

Atelier Zend Framework : Introduction au templating avec Zend_Layout

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

Date de publication : 21/01/2008

Dernière mise à jour : 04/04/2008

L'intégration et l'utilisation de template sont indispensables sur le web. Zend_Layout est le composant natif de Zend Framework, qui représente une solution très flexible.

Voyons ensemble à quoi ressemble Zend_Layout, et introduisons aussi quelques autres composants qui s'interconnectent bien ensemble.

Nous allons décrire quelques utilisations possibles, allant du fonctionnement des layouts, jusqu'à l'application d'une approche MVC dite 'push'

- I - Zend_Layout : définition et buts
 - I-A - Principe
 - I-B - Configuration
- II - Intégration dans le modèle MVC
 - II-A - Le fichier de template
 - II-B - Modèle MVC push
 - II-B-1 - Couple front-plugin/action-help : ActionStack
 - II-B-2 - Helper de vue : ActionHelper
- III - Autres helpers de vue
- IV - Conclusion

I - Zend_Layout : définition et buts

Zend_Layout est un composant basé sur le design pattern "Two Step View" (vue en 2 temps). Son but est clair : offrir un moyen d'utiliser le templating, à savoir d'intégrer des vues, dans des vues.

Comme à son habitude, Zend Framework offre une flexibilité très bonne. Le composant Zend_Layout arrive avec une configuration de base, mais il est appréciable de noter qu'elle est complètement personnalisable.

Les services de Zend_Layout son multiples :

- 1 Rendre automatiquement un ou des templates (un template est une vue contenant d'autres morceaux de vues)
- 2 Fournir un moyen de configurer l'objet Layout (nom des templates, chemins) , par défaut, il suit les mêmes règles que ViewRenderer
- 3 Permettre une approche utilisant le modèle MVC de ZF, ou pas (indépendance)

Zend_Layout apporte une solution native élégante au problème du templating. Accompagné d'une petite panoplie de helpers de vue, il va nous rendre bien des services.

I-A - Principe

Le principe du two step view est résumé **ici**. Nous savons que par défaut, grâce à **ViewRenderer**, chaque action rend une vue.

Ce principe reste totalement inchangé. Ce qui change en activant les layouts, c'est qu'au lieu que la vue soit rendue telle quelle; son rendu va être intégré dans une vue de plus haut niveau, appelé Layout, ou template (on peut jouer sur les mots). C'est le principe de la vue en 2 temps :

- 1 On rend la vue désirée, en général grâce à ViewRenderer (automatique : correspond à l'action dispatchée)
- 2 On rend un layout, dans lequel va être intégrée la vue précédemment rendue, ainsi qu'éventuellement, d'autres vues, issues éventuellement aussi, d'autres actions parallèles

Le layout est donc un fichier .phtml (une vue, oui), dans lequel on va marquer à quels endroits, on veut rendre quelle vue, ou quelle action.

Quelques notions peuvent paraître complexes, mais il n'en est rien.

I-B - Configuration

Bien qu'il soit possible, de mettre des layouts, dans les layouts (etc...), cet atelier restera simple, et utilisera un seul template; c'est d'ailleurs un cas assez fréquent sur le terrain.

La configuration est simple. Nous avons un objet Zend_Layout (on ne sait pas comment, pour le moment admettons le) , il faut qu'il connaisse plusieurs choses :

- 1 Le fichier de template, qui va contenir des 'marques' pour savoir où rendre les vues, la clé de config est 'layout' et il s'appelle layout.phtml par défaut
- 2 Le chemin d'accès aux fichiers de templates, nous avons dit que dans cet atelier nous n'aurons qu'un seul template, par défaut il s'agit du dossier app/view placé dans la clé de config 'layoutPath'

- 3 La clé spéciale qui, appelée dans le template, va rendre la vue actuellement dispatchée, par défaut il s'agit de 'layout' placé dans la clé de config 'contentKey'

Accessoirement, il est possible de configurer d'autres paramètres : Zend_Layout est connecté au modèle MVC via un plugin de contrôleur frontal, ainsi qu'un action helper. Nous détaillerons ceci plus tard.

Zend_Layout est compatible avec Zend_Config, et sait lire un objet instance de Zend_Config pour sa configuration. Accessoirement, tout objet présentant une méthode **toArray()** peut être utilisé aussi.

II - Intégration dans le modèle MVC

Pour entrer directement dans le vif du sujet, voici un exemple de template (de Layout), issu de la documentation officielle, ce fichier est donc un fichier .phtml (par défaut) :

```
<?= $this->docType('XHTML1_STRICT') ?>
<html>
  <head>
    <?= $this->headTitle() ?>
    <?= $this->headScript() ?>
    <?= $this->headStylesheet() ?>
  </head>
  <body>


Header
  
<?= $this->partial('header.phtml') ?>



Navigation
  
<?= $this->layout()->nav ?>



Sidebar
  
<?= $this->layout()->sidebar ?>



Content
  
<?= $this->layout()->content ?>



Footer
  
<?= $this->partial('footer.phtml') ?>


  </body>
</html>
```

A première vue on semble comprendre comment cela fonctionne, mais il y a des astuces au niveau conceptuel à bien intégrer :

Déjà, il faut pouvoir initialiser le Layout. En effet, notre modèle MVC n'est pas affecté par Zend_Layout, avant qu'on ne le lui spécifie clairement.

Ainsi, si vous ne touchez à rien; votre application MVC ne va pas utiliser Zend_Layout par défaut. Pour l'utiliser, Zend_Layout fournit une méthode statique : **startMVC()**.

bootstrap

```
<?php
// 'dev' ou 'prod'
define('APP_USE_TYPE', 'dev');

error_reporting(E_ALL | E_STRICT);
```

bootstrap

```
// Dans toute appli MVC, l'appDir doit être connu et rajouté à l'include_path
$appDir = realpath(dirname(dirname(__FILE__))) . '/app';
set_include_path($appDir . PATH_SEPARATOR . $appDir . '/modele' . PATH_SEPARATOR .
get_include_path());

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

/**
 * Configuration du site
 */
$config = new Zend_Config_Ini($appDir . '/config/config.ini', APP_USE_TYPE);

/**
 * base de données
 */
try{
    $db = Zend_Db::factory($config->database);
    $db->getConnection();
    Zend_Db_Table::setDefaultAdapter($db);
} catch(Exception $e){
    exit($e->getMessage());
}

Zend_Session::start();

/**
 * MVC
 */
$frontController = Zend_Controller_Front::getInstance();
$frontController->setParam('noErrorHandler', true); // j'ai décidé de ne pas utiliser le plugin
ErrorHandler dans cet article
$frontController->setParam('db', $db);
$frontController->throwExceptions(true);

Zend_Layout::startMvc($config->layout); // activation des layouts, sans ça, le modèle MVC demeure
le même que celui que vous connaissez

$frontController->setControllerDirectory($appDir . '/controllers');
try{
    $frontController->dispatch(); // dispatche !
} catch(Exception $e){
    exit($e->getMessage());
}

?>
```

Comme dans tous mes ateliers, j'essaie d'aller droit au but. Ce bootstrap est très simple, cependant il va utiliser les Layout, car la ligne `Zend_Layout::startMVC()` le lui indique.

Notez que j'ai passé mon objet de config à `Zend_Layout`, voyons le fichier de configuration :

config.ini

```
[app]
database.adapter      = pdo_mysql
database.params.host  = localhost
database.params.username = myname
database.params.password = mypass
database.params.dbname = mydb

layout.layout        = template
layout.contentKey    = contenu
```

config.ini

```
[dev : app]

[prod : app]
database.params.host = my.prod.host
```

Je ne **reviens pas** sur le fonctionnement de Zend_Config ni sur l'héritage des sections. Simplement, Zend_Layout attend pour configuration les clés 'layout', 'contentKey' ... je les ai détaillées plus haut.

J'ai décidé que mon fichier de template s'appelle 'template', il se situe à la racine de mon dossier de vue (comportement par défaut : je n'ai pas spécifié ce paramètre), et dans ce fichier, le paramètre qui me permettra de rendre le contenu de la vue actuellement dispatchée s'appelle 'contenu'. Par défaut, je ne l'ai pas changée non plus, l'extension d'un fichier de template est la même que celle des vues, à savoir : phtml.

La méthode **startMVC()** effectue plusieurs actions : Elle crée une instance de Zend_Layout, puis elle enregistre un plugin de contrôleur frontal, ainsi qu'un action helper.

Le plugin est enregistré avec un numéro de pile de manière à ce qu'il soit l'avant dernier à intervenir, en postdispatch bien entendu, le dernier étant **errorHandler**.

Son rôle est de prendre la vue rendue automatiquement **par viewRenderer**, puis de rendre le fichier de template en intégrant le contenu de cette vue là où la clé spéciale de rendu est localisée (dans notre cas : 'template').

L'action helper lui, va nous servir à accéder à l'objet layout, si on le désire, dans nos actions. On va pouvoir ainsi le piloter sereinement via ce helper; c'est un design pattern Proxy, un peu modifié pour l'occasion.

Nous allons rapidement passer à la pratique, simplement notez que le plugin et l'action helper sont des classes que ZF fournit avec Zend_Layout, mais que vous avez la possibilité d'entièrement réécrire. Zend_Layout vous offre pour ceci **setPluginClass()**, **setHelperClass()**, et même si vous le désirez, **setView()**. Par défaut Zend_Layout extrait la vue de viewRenderer, vous n'avez donc qu'un objet vue dans tout votre processus, sauf si vous désirez un comportement alternatif (Zend_Layout a besoin de Zend_View pour rendre le fichier de template).

II-A - Le fichier de template

Voici ce à quoi ressemble notre fichier de template, le code n'est pas compatible XHTML, et la mise en page est en tableaux.

Vous avez le droit de me cracher dessus si vous le désirez :), au passage notez qu'il n'a rien à voir avec le schéma ci dessus, qui est je répète un schéma d'exemple issu de la doc officielle.

```
<table border='1' cellpadding="0" cellspacing="0" width="100%" height="100%">
<tr>
<td colspan="3" align="center"><?php echo $this->layout()->header ?></td>
</tr>
<tr>
<td width="20%"><?php echo $this->layout()->gauche?></td>
<td><?php echo $this->layout()->contenu?></td>
<td width="20%"><?php echo $this->layout()->droite?></td>
</tr>
<tr>
<td colspan="3" align="center"><?php echo $this->layout()->footer ?></td>
</tr>
</table>
```

J'accède à mon objet Layout simplement avec **`$this->layout()`**, et comme je l'ai dit, mon action est rendue dans la clé que j'ai spécifié en configuration de Zend_Layout, à savoir 'contenu'.

Ainsi, si je laisse viewRenderer faire son travail, tout est rendu dans la clé que j'ai définie dans ma Layout. C'est en fait parce que par défaut le viewRenderer rend la vue correspondant à l'action dispatchée dans le segment 'default' de la réponse.

Voilà l'astuce géniale : si je rends une action dans un segment autre de la réponse HTTP, alors ce rendu sera accessible dans Zend_Layout via la clé du segment de la réponse.

Regardez comme j'utilise plusieurs clés : header, droite, gauche et footer. Ainsi, je vais en une seule action, rendre une vue par segment :

mon action

```
<?php
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->view->message = "test de message";
        $this->render('index');
        $this->render('header', 'header');
        $this->render('droite', 'droite');
        $this->render('gauche', 'gauche');
        $this->render('footer', 'footer');
    }
}
```

view/scripts/index/header.phtml

```
<a href="<?php echo $this->url(array('controller'=>'index'))?>">Accueil</a>
```

view/scripts/index/index.phtml

```
Voici l'affichage de index.phtml<br />
avec le message <b><?php echo $this->message; ?></b>
```

C'est bien, mais pas optimal, car toutes les vues se trouvent dans le même dossier, et bien que seule la vue centrale (index), contienne une variable, elles auraient pu toutes en contenir.

Une seule action ne peut se contenter de faire le travail pour toutes les vues, c'est en fait une à plusieurs actions par vue, qui doivent être exécutées.

Si je ne rend rien pour mon layout, alors il n'affiche tout simplement rien (je ne parle donc pas encore de 'gauche', et 'droite').

II-B - Modèle MVC push

Il est ainsi possible d'utiliser un modèle **MVC de type 'push'**. Le modèle push possède la particularité d'interdire à la vue, conceptuellement, d'accéder aux données métier.

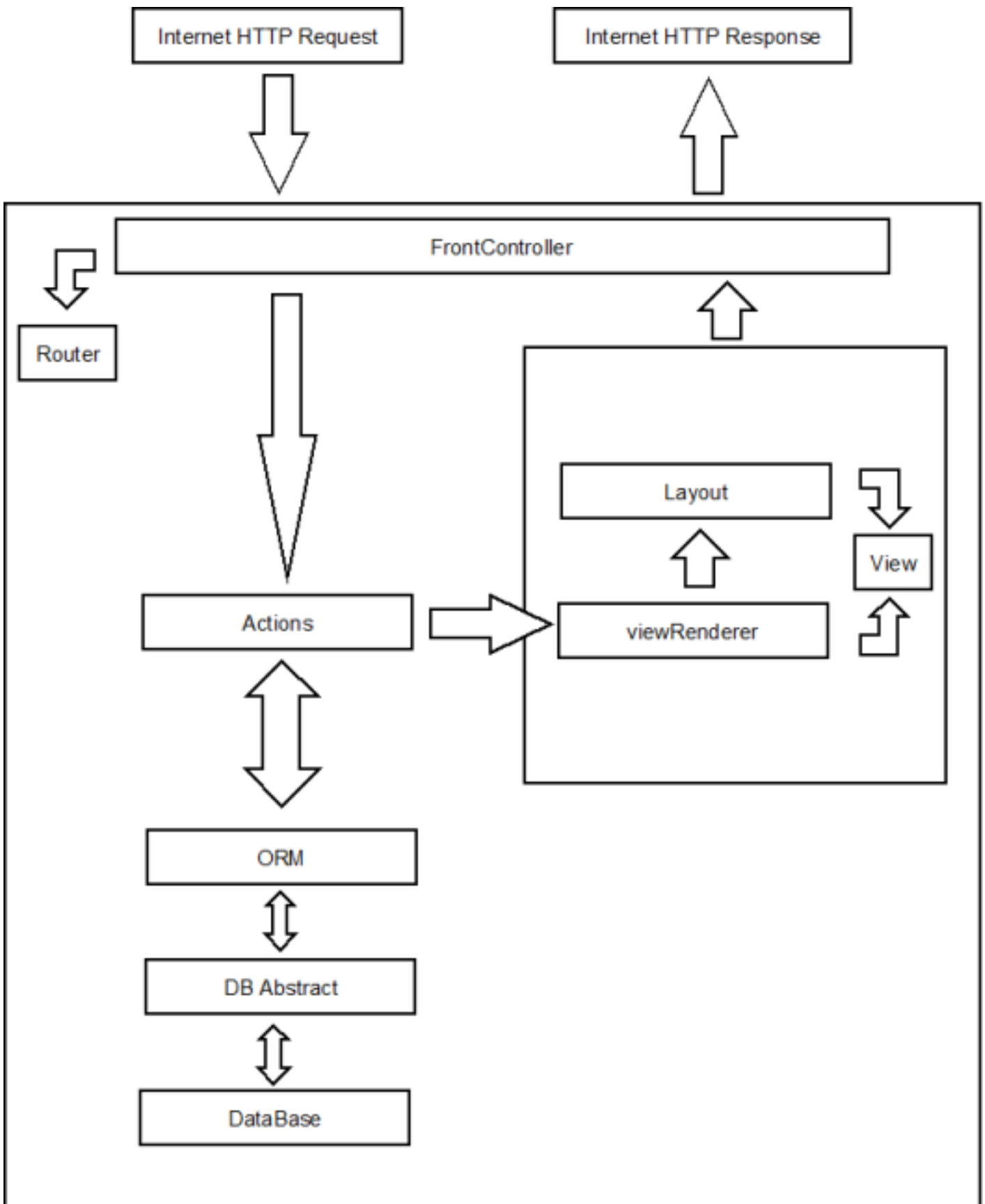
Plus simplement : toute vue doit être rendue par un contrôleur ou plusieurs contrôleurs (cas complexes). La vue n'accèdera jamais d'elle même au modèle.

J'aime ce type d'architecture MVC : elle est plus contraignante, mais plus découplée : nous avons 3 couches réellement identifiées, testables, et réutilisables, et puis toutes les actions vont hériter d'un mode de traitement commun

Cependant on voit bien que dans la méthode que j'ai utilisée, ca n'est pas top. Il est mieux de bien séparer les traitements, et de faire confiance à viewRenderer (tant qu'à faire).

Ainsi, il faut construire les autres actions (correspondant à header, footer, etc...), et j'ai plusieurs méthodes pour le faire.

Je pourrai faire une suite de **_forward()**, seulement dès que je veux changer mon processus : je galère, et pour les architectures complexes, il peut facilement y avoir des dizaines de **_forward()**.



II-B-1 - Couple front-plugin/action-helper : ActionStack

Un nouveau couple action-helper/front-plugin entre alors en jeu : **ActionStack** : c'est une gestion de pile d'actions, qui sont exécutées dans l'ordre FILO, voyons ça :

nouveau bootstrap

```
<?php
// le code ne change pas jusque là :

/**
 * MVC
 */
$frontController = Zend_Controller_Front::getInstance();
$frontController->setParam('noErrorHandler', true);
$frontController->setParam('db', $db);
$frontController->setRequest(new Zend_Controller_Request_Http()); // un bug relevé, actuellement
cette ligne est obligatoire
$frontController->throwExceptions(true);
Zend_Layout::startMvc($config->layout);

$actionStack = Zend_Controller_Action_HelperBroker::getStaticHelper('actionStack');
$actionStack->actionToStack('footer', 'index');
$actionStack->actionToStack('header', 'index');

$frontController->setControllerDirectory($appDir . '/controllers');
try{
    $frontController->dispatch(); // dispatche !
}catch(Exception $e){
    $log->emerg($e->getMessage());
}
```

nouveau indexcontroller

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

    public function headerAction()
    {
        $this->_helper->viewRenderer->setResponseSegment('header');
    }

    public function footerAction()
    {
        $this->view->titre = 'Atelier Zend_Layout';
        $this->_helper->viewRenderer->setResponseSegment('footer');
    }
}
```

Je récupère en bootstrap le helper *actionStack*, et je lui ajoute 2 actions : action header / contrôleur index et action footer / contrôleur index. Ordre FILO je répète : header sera exécuté en premier.

Tout ceci sera exécuté après mon premier dispatch, *actionStack* agit en *postDispatch*, et avant le plugin *Zend_Layout*, qui n'intervient qu'à la fin de toutes les requêtes.

Chaque action que vous voyez utilise ce dont elle a besoin, pour ensuite se rendre chacune dans son segment de réponse (grâce au `viewRenderer`). Je vous laisse factoriser le comportement si besoin.

On notera que la vue du footer utilise une variable, le traitement se fait bien dans l'action (même si ici le traitement est quasi nul, pour l'exemple), action qui passe à la vue son résultat.

Ici, j'ai décidé d'utiliser l'actionStack en bootstrap, rien ne vous empêche de l'utiliser où vous voulez dans votre logique contrôleurs.

Bien entendu, le plugin (et non plus l'action helper du même nom) propose des méthodes pour modifier la pile : **`popStack()`**, **`pushStack()`**, **`getStack()`**.

II-B-2 - Helper de vue : ActionHelper

Autre méthode : intégrer le rendu de la vue header et footer, dans le template directement. Je rappelle au passage que le fichier de template, bien qu'utilisé par `Zend_Layout`, demeure rendu par `Zend_View` : toutes les méthodes de `Zend_View` y existent.

Nous ne pouvons pas utiliser **`$this->render()`** : la vue serait alors rendue hors contexte MVC, si elle a besoin de variables, elle va devoir les chercher elle-même : banni (dans notre cas de modèle 'push').

Plutôt que **`$this->render()`**, un nouveau view-helper est arrivé : `ActionHelper`. Il suffit d'utiliser **`$this->action()`**, nous allons faire cela pour le segment de gauche :

```
<table border='1' cellpadding="0" cellspacing="0" width="100%" height="100%">
  <tr>
    <td colspan="3" align="center"><?php echo $this->layout()->header ?></td>
  </tr>
  <tr>
    <td width="20%"><?php echo $this->action('gauche', 'login') ?></td>
    <td><?php echo $this->layout()->contenu?></td>
    <td width="20%"><?php echo $this->layout()->droite?></td>
  </tr>
  <tr>
    <td colspan="3" align="center"><?php echo $this->layout()->footer ?></td>
  </tr>
</table>
```

Lorsque le template est rendu, **`action()`** va lancer une instance clonée du modèle MVC en cours d'exécution, pour exécuter en parallèle l'action demandée, sur le contrôleur demandé (et le module : facultatif).

Le rendu de la réponse de cette action est alors retourné (le code HTML en gros). Si l'action demande une redirection, elle n'est pas suivie, et rien n'est retourné => il ne s'agit pas de lancer toute une suite de tâches "parallèle".

Ainsi, ceci peut être pratique pour les petits rendus, par exemple : un formulaire d'authentification ? :

Vérification d'identité, si oui affichage d'un coucou, sinon : affichage d'un formulaire d'authentification, que voici d'ailleurs :

view/scripts/login/form.phtml

```
<form method="POST" action="<?php echo $this->url(array('controller'=>'login')) ?>">
  Login : <?php echo $this->formText('login', '')?>
  Password : <?php echo $this->formPassword('password', '')?>
```

```
view/scripts/login/form.phtml
```

```
<?php echo $this->formSubmit('Identifiant','identifiant')?>
</form>
```

Si on est identifié, ceci s'affiche :

```
view/scripts/login/hello.phtml
```

```
Hello <?php echo $this->identity ?>
<br />
<a href="<?php echo $this->url(array('controller'=>'login','action'=>'deco'))?>">Deconnexion</a>
```

Voilà qui nous amène donc à un contrôleur pour le login. Rien qu'ici, on voit bien qu'on a cassé notre application, et que grâce aux templates, chaque conteneur du template est rendu par son (sa suite d') action.

Pour ce contrôleur, faites un tour sur mon atelier Zend Framework : [Authentification HTTP avec Zend_Auth et Zend_Acl](#). Il vous expliquera les bases de l'authentification même si ici nous utiliserons une base de données, plutôt que HTTP.

```
LoginController
```

```
<?php
class LoginController extends Zend_Controller_Action
{
    protected $_flashMessenger;
    const MIN_CHARACTERS_FOR_CREDENTIALS = 6;

    public function init()
    {
        parent::init();
        $this->_flashMessenger = $this->_helper->FlashMessenger;
    }

    public function indexAction()
    {
        $auth = Zend_Auth::getInstance();
        if ($auth->hasIdentity())
        {
            $this->_redirect('/');
        }

        if ($this->_hasParam('login') && $this->_hasParam('password'))
        {
            $adapter = new Zend_Auth_Adapter_DbTable($this->getFrontController()->getParam('db'),
            'users', 'login', 'password');

            $stringLength = new Zend_Validate_StringLength();
            $stringLength->setMin(self::MIN_CHARACTERS_FOR_CREDENTIALS);

            $validator = new Zend_Validate();
            $validator->addValidator($stringLength);

            if (!$validator->isValid($this->_getParam('login')) ||
            !$validator->isValid($this->_getParam('password')))
            {
                $this->_flashMessenger->addMessage(sprintf("les identifiants doivent faire %d
                caractères minimum",self::MIN_CHARACTERS_FOR_CREDENTIALS));
                $this->_forward('error');
                return;
            }

            $adapter->setIdentity($this->_getParam('login'))
            ->setCredential($this->_getParam('password'));
        }
    }
}
```

LoginController

```

$result = $auth->authenticate($adapter);
if (!$result->isValid())
{
    $this->_flashMessenger->addMessage('authentification incorrecte');
    $this->_forward('error');
}
else{
    Zend_Session::regenerateId();
    $this->_redirect('/');
}
else{
    $this->_flashMessenger->addMessage('Veuillez fournir des identifiants');
    $this->_forward('error');
}
}

public function gaucheAction()
{
    $auth = Zend_Auth::getInstance();
    if ($auth->hasIdentity())
    {
        $this->view->identity = $auth->getIdentity();
        $this->render('hello');
    }
    else{
        $this->render('form');
    }
}

public function errorAction()
{
    $this->view->authmessage = $this->_flashMessenger->getCurrentMessages();
}

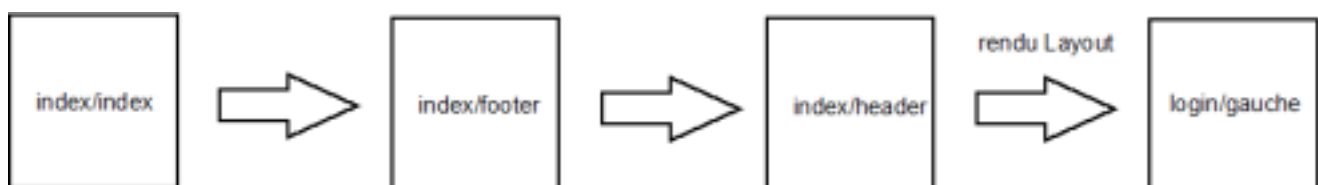
public function decoAction()
{
    $auth = Zend_Auth::getInstance();
    if ($auth->hasIdentity())
    {
        Zend_Session::forgetMe();
        Zend_Session::destroy();
    }
    $this->_redirect('/');
}
}

```

Je ne détaille pas trop le fonctionnement de Zend_Auth, mais voyez comme chaque traitement se fait dans une action spécifique.

Le template demande gauche, pour l'afficher dans le segment de gauche.

Les autres actions sont rendues lors de l'interrogation de ce contrôleur par le formulaire, dans le processus MVC normal; elles sont donc rendues via leurs viewRenderer, dans le segment contenu de la template (comme l'est le contrôleur d'index).



III - Autres helpers de vue

Petit tour rapide d'autre helpers qui ont pour but de simplifier la mise en page d'un site web; comme les layouts. Le `placeholder()` est pratique et est destiné à la réutilisabilité de code fixe.

Des listes préformatées, des formulaires, ou des tableaux de données.

Voyons un exemple, pas des meilleurs pour cette application, mais vous comprendrez l'intérêt :

views/scripts/login/form.phtml

```
<?php echo $this->partial('form.htm', array(
    'action' => $this->url(array('controller'=>'login')),
    'login' => 'login',
    'password' => 'password',
    'submit' => 'S\'identifier',
));

echo $this->authmessage
?>
```

J'ai simplement séparé le dessin de mon formulaire, de ces variables. L'exemple semble idiot je vous l'accorde, ça peut servir à réutiliser le formulaire alors (même si login et password restent écrits 'en dur'). Le fichier form.htm ressemble alors à ceci :

views/scripts/form.thm

```
<form method="POST" action="<?php echo $this->action ?>">
  Login : <?php echo $this->formText($this->login,$this->initialLogin)?><br />
  Password : <?php echo $this->formPassword($this->password,$this->initialPassword)?><br />
  <?php echo $this->formSubmit($this->submit,$this->submit)?><br />
</form>
```

Dans le même style, des helpers spécifiques existent pour la gestion des balises

- 1 <doctype
- 2 <script>
- 3 <title>
- 4 <style>
- 5 <link>
- 6 <meta>

```
<?php
$request = Zend_Controller_Front::getInstance()->getRequest();

echo $this->doctype('XHTML1_STRICT'),
$this->headLink()->appendStylesheet('/styles/basic.css')
->headLink(array('rel' => 'favicon', 'href' => '/img/favicon.ico'), 'PREPEND')
->prependStylesheet('/styles/moz.css', 'screen', true),

$this->headMeta()->appendName('keywords', 'framework php productivity')
->appendHttpEquiv('expires', 'Wed, 26 Feb 1997 08:21:57 GMT')
->appendHttpEquiv('pragma', 'no-cache')
->appendHttpEquiv('Cache-Control', 'no-cache'),
```

```
$this->headScript()->appendFile('/js/prototype.js')
    ->appendScript($onloadScript),

$this->headStyle()->appendStyle($styles),

$this->headTitle($request->getActionName())
    ->headTitle($request->getControllerName());
```

La documentation vous en apprendra plus. Il reste quelques options, telles que ***captureStart()***, pour capturer des patterns et les rejouer dans d'autres vues,, ou ailleurs...

IV - Conclusion

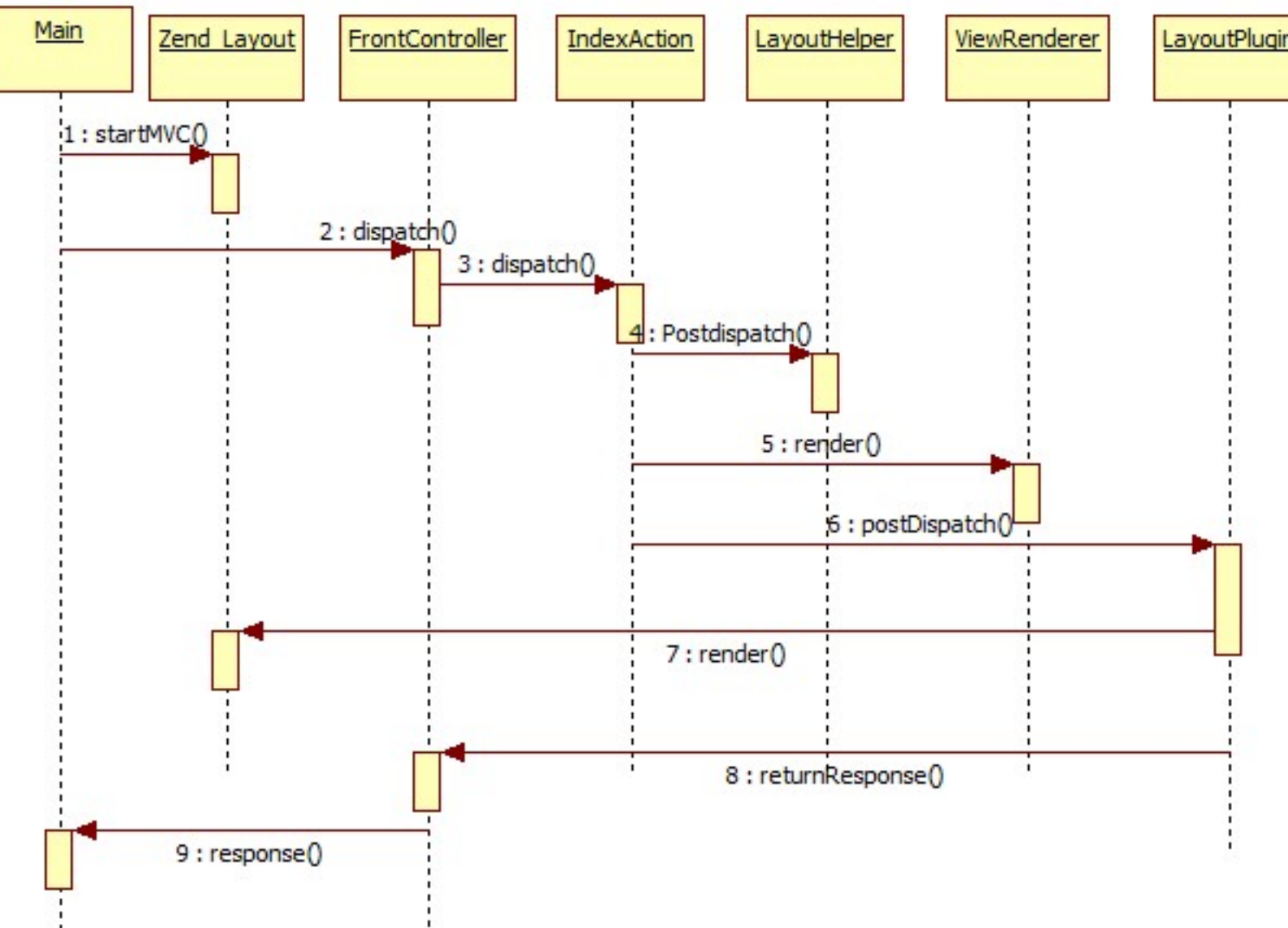


Diagramme de séquence simplifié

Cette maigre application a eu pour but de vous faire comprendre le pattern utilisé pour Zend_Layout, et son fonctionnement global.

Zend Framework nous présente une solution de templating toute prête, et surtout très flexible.

En témoignent les méthodes du style **setLayoutPath()**, **setPluginClass()**, **setHelperClass()**, **setView()** ...

De la même manière, je n'ai pas traité le cas mais Zend_Layout n'est pas un composant de MVC. Son arborescence est telle qu'il n'est pas intégré dans le modèle MVC, il peut donc être utilisé en dehors. Parallèlement, le modèle MVC peut être utilisé sans Zend_Layout.

