp:admin
guest:dvp:guest
```

IV - Liaison avec un modèle MVC

IV-A - Rappel sur le Contrôleur Frontal (FC) de ZendFramework

Le processus de contrôleur frontal est un design pattern très connu qui s'intègre dans un modèle MVC. Le but du FC est de prendre en charge la totalité du traitement de la requête cliente.

Pour ceci, le FC joue le rôle de chef d'orchestre, c'est d'ailleurs un singleton, à juste titre. Il va faire intervenir beaucoup d'objets, et va les faire jouer ensemble.

On notera un objet de requête, un objet de réponse, un ensemble d'objet routeurs, un dispatcher, un ensemble de contrôleurs dits "d'action", des aides aux actions (Action Helpers), puis des plugins et un objet conteneur de plugins.

Notez que FC ne touche pas à l'objet de vue.

Le processus complet du MVC de Zend Framework peut être résumé à ce plan là :

- 1 -FC initialise les plugins, le router, et le dispatcher
- 2 -FC execute routeStartup() pour tous les plugins qui lui sont envoyés
- 3 -FC execute le routeur pour récupérer un objet \$request correspondant à un module, controller, action
- 4 -FC execute routeShutdown() pour tous les plugins qui lui sont envoyés
- 5 -FC execute dispatchLoopStartup() pour tous les plugins qui lui sont envoyés
- 6 -FC entre dans la boucle de dispatching
- 7 ---FC marque {\$request} comme dispatchée (isDispatched() = TRUE)
- 8 ---FC execute preDispatch() pour tous les plugins qui lui sont envoyés
- 9 ---FC demande au dispatcher de dispatcher la requête \$request
- 10 -----dispatcher instancie le contrôleur d'action
- 11 -----dispatcher marque la requête \$request comme dispatchée
- 12 -----dispatcher execute dispatch() sur le contrôleur d'action
- 13 -----le contrôleur d'action execute son init()
- 14 -----le contrôleur d'action execute notifyPreDispatch() pour tous les helpers qui lui sont passés
- 15 -----le contrôleur d'action execute son preDispatch()
- 16 -----le contrôleur d'action execute la requête \$action (PRINCIPAL)
- 17 -----A ce stade on peut demander à FC d'abandonner immédiatement tout le processus si une exception est envoyée
- 18 -----le contrôleur d'action execute son postDispatch()
- 19 -----le contrôleur d'action execute notifyPostDispatch() pour tous les helpers qui lui sont passés
- 20 -----FC execute postDispatch() pour tous les plugins qui lui sont envoyés
- 21 -----la boucle de dispatching recommence si la requête n'est pas marquée comme dispatchée (redirection)
- 22 -FC sort de la boucle de dispatching, le traitement de l'action est terminé
- 23 -FC execute dispatchLoopShutdown() pour tous les plugins qui lui sont envoyés
- 24 -FC retourne \$response ou l'affiche directement

IV-B - Intégrer la vérification d'identité sous forme de plugin FC

Nous interviendrons sur le predispatch des plugins, en créant justement un plugin pour cela. Nous pourrions agir sur le dispatchLoopStartup, ou le routeShutdown (moins approprié).

L'idée est : en predispatch, je vérifie l'identité, si elle n'est pas bonne, je change la requête, et je la dirige vers une page qui signale qu'une authentification est requise.

L'idée est toute simple, mais dès qu'on manipule un plugin, il y a danger de marcher sur les autres plugins. 2 plugins peuvent avoir des effets ping-pong désagréables qu'il faut bien prendre en note et analyser. Nous verrons que ça peut être le cas avec le plugin préenregistré "ErrorHandler".

Notre bootstrap

```
<?php
$appPath = realpath(dirname(__FILE__) . '/../application');

set_include_path($appPath . PATH_SEPARATOR . get_include_path() );

require_once 'Zend/Loader.php';
Zend_Loader::registerAutoload();

$authconfig = array(
    'accept_schemes' => 'digest',
    'realm'           => 'anaska',
    'digest_domains' => '/',
    'nonce_timeout'  => 3600,
);

$httpauth = new Zend_Auth_Adapter_Http($authconfig);
$resolver = new Zend_Auth_Adapter_Http_Resolver_File();
$resolver->setFile($appPath . '/auth.txt');

$httpauth->setDigestResolver($resolver);
$httpauth->setRequest($request = new Zend_Controller_Request_Http());
$httpauth->setResponse($response = new Zend_Controller_Response_Http());

$front = Zend_Controller_Front::getInstance();
$response = $front->throwExceptions(false) // comportement par défaut
    ->registerPlugin(new MyPluginAuth($httpauth, new MyAcl()))
    ->setRequest($request)
    ->setResponse($response)
    ->setControllerDirectory($appPath . '/controllers/')
    ->dispatch();
```

un contrôleur d'index banal

```
<?php
class indexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->view->message = "Bienvenue";
    }
}
```

Je vous passe la vue...

Remarquez : il est impératif que le FC partage les mêmes objets de requête et réponse que l'authentificateur.

Nous avons vu dans les chapitres précédents que l'objet **Zend_Auth_Adapter_Http** gère la requête du client (ses éventuels identifiants), ainsi que la réponse qu'il lui retourne (les en-têtes en particulier). Notez aussi que pour changer, nous utilisons cette fois ci un procédé d'identification HTTP dit "digest", il diffère un peu du procédé dit "basic" : il est cependant bien plus sécurisé, et autorise l'identification à perdre sa valeur dans le temps.

On peut le dire : l'adapter auth http et le processus MVC matérialisé (ici) par le FC se "court-circuitent" au niveau des objets de requête, et de réponse. Mais aucune inquiétude, tout a été prévu.

Notre ACL ressemble beaucoup à celle du chapitre précédent, 2 rôles : guest et admin. Les ressources sont des noms de contrôleur, qu'il faudra donc vérifier.

Notre site comporte 2 contrôleurs principaux :

- 1 l'index : le contrôleur par défaut, il faut s'identifier en tant que guest (invité), pour y accéder.
- 2 private : un espace privé, seul un admin peut y avoir accès, les privilèges guest sont insuffisants

Si on voulait créer d'autres contrôleurs, il faudrait à chaque fois les intégrer dans l'ACL; du moins c'est comme cela que ce tutoriel est tourné, évidemment des systèmes plus adéquats existent pour des cas plus "banaux".

MyACL.php

```
<?php
class MyAcl extends Zend_Acl
{
    public function __construct()
    {
        $this->add(new Zend_Acl_Resource('index'));
        $this->add(new Zend_Acl_Resource('private'));

        $this->addRole(new Zend_Acl_Role('guest'));
        $this->addRole(new Zend_Acl_Role('admin'), 'guest');

        // Les invités peuvent uniquement voir le contenu
        $this->allow('guest', 'index');
        $this->allow('admin', 'private');
    }
}
```

Notre plugin va utiliser cette classe ACL :

MyPluginAuth.php

```
<?php
class MyPluginAuth extends Zend_Controller_Plugin_Abstract
{
    private $_auth;
    private $_acl;

    const FAIL_AUTH_MODULE      = 'default';
    const FAIL_AUTH_ACTION      = 'auth';
    const FAIL_AUTH_CONTROLLER = 'auth';

    const FAIL_ACL_MODULE      = 'default';
    const FAIL_ACL_ACTION      = 'acl';
    const FAIL_ACL_CONTROLLER = 'auth';

    public function __construct(Zend_Auth_Adapter_Interface $auth, Zend_Acl $acl)
    {
        $this->_auth = $auth;
        $this->_acl  = $acl;
    }

    /**
     * Vérifie les autorisations
     * Utilise _request et _response hérités et injectés par le FC
     *
     * @param Zend_Controller_Request_Abstract $request : non utilisé, mais demandé par l'héritage
     */
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $role = $this->_auth->authenticate()->getIdentity();

        if (!empty($role)) {
            $role = $role['username'];
            try {

```

MyPluginAuth.php

```
if (!$this->_acl->isAllowed($role, $this->_request->controller) &&
is_null($this->_request->getParam('error_handler'))) {
    $this->getResponse()->clearAllHeaders();
    $this->_request->setParam('role fourni',$role)
        ->setModuleName(self::FAIL_ACL_MODULE)
        ->setControllerName(self::FAIL_ACL_CONTROLLER)
        ->setActionName(self::FAIL_ACL_ACTION);
}
} catch (Zend_Acl_Exception $e){
// l'ACL ne trouve pas la ressource, probablement contrôleur ou action incorrects :
non répertoriés dans l'ACL
return;
}
//Est ce qu'on a déjà essayé ça et qu'un problème est survenu ? (capté par le plugin
error_handler)
} elseif (is_null($this->_request->getParam('error_handler'))) {
    $this->_request->setModuleName(self::FAIL_AUTH_MODULE)
        ->setControllerName(self::FAIL_AUTH_CONTROLLER)
        ->setActionName(self::FAIL_AUTH_ACTION)
        ->setDispatched(true);
}
}
}
```

En predispatch, donc avant le traitement de notre action (la page demandée en général, ou une partie de celle-ci), on demande l'identité (**authenticate()**), ceci aura pour action de faire apparaître une fenêtre de demande d'identification http au client.

\$role récupère ce qu'il a saisi, la clé ['username'] va définir le rôle. On interroge alors l'ACL pour lui demander si cet identifiant, peut accéder ou non, au contrôleur demandé actuellement dans la requête. (On aurait pu intégrer l'action aussi par exemple). Si ça n'est pas le cas, alors on re-route la requête vers le contrôleur d'échec ACL.

Que se passe-t-il si on demande l'accès à un contrôleur inexistant (non routé) ? La demande d'ACL va renvoyer une exception, on la capture simplement et avec un return, on quitte le plugin.

Le plugin de postDispatch, **ErrorHandler**, va lui traiter le fait que le contrôleur est introuvable. En parlant de ce plugin : poursuivons :

Si le client ne fournit pas d'identifiants (appui sur annuler, trop d'essais avec échec : le comportement est propre au navigateur), nous modifions la requête pour lui faire pointer vers un contrôleur d'échec d'authentification cette fois.

IV-C - Attention aux conflits de plugins

Que se passe-t-il si ce contrôleur là, n'est pas trouvé, ou aussi, si le contrôleur d'échec ACL n'est pas trouvé ?

Il faut faire attention car "en dessous" de notre plugin auth, il y a **le plugin ErrorHandler**, activé par défaut, et peut-être encore d'autres. Le rôle de celui-ci est de traiter justement les cas où le contrôleur, le module ou l'action sont introuvables (entres autres cas).

Si il agit, il redémarre toute la boucle de dispatching pour dispatcher vers un contrôleur d'erreur. Et notre plugin auth va ré-intervenir Il va donc à nouveau recharger la requête, vers un contrôleur toujours invalide.

ErrorHandler va repasser une fois de plus derrière, mais vu qu'il a été prévu pour gérer ce genre de conflit, il va envoyer une exception au FC.

Il faut donc vérifier dans notre plugin si le ErrorHandler n'est pas déjà passé par là.

```
}elseif (is_null($this->_request->getParam('error_handler'))) { vérifie ceci, car ce plugin "marque" la requête.
```

Le contrôleur d'erreur pourra alors être traité. Cependant, nous avons bien entendu un contrôleur authentification en bonne et dû forme, le voici :

Contrôleur authentification

```
<?php
class AuthController extends Zend_Controller_Action
{

    public function init()
    {
        $this->getHelper('ViewRenderer')->setNeverRender(true);
        $this->_response->clearBody();
    }

    public function authAction()
    {
        $this->_response->appendBody('authentication required','auth');
    }

    public function aclAction()
    {
        $this->_response->appendBody($this->_request->getParam('role fourni') . " n'est pas autorisé à accéder à cette ressource",'auth');
    }

}
```

Je ne rends pas de vue, pour simplifier, je ne fais que remplir manuellement la réponse (rendre une vue, par défaut, consiste en fait à la traduire, et à ajouter son contenu à la suite du segment par défaut).

On pourra noter que le code de réponse est 401, car l'objet auth a manipulé la réponse avant le processus MVC.

Voyez un peu ces sales effets que des tests d'intégration ont justement pour but d'éliminer. Le plugin ErrorHandler a été prévu pour ne pas entrer dans une boucle infinie de dispatching.

Si les module/contrôleur/action d'erreur ont été demandés en dispatch, mais non trouvés, alors (dernier recours) : une exception est lancée.

Pour éviter cela, on aurait pu utiliser un plugin qui agit sur dispatchLoopStartup. Soit, AVANT la boucle de dispatching, et donc sans effets de bords (à priori). Cependant il est important que ce plugin d'authentification agisse dans la boucle.

Si un contrôleur, validé, demande la redirection vers un contrôleur a plus hauts privilèges ? Là, notre plugin doit intervenir pour stopper l'accès, mais s'il est hors boucle, il ne pourra plus intervenir sur cette requête (dans la logique de la conception).

De la même manière, les plugins enregistrés sur les mêmes événements peuvent aussi être incompatibles entre eux ou avoir des effets de bord. Individuellement, ils fonctionnent, mais il faut vérifier qu'ils fonctionnent tous ensemble.

La méthode **registerPlugin()** du FC, possède un 2ème paramètre qui va situer ledit plugin dans la pile des plugins.

On saisit un entier, et FC charge alors tous les plugins dans l'ordre donné. Il commence par le paramètre de pile le plus bas, vers le plus haut.

V - Conclusion

Zend Framework est décidément plaisant. Il ne propose pas une manière unique de faire, mais propose des composants qui eux, sont clairement utiles.

Ca sera après à chacun de les implémenter de la manière dont ils le souhaitent.

Cet atelier, comme d'habitude, a pour but de faire comprendre un mécanisme, et le code ici nécessitera des arrangements avant de pouvoir être utilisé. Il ne fait qu'exposer des idées d'implémentation et de construction avec Zend Framework.

Vous pouvez aussi suivre ces quelques liens :

[Zend_Acl / Zend_Auth scénario d'exemple](#)

[Débuter avec le Zend Framework \(approche MVC\)](#)

[Notre rubrique consacrée au Zend Framework](#)

