

# Le modèle MVC et le contrôleur sous PHP

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

Date de publication : 28/06/2007

Dernière mise à jour : 26/10/2007

De tous les motifs de conception (Design patterns), le motif MVC est sans doute celui sur lequel il y a le plus à dire.

Je vais cependant rester simple, le but de cet article est de comprendre MVC, et pourquoi MVC ?

- I - Introduction
- II - Principe du modèle MVC
- III - Le frontController ou contrôleur frontal
- IV - Motif actionController
- V - Conclusions

## I - Introduction

Le MVC tout comme l'orientation objet du code, semble être devenu un standard dans le développement d'applications web, avec la réputation d'être une bonne pratique de conception.

Cependant, trouver une définition exacte et précise du MVC semble impossible, notamment au regard de PHP, langage scripté interprété à chaque requête, à la différence de java avec qui on peut utiliser les [threads](#). De nombreux Frameworks utilisent aujourd'hui [MVC](#), car le but principal de ce motif est de séparer les couches logiques d'une application. Nous allons voir qu'il n'est pas toujours judicieux d'utiliser MVC, ce motif sera plutôt réservé aux sites touffus, dans lesquels beaucoup de redondances de code apparaissent en général, des sites souvent mis à jour, souvent remodelés. Dans le cadre d'une petite application web gentilette, la complexité en terme de code, apportée par MVC, ne sera pas justifiée. En revanche pour tout autre projet web, la séparation en plusieurs couches permet à différentes équipes de bosser chacune sur une couche, indépendamment des autres.

Et c'est aussi un gros avantage lorsqu'une couche doit évoluer, sans que les autres n'en aient besoin.

Voyons déjà un petit code :

```
<?php
$connect = mysql_connect('myserver', 'mylogin', 'mypassword');
mysql_select_db('myDB');
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $news_id = $_POST['news_id'];
    mysql_query("INSERT INTO commentaires SET news_id='$news_id',
                auteur='".mysql_escape_string($_POST['auteur'])."',
                texte='".mysql_escape_string($_POST['texte'])."',
                date=NOW()"
                );
    header("location: ".$_SERVER['PHP_SELF']."?news_id=$news_id");
    exit;
} else {
    $news_id = $_GET['news_id'];
}
?>
<html>
<head>
<title>Les news</title>
</head>
<body>
<h1>Les news</h1>
<div id="news">
    <?php
    $news_req = mysql_query("SELECT * FROM news WHERE id='$news_id'");
    $news = mysql_fetch_array($news_req);
    ?>
    <h2><?php echo $news['titre'] ?> postée le <?php echo $news['date'] ?></h2>
    <p><?php echo $news['texte_nouvelle'] ?> </p>
    <?php
    $comment_req = mysql_query("SELECT * FROM commentaires WHERE news_id='$news_id'");
    $nbre_comment = mysql_num_rows($comment_req);
    ?>
    <h3><?php echo $nbre_comment ?> commentaires relatifs à cette nouvelle</h3>
    <?php while ($comment = mysql_fetch_array($comment_req)) {?>
        <h3><?php echo $comment['auteur'] ?> a écrit le <?php echo $comment['date'] ?></h3>
        <p><?php echo $comment['texte'] ?></p>
    <?php } ?>
    <form method="POST" action="<?php echo $_SERVER['PHP_SELF'] ?>" name="ajoutcomment">
    <input type="hidden" name="news_id" value="<?php echo $news_id?>">
    <input type="text" name="auteur" value="Votre nom"><br />
    <textarea name="texte" rows="5" cols="10">Saisissez votre commentaire</textarea><br />
```

```
<input type="submit" name="submit" value="Envoyer">
</form>
</div>
</body>
</html>
```

C'est volontairement simple et simplifié, peu sécurisé, ceci pour demeurer concis et clair, constat : tout le monde connaît un code comme celui-ci, et ça ne choquera personne. PHP est mélangé au sein du HTML, après tout c'est son but premier : langage préprocesseur.

A la lecture du code, on comprend immédiatement ce à quoi il est destiné, c'est l'avantage. On appelle ça un traitement procédural. L'inconvénient, c'est que les couches sont mélangées. Dans le traitement d'informations sur le web, généralement on distingue 3 couches : le modèle qui gère la manière d'accéder aux données, la vue qui se charge de présenter des données, et le contrôle, c'est l'étape intermédiaire de sélection et fusion des données. On peut donc factoriser le code pour bien séparer ces 3 couches, voyez plutôt :

#### mymodel.php

```
<?php
function dbconnect()
{
    static $connect = null;
    if ($connect === null) {
        $connect = mysql_connect('myserver', 'mylogin', 'mypassword');
        mysql_select_db('myDB');
    }
    return $connect;
}

function get_news($id)
{
    $news_req = mysql_query("SELECT * FROM news WHERE id='$news_id'",dbconnect());
    return mysql_fetch_array($news_req);
}

function get_comment($news_id)
{
    $comment_req = mysql_query("SELECT * FROM commentaires WHERE news_id='$news_id'",dbconnect());
    $result = array();
    while ($comment = mysql_fetch_array($comment_req)) {
        $result[] = $comment;
    }
    return $result;
}

function insert_comment($comment)
{
    mysql_query("INSERT INTO commentaires SET news_id='{ $comment['news_id']}',

    auteur='".mysql_real_escape_string($comment['auteur'])."',
                                texte='".mysql_real_escape_string($comment['texte'])."',
                                date=NOW()"

    ,dbconnect() );
}
```

#### myview.php

```
<html>
<head>
    <title>Les news</title>
</head>
<body>
    <h1>Les news</h1>
```

## myview.php

```
<div id="news">
<h2><?php echo $news['titre'] ?> postée le <?php echo $news['date'] ?></h2>
<p><?php echo $news['texte_nouvelle'] ?> </p>
<h3><?php echo $nbre_comment ?> commentaires relatifs à cette nouvelle</h3>
<?php foreach ($comments AS $comment) {?>
  <h3><?php echo $comment['auteur'] ?> a écrit le <?php echo $comment['date'] ?></h3>
  <p><?php echo $comment['texte'] ?></p>
<?php } ?>
<form method="POST" action="<?php echo $_SERVER['PHP_SELF'] ?>" name="ajoutcomment">
  <input type="hidden" name="news_id" value="<?php echo $news_id?>">
  <input type="text" name="auteur" value="Votre nom"><br />
  <textarea name="texte" rows="5" cols="10">Saisissez votre commentaire</textarea><br />
  <input type="submit" name="submit" value="Envoyer">
</form>
</div>
</body>
</html>
```



## mycontroller.php


```
<?php
require ('mymodel.php');
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
  insert_comment($_POST);
  header("HTTP/1.1 301 Moved Permanently");
  header("location: {$_SERVER['PHP_SELF']}news_id={$_POST['news_id']}");
  exit;
} else {
  $news = get_news($_GET['news_id']);
  $comments = get_comments($_GET['news_id']);
  require ('myview.php');
}
?>
```

Et voilà, on a appliqué le modèle **MVC** ;-). On a bien séparé le code en 3 couches distinctes, qui se connaissent l'une l'autre. Dans ce code-là, on peut facilement changer la manière d'accéder aux données (changer de SGBD, par exemple), sans pour autant se soucier le moins du monde de la présentation, ou de la manipulation future de ces données. Et nous n'avons pas utilisé la POO ;-).

Voilà l'intérêt principal d'un modèle MVC : séparer les couches de conception, de manière à ce que plusieurs groupes de personnes puissent bosser chacun sur leur couche, sans même qu'ils connaissent les personnes qui bossent la couche du dessous (ou du dessus, ou d'à coté).



Il n'est cependant pas toujours facile de bien distinguer les couches, et le motif MVC reste sans aucun doute un motif complexe, car, nous allons le voir, on peut fortement le complexifier, avec MVC2, mais les développeurs habitués à ce modèle apprécient particulièrement la lisibilité du code.

3 couches sont un minimum, MVC2 en implémente minimum 4, et on peut même en distinguer jusqu'à 5 voire 6 ou plus sur des gros projets. Mais comme toujours, la séparation des entités logiques, réalisable grâce à la programmation orientée objet, simplifie et clarifie la conception d'un logiciel (et dans notre cas d'une application web). Il est plus facile de dire de quoi est constituée une vinaigrette, lorsque les ingrédients sont tous posés sur la table, plutôt que d'essayer d'analyser leur mélange au fond du saladier. Il n'y a pas à proprement parler de nombre de classes défini, MVC est un  **design pattern** architectural, c'est un ensemble de principes à suivre à l'intérieur duquel un grand nombre d'autres  **motifs de conception** interviennent.

On peut ainsi créer "plein de" motifs MVCs, chacun répondra à un problème particulier. Cependant la plupart du temps, les développeurs utiliseront des modèles MVC déjà écrits, dans des  **frameworks** ayant fait leurs preuves.

## II - Principe du modèle MVC

Le but de MVC est de séparer les couches d'une application (en 3 couches distinctes, au minimum, et le plus souvent). On va donc distinguer :

- 1 Le modèle qui encapsule la logique métier et la manipulation des sources de données. Ici, des design patterns peuvent intervenir : TableDataGateway, ActiveRecord ....
- 2  **La vue** : présente les données à l'utilisateur. Ici aussi on peut introduire des motifs : Factory ,  **Composite**, TemplateView; par exemple
- 3 Le contrôleur : c'est tout le reste, c'est lui qui analyse la requête client, accède aux données, formate le tout, et balance le tout à la partie vue, qui va afficher ça.


On va détailler un peu, **la vue d'abord**, c'est un TemplateView. On utilise PHP lui-même pour écrire nos variables dans du HTML, ou alors un moteur de template.

Le TransformView peut aussi être utilisé pour sortir en transformant les données renvoyées par les modèles, qui sont en  **XML**, grâce à  **XSLT** et les feuilles de style  **XSL**. Une sortie en PDF peut aussi être utilisée.

Il est très important de noter que la vue ne fait que présenter des données, c'est le modèle qui va fournir toutes les méthodes nécessaires à la présentation. Si par exemple on doit afficher en gras les commentaires datant de moins de 3 jours, le modèle fournira une méthode d'accès à ces commentaires, et la vue comportera une condition de vérification.

Du côté du **modèle**, il est très important que celui-ci ne connaisse ni les contrôleurs, ni les vues. Les liens vont des contrôleurs vers le modèle, de temps en temps, des vues vers le modèle (on va voir ça), mais jamais dans le sens inverse.

Le modèle étant totalement indépendant du reste, il pourra faire l'objet d'une réflexion et d'un développement à part du reste. Il pourra de même être utilisé par d'autres composants, tels que des webservices.

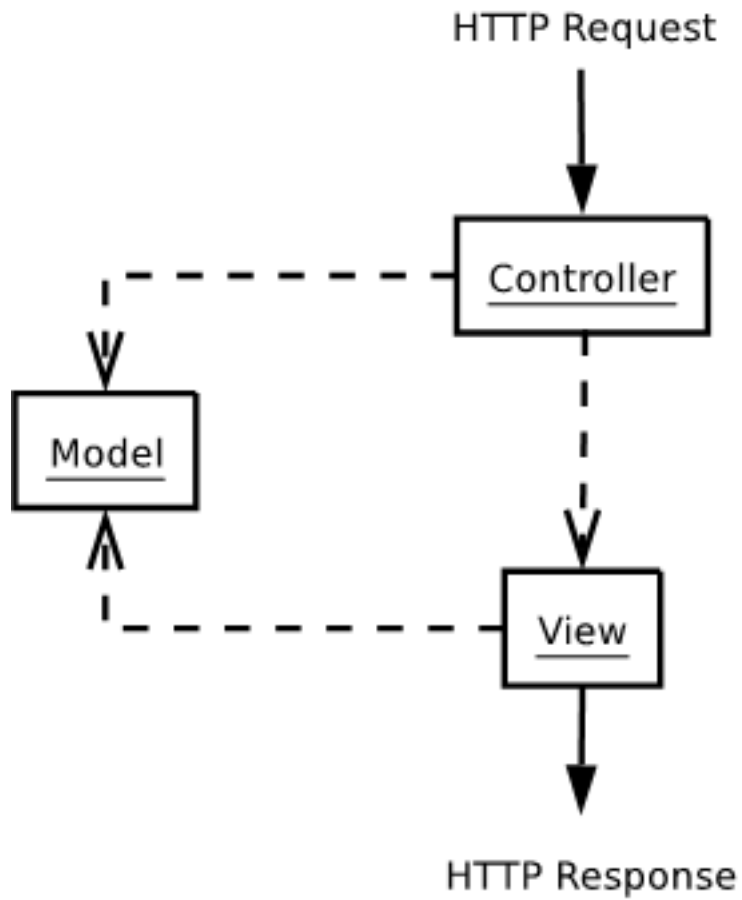
Il existe plein de méthodes d'accès aux données. Dans la majeure partie des cas, la source de données est une base de données, et le mapping objet relationnel ( **ORM**) pourra être utilisé, ou l'ActiveRecord, ce qui ne fait pas l'objet de cet article.

Il est possible que la vue accède au modèle. C'est un choix architectural qui devra être bien réfléchi, car il impose un plus fort couplage et un mélange des couches. On appelle ça l'approche pull. C'est souvent réservé aux opérations légères, comme par exemple afficher une liste de catégories d'éléments quelconques ; et aux opérations de lecture uniquement.

L'autre approche, à l'inverse, appelée push, interdit à la vue d'accéder au modèle. En général, on utilise les 2 choix en même temps : une instruction de lecture simple (récupérer la liste des membres) sera demandée directement par la vue, ou un de ses sous-composants.

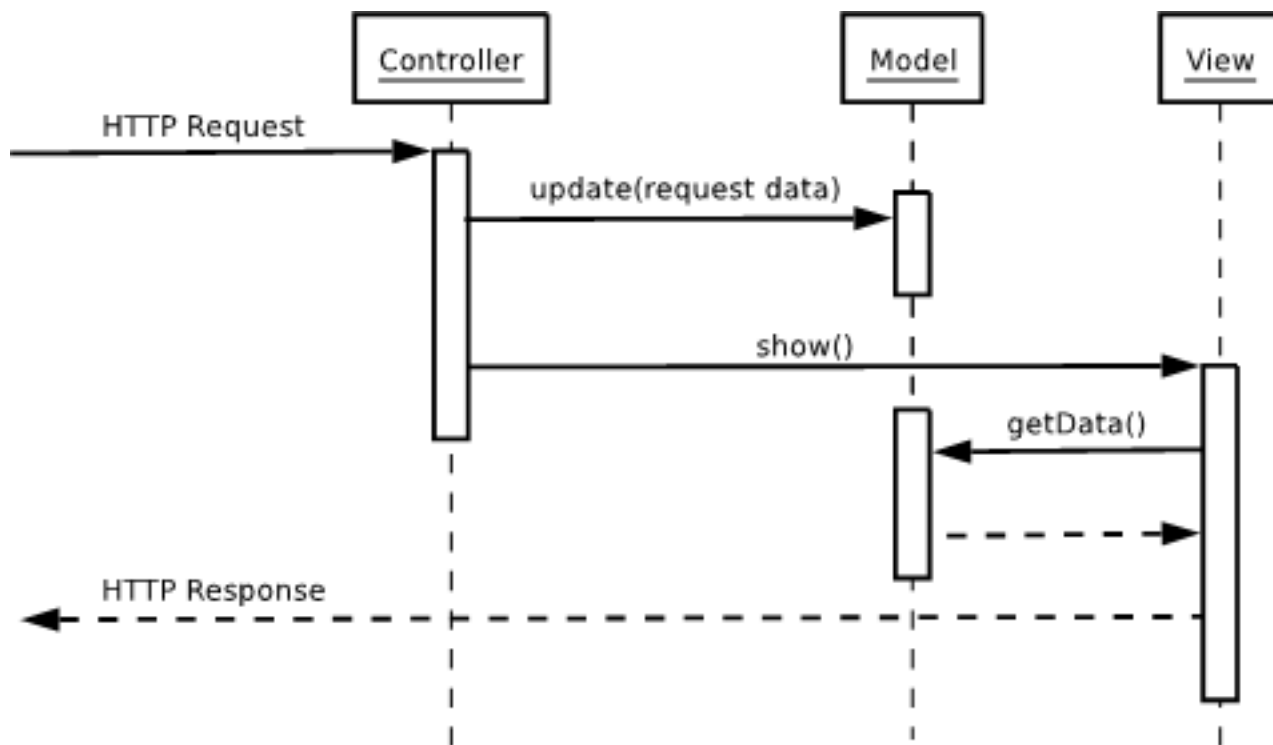
En revanche une opération de modification du modèle, sera demandée par un contrôleur.

*Modèle MVC global :*



*MVC général*

*Diagramme de séquence d'un MVC simple :*



MVC details

Et on arrive enfin au **contrôleur**, c'est un peu l'objet de cet article, car c'est lui le composant principal du modèle MVC. Il agit avec le modèle, la vue, et d'autres contrôleurs, à la fois.

Quel qu'il soit, le contrôleur reçoit et analyse la requête du client, accède au modèle, et présente la vue.

### III - Le frontController ou contrôleur frontal

Il existe plusieurs motifs associés au contrôleur. Le PageController impose un contrôleur par page. C'est le schéma 'logique' qui nous vient à l'esprit, l'exemple montré plus haut implémente cette idée.

De manière simple nous avons séparé les 3 couches, et chaque action à réaliser va se trouver dans un fichier distinct, par exemple `post_comment.php`, `read_new.php`; etc...

#### Le contrôleur doit donc :

- 1 Analyser la requête, soit extraire les informations dans l'URL
- 2 Faire une mise à jour du modèle si nécessaire, en lui passant des paramètres
- 3 Déterminer la vue à afficher et demander son affichage

Chaque contrôleur regroupe l'ensemble des responsabilités dans un seul fichier, appelé dans l'URL, c'est le motif le plus clair et le plus facile à mettre en œuvre, la POO n'étant pas nécessaire.

Cependant dans des applications qui gagnent un peu en complexité, de la factorisation s'impose, afin de séparer certains rôles :

Le **FrontController**, ou contrôleur frontal, est un motif qui est destiné à prendre en charge l'analyse de la requête client. Il est donc le seul et unique point d'entrée de l'application, et va orchestrer toute la suite. C'est un singleton et généralement on l'appelle `index.php` et on lui fournit les actions demandées dans l'URL :

`http://monsite.com/index.php?action=show_livres&categorie=x` ou pour faire plus joli  
`http://monsite.com/index.php/action?categorie=x`

Le site étant en général plus complexe que ça, l'url intégrera aussi ce qu'on appelle un module :

`http://monsite.com/index.php?module=bibliotheque&action=show_livres&categorie=x` ou pour faire plus joli  
`http://monsite.com/index.php/bibliotheque/show_livres?categorie=x`

FrontController est associée au design pattern Command qui va utiliser des objets `$request` et `$response` pour représenter les actions. La classe `request` est en charge de l'analyse de l'URL fournie, elle va déterminer un module, une action, et des paramètres. Le frontController appelle alors le fichier correspondant à l'action (motif Command), et va appeler sa méthode `launch()`. L'action va être exécutée. A la fin, l'action demande le rendu d'une vue via `render()`, puis son affichage via `printOut()`, mais elle peut aussi demander une redirection grâce à `redirect()`, ou passer la main à une autre action avec `forward()`.

#### classe Request

```
<?php
class Request
{
    public function getParam($key)
    {
        return filter_var($this->getTaintedParam($key), FILTER_SANITIZE_STRING,
        FILTER_FLAG_NO_ENCODE_QUOTES);
    }

    public function getTaintedParam($key)
    {
        if ($_SERVER['REQUEST_METHOD'] == 'POST' && isset($_POST[$key])){
            return $_POST[$key];
        }else{
            return $_GET[$key];
        }
    }
}
```

## classe Request

```
}
}

public function route()
{
    $requestUri = substr($_SERVER['REQUEST_URI'],
        strpos($_SERVER['REQUEST_URI'], '/'.basename(__FILE__)) +
        strlen('/'.basename(__FILE__)))
    );
    if (empty($requestUri)) return array();

    $path = parse_url($requestUri, PHP_URL_PATH);
    preg_match('#^(/(?P<module>w+))/(?P<action>w+)?/?$#', $path, $matches);

    $args = explode('&', parse_url($requestUri, PHP_URL_QUERY));

    return $matches;
}
}
```

## classe Response

```
<?php
class Response
{
    private $_vars = array();
    private $_headers = array();
    private $_body;

    public function addVar($key, $value)
    {
        $this->_vars[$key] = $value;
    }

    public function getVar($key)
    {
        return $this->_vars[$key];
    }

    public function setVar($key, $value)
    {
        $this->_vars[$key] = $value;
    }

    public function getVars()
    {
        return $this->_vars;
    }

    public function setBody($value)
    {
        $this->_body = $value;
    }

    public function redirect($url, $permanent = false)
    {
        if ($permanent){
            $this->_headers['Status'] = '301 Moved Permanently';
        }else{
            $this->_headers['Status'] = '302 Found';
        }
        $this->_headers['location'] = $url;
    }

    public function printOut()
```

## classe Response

```
{
    foreach ($this->_headers as $key => $value) {
        header($key . ':' . $value);
    }
    echo $this->_body;
}
```

## classe frontController

```
class frontController
{
    private $_defaults = array('module' => 'index', 'action' => 'index');
    private $_request;
    private $_response;
    private static $_instance = null;

    private function __construct()
    {
        $this->_request = new Request();
        $this->_response = new Response();
    }

    public static function getInstance()
    {
        if (is_null(self::$_instance)) {
            self::$_instance = new self();
        }
        return self::$_instance;
    }

    public function dispatch($defaults = null)
    {
        $parsed = $this->_request->route();
        $parsed = array_merge($this->_defaults, $parsed);
        $this->forward($parsed['module'], $parsed['action']);
    }

    public function forward($module, $action)
    {
        $command = $this->_getCommand($module, $action);
        $command->launch($this->_request, $this->_response);
    }

    private function _getCommand($module, $action)
    {
        if (!file_exists($path = "$module/$action.php")) {
            throw new Exception("Commande inconnue $module/$action.php");
        }
        require($path);
        $class = $action.'Action';
        return new $class($this);
    }

    public function getResponse()
    {
        return $this->_response;
    }

    public function redirect($url)
    {
        $this->_response->redirect($url);
    }

    public function render($file)
    {

```

**classe frontController**

```
$view = new View();
$this->_response->setBody($view->render($file,$this->_response->getVars()));
}
```

**classe View**

```
class View
{
    public function render($file, $assigns = array())
    {
        extract($assigns);
        ob_start();
        require($file);
        $str = ob_get_contents();
        ob_end_clean();
        return $str;
    }
}
```

**classe Action**

```
abstract class Action
{
    protected $_controller;

    public function __construct($controller)
    {
        $this->_controller = $controller;
    }

    abstract public function launch(Request $request, Response $response);

    public function render($file)
    {
        $this->_controller->render($file);
    }

    public function printOut()
    {
        $this->_controller->getResponse()->printOut();
    }

    protected function _forward($module, $action)
    {
        $this->_controller->forward($module, $action);
    }

    protected function _redirect($url)
    {
        $this->_controller->redirect($url);
    }
}
```

**index (main)**

```
$front = frontController::getInstance()->dispatch();
```

Le code est déjà plus complexe, on est passé à de l'objet. On le pilotera de cette manière :

**une commande quelconque**

## une commande quelconque

```
<?php
public function launch(Request $request, Response $response)
{
    $variable = 'essai de variable';
    $response->addVar('unevariable', $variable);
    $this->render(dirname(__FILE__) . '/show.php');
    $this->printOut();
}
}
```

## show.php

```
<?php
echo $unevariable
?>
```

C'est rapide, mais c'est destiné à vous faire intégrer ce concept de séparation des rôles, c'est ça qui est important dans MVC.

Il ne sert à rien d'avoir une structure telle que celle-ci, car en fait apache peut aiguiller lui-même les requêtes, grâce au `mod_rewrite`, et tout le code peut être écrit de manière procédurale.

En fait, le code objet écrit plus haut montre globalement le principe utilisé dans la plupart des frameworks, ceux-ci s'efforcent en plus de rajouter un vrai plus à la structure : `Zend_Controller_Router_Rewrite`, par exemple, est un composant de routage intégré au Zend Framework, qui va bien plus loin que le `mod_rewrite` d'Apache, en permettant d'écrire à la volée certaines URLs de manière totalement personnalisée, en n'importe quel point du site.

Ce n'est pas le seul ajout qui y est fait. Le `frontController` fonctionne de mèche aussi avec le motif `InterceptingFilter`, qui fait partie du design plus global `ChainOfAction`. Ce motif permet de centraliser toutes les actions qui sont communes à chaque requête : vérification d'identité de la personne, affichage du header du site... Le contrôleur frontal recevant toutes les requêtes, il est facile d'y intégrer des traitements qui se répètent pour toutes les requêtes.

Exemple `interceptingFilter`

```
<?php
class frontController
{
    private $_filters = array();

    public function addFilter(Filterable $filter)
    {
        $this->_filters[] = $filter;
    }

    public function dispatch($defaults = null)
    {
        //...

        foreach($this->_filters AS $filter){
            $filter->preFilter();
        }

        $this->forward($parsed['module'], $parsed['action']);

        foreach(array_reverse($this->_filters) AS $filter){
            $filter->postFilter();
        }
    }
}
```

#### interface Filterable

```
interface Filterable
{
    public function preFilter();
    public function postFilter();
}
```

## IV - Motif actionController

Nous venons de voir le fonctionnement d'un frontController, associé au motif command : Le frontController déclenche la méthode launch() du fichier action trouvé.

Il est possible de factoriser encore un peu plus le traitement, en implémentant les **ActionController**. Ils ont pour but de regrouper toutes les actions faisant référence à une même entité logique sur le site. Une partie de la logique du frontController est alors intégrée dans les actionController, à qui il passe l'objet de requête et l'objet de réponse. Un étage de plus s'est créé, ce qui permet d'y voir réellement plus clair dans notre code et le cheminement de nos actions, on peut aussi intégrer la gestion des codes http avec la gestion des exceptions, etc ...

Ici, nous arrivons à un niveau de difficulté de conception qui impose l'orientation objet du code, afin d'en tirer toute sa puissance. Mais à mesure que la conception se complique, l'utilisation du modèle objet, elle, devient tout de suite simple et claire.

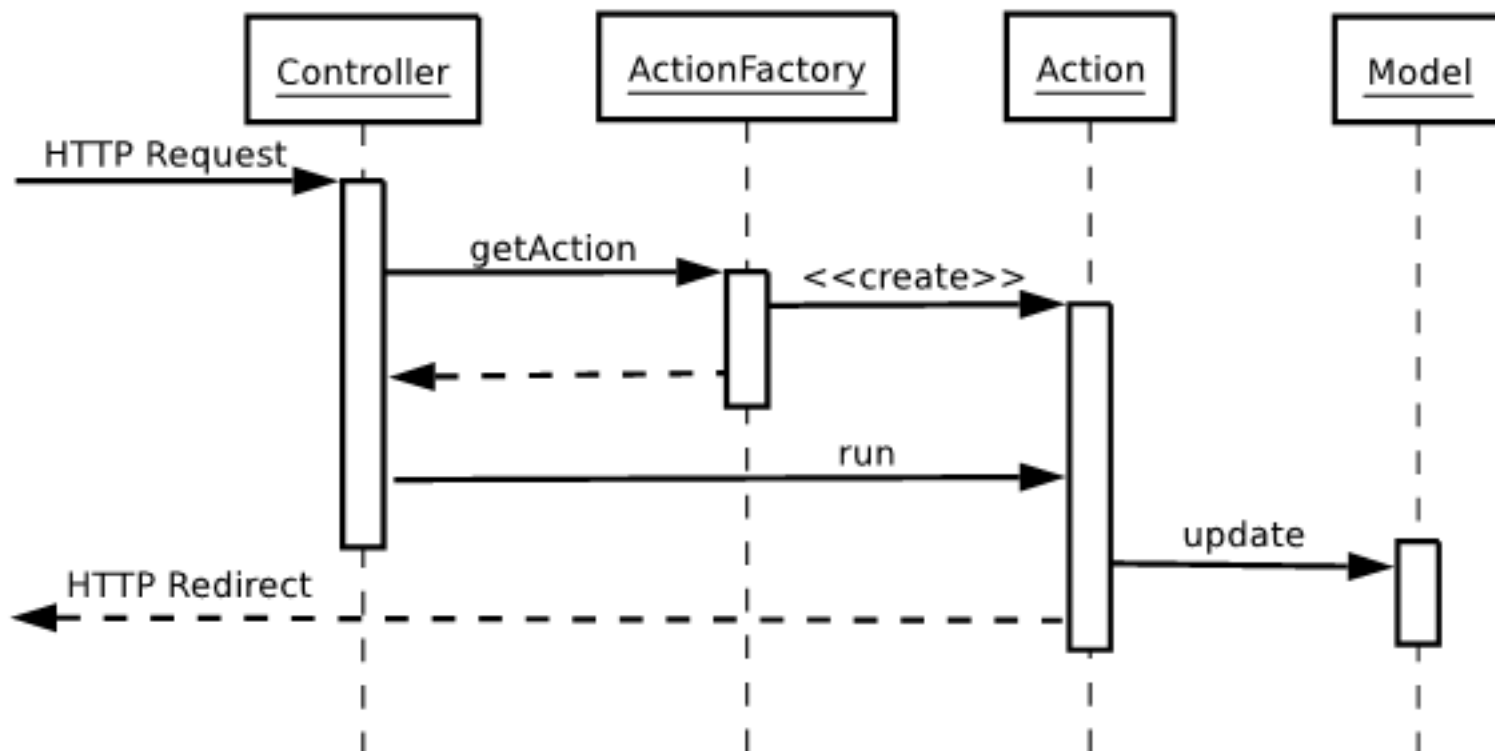


Diagramme de séquence du MVC ActionController

classe Request

```

<?php
class Request
{
    public function getParam($key)
    {
        return filter_var($this->getTaintedParam($key), FILTER_SANITIZE_STRING,
        FILTER_FLAG_NO_ENCODE_QUOTES);
    }

    public function getTaintedParam($key)
    {
        if ($_SERVER['REQUEST_METHOD'] == 'POST' && isset($_POST[$key])){
    
```

**classe Request**

```
    return $_POST[$key];
} else {
    return $_GET[$key];
}
}
```

**class Response**

```
class Response
{
    private $_vars = array();
    private $_headers = array();
    private $_body;

    public function addVar($key, $value)
    {
        $this->_vars[$key] = $value;
    }

    public function getVar($key)
    {
        return $this->_vars[$key];
    }

    public function setVar($key, $value)
    {
        $this->_vars[$key] = $value;
    }

    public function getVars()
    {
        return $this->_vars;
    }

    public function setBody($value)
    {
        $this->_body = $value;
    }

    public function redirect($url, $permanent = false)
    {
        if ($permanent){
            $this->_headers['Status'] = '301 Moved Permanently';
        } else {
            $this->_headers['Status'] = '302 Found';
        }
        $this->_headers['location'] = $url;
    }

    public function printOut()
    {
        foreach ($this->_headers as $key => $value) {
            header($key . ':' . $value);
        }
        echo $this->_body;
    }
}
```

**classe frontController**

```
abstract class frontController
{
    public static function dispatch()
    {

```

## classe frontController

```
try{
    $request = new Request();
    $response = new Response();
    ActionController::process($request, $response) -> printOut();
} catch (Exception $e){
    ActionController::processException($request, $response, $e) -> printOut();
}
}
```

## classe View

```
class View
{
    public function render($file, $assigns = array())
    {
        extract($assigns);
        ob_start();
        require($file);
        $str = ob_get_contents();
        ob_end_clean();
        return $str;
    }
}
```

## classe ActionController

```
class ActionController
{
    protected $_request;
    protected $_response;
    protected $_redirected;

    public static function process(Request $request, Response $response)
    {
        if (!file_exists($path = 'controllers/' . $request->getParam('controller') . '.php')){
            throw contrôleurIntrouvableException ('contrôleur introuvable');
        }
        require_once($path);
        $class = $request->getParam('controller') . 'Controller';
        $controller = new $class($request, $response);
        return $controller->launch();
    }

    public static function processException(Request $request, Response $response, $e)
    {
        $controller = new ActionController($request, $response);
        return $controller->launchException($e);
    }

    public function __construct(Request $request, Response $response)
    {
        $this->_request = $request;
        $this->_response = $response;
        $this->_redirected = false;
    }

    private function _actionExists($action)
    {
        try{
            $method = new ReflectionMethod(get_class($this), $action);
            return ($method->isPublic() && !$method->isConstructor());
        } catch (Exception $e){
            return false;
        }
    }
}
```

## classe ActionController

```
}

public function redirect($url)
{
    if ($this->_redirected == true){
        throw Exception('Une redirection a déjà été demandée');
    }
    $this->_response->redirect($url);
    $this->_redirected = true;
}

private function _render($file)
{
    $view = new View();
    $this->_response->setBody($view->render(dirname(__FILE__) . '/views/' . $file,
    $this->_response->getVars()));
}

public function __get($param)
{
    return $this->_response->getVar($param);
}

public function __set($name,$param) {
    $this->_response->setVar($name, $param);
}

public function launch()
{
    $action = $this->_request->getParam('action');
    if (!$this->_actionExists($action)){
        throw ActionIntrouvableException('Action introuvable');
    }
    // prefiltering

    $this->$action();

    // postfiltering

    if (!$this->_redirected){
        $this->_render($this->_request->getParam('action') . '.php');
    }
    return $this->_response;
}

public function launchException(Exception $e)
{
    if ($e instanceof MVCException){
        $this->_render('404.php');
    }else{
        $this->_render('500.php');
    }
    return $this->_response;
}
}
```

## classes d'Exception

```
class ActionIntrouvableException extends MVCException { }
class contrôleurIntrouvableException extends MVCException { }
class MVCException extends Exception { }
```

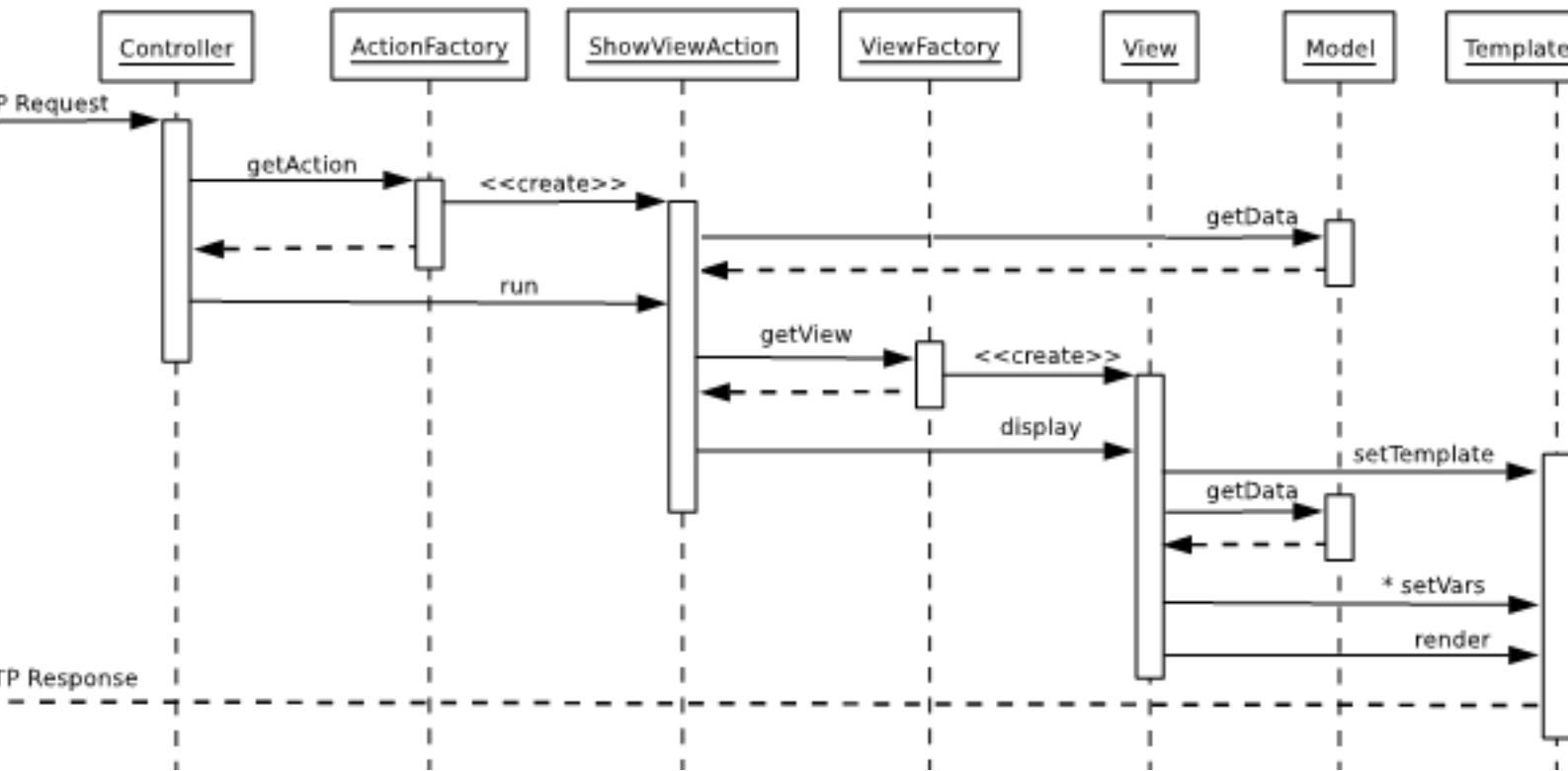
## index (main)

```
$front = frontController::dispatch();
```

Ca se complique encore, on voit à première vue que le rôle du frontController est réduit à un simple bloc try - catch. Une grosse partie de la logique de controle a été déportée dans l'ActionController, dont tous les contrôleurs applicatifs hériteront.

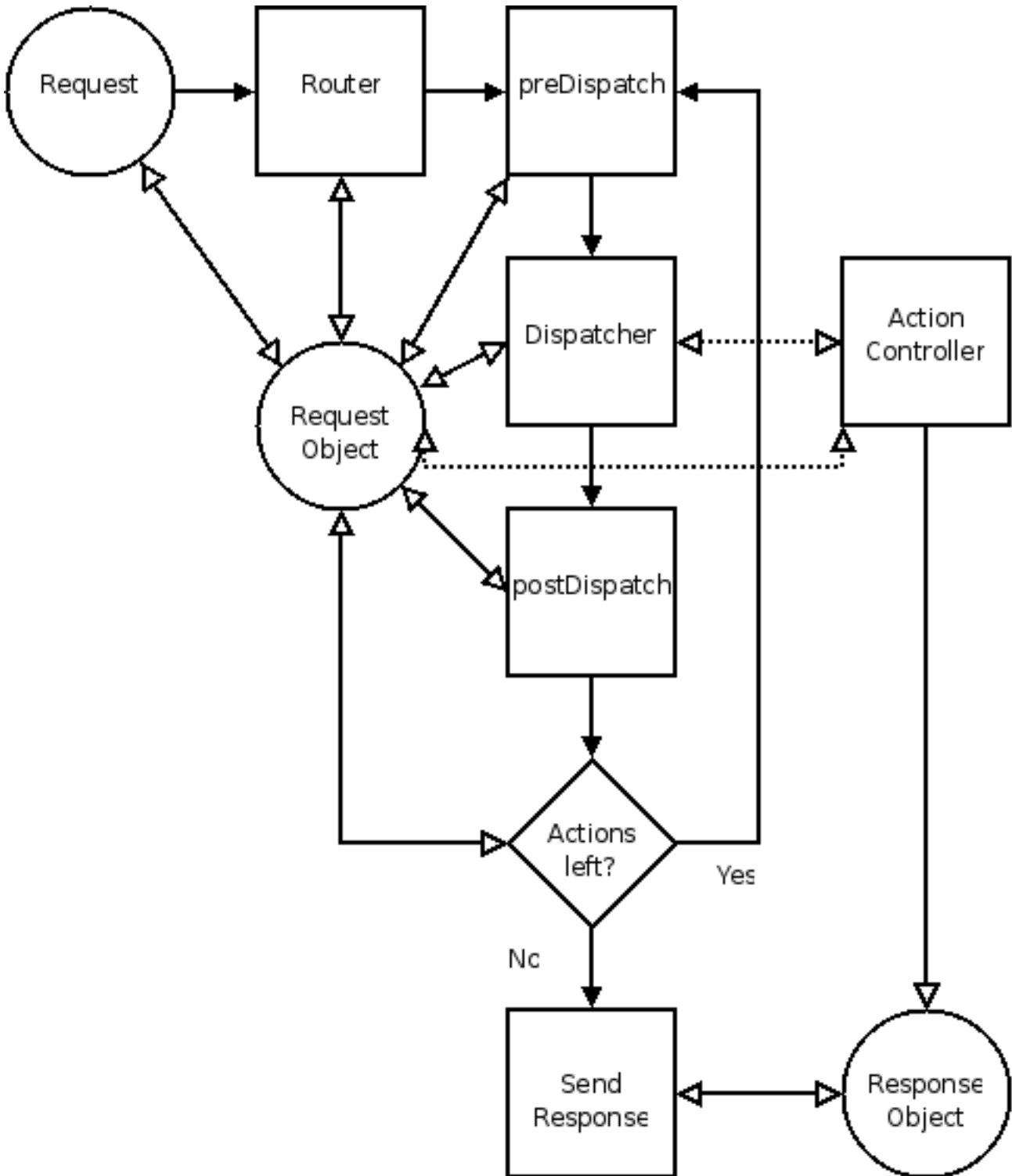
La partie vue est simplifiée ici. Grace aux méthodes magiques \_\_get et \_\_set, le contrôleur peut assigner directement une valeur à la vue via \$this->unePropriete. C'est un système que l'on retrouve souvent.

Souvent aussi, la gestion des vues est un peu plus complexe, avec des Factory et Composite, on peut alors arriver à un **diagramme de séquence** qui ressemble à ça :



MVC actionController + ViewFactory

A titre informatif, on peut regarder le diagramme de flux du modèle MVC du Zend Framework :



Zend Framework MVC



## V - Conclusions

Nous venons de voir un petit bout du motif MVC, notamment la partie contrôleur en liaison avec beaucoup de Design Patterns connus. Pour un site rapide, petit, ou n'étant pas amené à évoluer rapidement, un tel motif peut s'avérer lourd en gestion. D'autant plus qu'Apache + PHP permettent déjà pas mal de liberté.

Sur une application plus complexe, en revanche, développer au sein d'un Framework tel que Zend Framework, Symfony, ou CakePHP ; permettra de produire du code plus clair, de séparer chaque rôle de votre application, de programmer à plusieurs (les règles étant communes) et de faire évoluer très simplement le code, pour peu que celui-ci ait été pensé convenablement à ses débuts.

Ceci permet aussi de garder en tête le cheminement des requêtes, de bosser avec des diagrammes, de simplifier indirectement la gestion de projet.

Alors que PHP était à l'origine conçu comme un langage "glue", le modèle objet de PHP5 allié à la puissance interne de PHP montre bien que celui-ci est autant capable que Java, .NET, Ruby ou autre, en terme de conception et d'architecture logicielle.

Il faut aussi savoir manger à toutes les gamelles, je vous conseille de vous pencher sur Java  **EE**, et ses frameworks MVC, comme  **Struts** par exemple, ou Rails pour Ruby, vous y apprendrez beaucoup de choses.

### [Programmation web/PHP et architecture MVC](#)

### [Implémentation du pattern MVC \(Java\)](#)

### [Présentation de tout un tas de Design Pattern \(exemples en Java\)](#)

### [La rubrique ruby pourrait vous être utile aussi](#)

### [Notre rubrique conception](#)

### [Notre espace Zend Framework](#)

