

L'article provient du magazine PHP
Solutions.

<http://www.phpsolmag.org/fr>

La copie et la distribution gratuites de l'article
sont permise sa condition que
sa forme et son contenu original soient conservés.

Frameworks pour PHP, comment créer efficacement les applications Web ?

Pawel Kozlowski

L'idée de préparer les squelettes des applications vient du développement propre au programmeur. Il est connu que le code d'une application donnée ne constitue qu'une partie de programme. Tout le reste se répète et peut être utilisé plusieurs fois dans d'autres applications.

Quand vous écrivez plusieurs fois les mêmes applications, vous élaborer des méthodes spécifiques d'action. Vous avez un ensemble de bibliothèques favorites, une manière de diviser les scripts en sous-répertoires et en un format d'enregistrement de données de configuration. Quand vous commencez un nouveau projet, vous vous tournez par habitude vers des éléments vérifiés.

Vous les organisez pour créer une nouvelle application. C'est ainsi que se créent les squelettes de programme, appelés cadres (en anglais *frameworks*). Dans un projet concret, il suffit de les remplir d'un code spécifique pour une tâche donnée.

Les cadres ne sont pas des bibliothèques individuelles mais des ensembles des fragments de code bien choisis et qui travaillent entre eux. Ils constituent le corps de l'application et contiennent les sous-systèmes, responsables p. ex. de se connecter à la base de données, de véri-

fier les droits, de tracer les événements, de gérer les erreurs, etc.

Quelles caractéristiques des cadres font en sorte qu'ils sont couramment utilisés ? Les programmeurs apprécient notamment :

- La possibilité de se libérer du travail dur répétitif présent dans chaque projet.
- La minimalisation du risque de présence d'erreurs, qui résulte de la

Sur le réseau



1. <http://www.phppatterns.com> – la meilleure source d'infos sur frameworks
2. wact.sourceforge.net – page d'accueil de WACT
3. <http://java.sun.com/developer/technicalArticles/J2EE/patterns/>
4. <http://www.martinfowler.com/eaCatalog/>

Sur le CD



Le CD contient tous les scripts analysés ainsi que quelques cadres populaires : WACT, Mojavi, PRADO, Seagull.

Ce qu'il faut savoir ...

Vous devriez connaître les bases de PHP. Une expérience de réalisations des applications Web serait utile pour comprendre l'article.

Cet article explique ...

Dans l'article, nous expliquerons pas à pas comment créer son propre Framework. Cette solution se reposera sur l'expérience de plusieurs années de travail sur la réalisation des squelettes des applications des programmeurs professionnels.

limitation du nombre de tâches répétitives et le fait d'éviter la répétition du code.

- La possibilité de se servir du savoir et des expériences des concepteurs, rassemblés durant des années de travail sur un cadre donné. Vous pouvez vous attendre à ce que les solutions utilisées s'avèrent les meilleures possibles grâce à la pratique de plusieurs années.
- L'organisation et la systématisation du projet. Grâce à l'utilisation des cadres, il n'y a qu'une seule structure de répertoires, une structure type de fichiers de configuration, etc.
- La possibilité de systématiser la terminologie utilisée par les programmeurs qui travaillent sur un projet donné, ce qui facilitera la communication.
- La facilité d'identifier les manques dans le framework et d'y ajouter de nouvelles fonctionnalités.

Il existe pour PHP un nombre assez grand d'implémentations prêtes à l'emploi. D'autres se créent en se servant de l'existence d'un nouveau modèle orienté objets PHP5. Les concepteurs ont offert beaucoup de cadres (p. ex. sur Internet) à l'ensemble de programmeurs.

Tache sur l'image idyllique

Quand vous décidez de choisir une solution concrète, vous devez être conscients des dangers résultant de l'utilisation des cadres. Le temps consacré pour connaître les fonctionnalités et la philosophie de leur fonctionnement constitue un problème essentiel. Il faut comprendre que le concepteur du produit imaginait la configuration de l'application, le transfert de contrôle (p. ex. de la validation du formulaire à son enregistrement dans une base) ou les questions de droits. Il est clair que chacun d'entre nous a ses habitudes et la vision du concepteur d'un cadre déterminé peut être différente de notre pratique. Dans un tel cas, la meilleure solution consisterait à tester le cadre donné ou à écrire notre propre cadre.

MVC

Vous savez donc déjà ce que sont les cadres, où ils peuvent être utiles et quels problèmes vous pouvez rencontrer en les

utilisant pour réaliser vos applications. Si vous vous demandez comment est construit un cadre : de nombreuses personnes au monde y ont réfléchi et y réfléchissent encore. Les résultats de nombreuses tentatives, erreurs et réflexions théoriques sont des modèles qui décrivent la façon de réaliser des applications d'un type donné. Ce ne sont pas des bibliothèques prêtes mais les principes et les conseils à partir d'expériences pratiques.

Quand nous réalisons une architecture générale d'applications Web complexes, nous nous servons le plus souvent d'un modèle architectonique MVC (en anglais *Model View Controller*). Les directives de ce modèle informent comment diviser une application en trois couches logiques, liées entre elles : modèle, vue et contrôleur. L'avantage le plus important de l'utilisation de MVC consiste à organiser la réalisation de l'application et à séparer complètement la présentation (vue) des données (modèle) et de la manière de leur enregistrement (contrôleur).

Regardons de près chaque couche MVC :

- modèle (en anglais *model*) – couche d'objets d'affaires et d'accès aux données qui créent ces objets. Le modèle, ce sont des objets liés uniquement au domaine pour lequel vous créez l'application,

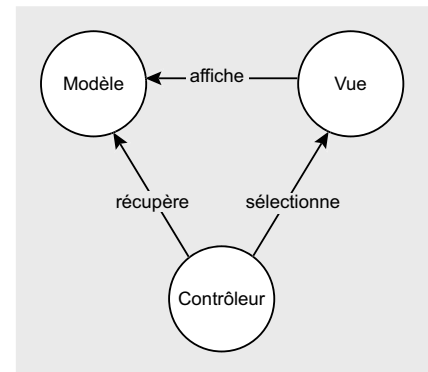


Figure 1. Couches MVC et leurs liens

- vue (en anglais *view*) – définit comment présenter les objets de la couche modèle et les éléments de l'interface utilisateur. Elle est chargée de bien formater les données de modèle selon la présentation (p. ex. navigateur Web ou PDF),
- contrôleur (en anglais *controller*) – est un « dirigeant » qui, en réponse aux actions de l'utilisateur, récupère les objets du modèle et les transmet à une vue sélectionnée.

Le plus grand avantage de l'utilisation de MVC consiste à séparer complètement les données (et les règles qui les gèrent) de leur présentation. Grâce à cette démarche, il est possible de présenter les mêmes objets (p. ex. liste d'actualités) dans le navigateur Web comme PDF ou

Listing 1. Fragment du script avec les couches MVC confondues

```

<?php
// connexion à la DB
$link = mysql_connect("localhost", "root", "")
    or die("Impossible de se connecter à DB : " . mysql_error());
mysql_select_db('newsdb') or die(" Impossible de sélectionner la DB!");
//télécharger les données
$query = "SELECT * FROM my_news";
$result = mysql_query($query) or die("Erreur de requête : " . mysql_error());
//télécharger et présenter les données
echo "<table border='1'>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
    echo "\t<tr>\n";
    foreach ($line as $col_value) {
        echo "\t\t<td>$col_value</td>\n";
    }
    echo "\t</tr>\n";
}
echo "</table>\n";
//se déconnecter de la DB
mysql_free_result($result);
mysql_close($link);
?>
  
```

p. ex. RSS sans changer la logique liée au téléchargement ou à la transformation de cette liste !

Pour que cette séparation soit possible, les objets de la couche du modèle ne peuvent pas être conscients comment ils seront présentés. C'est la couche de la vue qui peut les afficher car elle sait de quels objets il s'agit. D'un autre côté, la vue ne présente que les données préparées à l'avance : le contrôleur est chargé de traiter la requête et de télécharger les valeurs adéquates. Il est possible de dessiner tous ces liens sous forme de diagramme comme celui de la Figure 1.

Comment se servir en pratique de ce savoir ? Le Listing 1 présente un exemple type de code, basé sur un extrait de la documentation originale PHP.

Faites attention en particulier au fragment du code lié à la boucle `while`. Non seulement vous téléchargez les données concernant les actualités (action sur le modèle) mais vous les affichez aussi (vue). Une section du programme contient les couches de modèle et de vue confondues. Si vous modifiez la façon de présenter, vous devrez aussi modifier la logique de l'application. Si vous vouliez maintenant présenter les mêmes actualités comme RSS (une autre vue), vous devriez copier aussi la boucle entière. Il s'agirait là d'une duplication du code chargé de télécharger les données.

Essayons à présent de résoudre les problèmes identifiés au moyen du modèle MVC et en séparant le modèle de la vue. Le Listing 2 présente la partie séparée, liée à la présentation et aux données. Cette petite modification influence en effet la souplesse de votre application. Il est facile de récupérer l'extrait lié à la vue dans un fichier à part. À ce moment là, vous pourriez inclure (`include`) le script adéquat contenant la description de la présentation en tenant compte de l'outil final où le service est affiché. Le Listing 3 est un exemple d'utilisation des mêmes données pour générer RSS.

Vous avez séparé la vue, vous pouvez passer à présent à la couche du modèle. Dans un premier temps, vous transmettez l'extrait lié au téléchargement et au transfert des données dans des classes séparées (Listing 4). Cette démarche vous permettra d'utiliser les objets du modèle plusieurs fois à des endroits différents de l'application (p. ex. dans la présentation des actualités sur le site Web et dans

le module d'administration). Tant l'accès aux données (classe `NewsModelDao`) que les méthodes d'objet (p. ex. `NewsModel::isValid()`) ont été séparés dans une couche à part. Grâce à cette démarche, le fragment de programme lié à l'interaction avec la DB se trouve à un seul endroit. Vous ne craignez plus les modifications de noms de tables. La même situation concerne les objets de la classe `NewsModel` : si les principes, selon lesquels nous considérons qu'une actualité est disponible, changent, nous devons modifier cette logique seulement dans un seul endroit. Si tous les principes décrits sont codés directement en PHP ou dans les requêtes SQL, il faut parcourir et modifier de nombreux scripts à chaque modification.

La dernière couche MVC non analysée est le contrôleur. En général, les tâches du contrôleur consistent à analyser la requête de l'utilisateur et, en se basant sur cette analyse :

- à récupérer les objets du modèle adéquat,
- à sélectionner une vue adéquate et à y transmettre les objets sélectionnés à l'avance.

Les exemples décrits montrent clairement les avantages de l'utilisation de MVC. Hé-

las, il n'y a point de roses sans épines : l'application est plus souple mais aussi plus complexe.

Comment réaliser votre propre cadre ?

Comme vous l'avez remarqué, le modèle MVC vous donne des bases solides ; en vous y basant, vous pouvez commencer à réaliser votre propre cadre. Puisque les directives de MVC sont assez générales, nous allons présenter en détails chaque couche et les remplir d'un vrai code fonctionnant. Tous les exemples présentés sont écrits en PHP5.

Contrôleur

Si vous séparez une couche de modèle et de vue du programme sur le Listing 4 dans les fichiers séparés, il vous restera un script pour vous connecter à la base de données, récupérer le modèle et sélectionner la vue. Même avec un simple contrôleur, il est possible de remarquer deux particularités. Il ne supporte qu'une seule action logique (ici : téléchargement de la liste d'actualités) mais il contient également le code à réaliser pour chaque action. Sur le Listing 4, c'est une connexion à la base de données mais il est facile de citer d'autres exemples : authentification, ouverture de session, journal d'événements, etc. Vous allez donc séparer les parties répétitives du code dans une

Listing 2. Séparation de la vue du script du Listing 1

```
<?php
...
// connexion à la DB et
// téléchargement des données -
// comme sur le Listing 1
$result_arr = array();
while ($line = mysql_fetch_array
($result, MYSQL_ASSOC)){
    $result_arr[] = $line;
}
...
// déconnexion de la DB
...
?>
<table border='1'>
<?php foreach ($result_arr
as $line) { ?>
<tr>
<?php foreach ($line
as $col_value) { ?>
<td><?php echo $col_value; ?>
</td>
<?php }?>
</tr>
<?php } ?>
</table>
```

Listing 3. Les mêmes données que sur le Listing 2 – autre présentation

```
<?php
include("class.myRSS.php");
$myRSS = new myRSS;
$myRSS->channelTitle =
"Framework news";
$myRSS->channelLink = "http://
www.phpsolmag.org";
$myRSS->channelDesc =
"Framework news";
foreach ($result_arr as $news)
{
    $myItem = new myRSSItem();
    $myItem->link =
'http://www.phpsolmag.org/news/'
. $news['news_id'];
    $myItem->title =
    $news['news_title'];
    $myItem->description =
    $news['news_text'];
    $myRSS->addItem();
}
header("Content-type: text/xml");
echo $myRSS->getOutput();
?>
```

partie à part du contrôleur. La séparation que vous venez d'effectuer est très souvent réalisée lors de la création de cadres et le bloc commun du contrôleur a même un nom particulier : *Front Controller*.

Front Controller supporte toutes les requêtes dirigées à l'application et réalise les opérations communes de toutes les actions. Dans l'étape suivante, il trouve un contrôleur caractéristique pour l'action donnée et lui transmet le contrôle.

Les stratégies, avec lesquelles les contrôleurs d'action sont trouvés et exécutés, peuvent être diverses. Dans un premier temps, vous allez en utiliser la plus simple : chaque action sera enregistrée dans un fichier séparé, appelé *[nom_de_l'action].action.php*. L'utilisateur décide quelle action appeler, en donnant à Front Controller un paramètre, p. ex. *http://[URL de Front Controller]?action=[nom_de_l'action]*.

Le fonctionnement du contrôleur doit donner comme résultat le nom de la vue et les objets de modèle préparés. Vous formalisez les valeurs retournées en les compressant dans une classe appelée *ModelAndView*.

Vous avez vu beaucoup de nouvelles notions, retournez à présent au code PHP pour observer comment l'ensemble fonctionne dans la nouvelle application. Le Listing 5 définit la classe *ModelAndView*. Le code suivant du programme, présenté sur les Listings 6 et 7, contient les classes reconstruites du contrôleur d'action et de Front Controller.

Bien évidemment, l'application peut contenir plusieurs Front Controller. Il est possible d'utiliser une telle solution quand les différents ensembles d'opérations sont à effectuer avant de lancer le contrôleur adéquat. Notre module d'actualités, doté de la partie administrative, est un bon exemple d'une telle situation. Nous avons ainsi deux Front Controller : l'un qui gère la présentation et l'autre qui gère l'administration avec l'authentification.

Vue

Remarquez que le contrôleur active un fichier concret contenant la vue. Cette solution peut être acceptée quand il n'y a qu'un seul type de présentation. Hélas, s'il y a beaucoup de type d'affichage, ce n'est pas la meilleure solution : il faudrait dans ce cas placer toutes les informations sur plusieurs types de présentation dans un seul fichier de la vue.

Un autre problème de la couche de vue est lié au format d'enregistrement du code HTML et de l'information dynamique à travers un navigateur Web. Est-ce que la manière d'enregistrer présentée vous

rappelle quelque chose ? Oui, c'est tout simplement un template.

Essayons de résoudre les deux problèmes décrits. La première étape consiste à ajouter un système de templates

Listing 4. Séparation de la couche du modèle et du contrôleur

```
<?php
// MODÈLE
class NewsModel {
    public function __construct($data_array) {
        foreach ($data_array as $k => $v) {
            $this->{$k} = $v;
        }
    }
    public function isValid() {
        if ((int)$this->news_valid) {
            if (($this->news_validfrom == '0000-00-00 00:00:00') &&
                ($this->news_validto == '0000-00-00 00:00:00')) {
                return true;
            }
            elseif (($this->news_validfrom == '0000-00-00 00:00:00') &&
                (time() < strtotime($this->news_validto))) {
                return true;
            }
            elseif (($this->news_validto == '0000-00-00 00:00:00') &&
                (time() > strtotime($this->news_validfrom))) {
                return true;
            }
            elseif ((time() > strtotime($this->news_validfrom)) &&
                (time() < strtotime($this->news_validto))) {
                return true;
            }
            else {
                return false;
            }
        }
        else {
            return false;
        }
    }
}

class NewsModelDao {
    public function findAllNews() {
        $query = "SELECT * FROM my_news";
        $result = mysql_query($query)
            or die("Erreur de requête : " . mysql_error());
        $result_arr = array();
        while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
            $result_arr[] = new NewsModel($line);
        }
        mysql_free_result($result);
        return $result_arr;
    }
}

// CONTRÔLEUR
// connexion à la DB
$link = mysql_connect("localhost", "root", "");
    or die("Impossible de se connecter à la DB : " . mysql_error());
mysql_select_db('newsdb') or die("Impossible de sélectionner la DB !");
$news_dao = new NewsModelDao();
$result_arr = $news_dao->findAllNews();
// déconnexion de la DB
mysql_close($link);
// VUE
include('news_list_view.php');
?>
```

Smarty au cadre créé. Grâce à la modification de la structure du contrôleur d'action et à l'ajout de la classe `ModelAndView`, vous pourrez mieux organiser le transfert des informations concernant la vue.

Modèle

La couche modèle pourrait également être améliorée. Tout d'abord, présentons brièvement la classe `NewsModelDao`. Son nom n'a pas été choisi au hasard car le sigle DAO vient d'un autre modèle de projet : en anglais *Data Access Object*. Vous vous rappelez sûrement que le cadre est une solution commune à des problèmes de système qui se répètent souvent. Cette solution repose sur les meilleures pratiques et les expériences de longue date. C'est pour cette raison, les différents modèles architecturaux de projet apparaissent aussi souvent dans les descriptions de cadres. L'un de ces modèles est DAO. Il décrit comment le mode d'accès la source constante des données (il s'agit le plus souvent de la base de données). Conformément au modèle DAO, la logique entière de l'accès aux données externes d'un type concret se trouve dans une classe. Une telle approche vous donne d'énormes avantages. Premièrement, les objets qui se servent de DAO sont indifférents à la provenance de données. Il est donc possible de modifier l'emplacement des informations sans toucher

aux autres fragments du code. Imaginez la situation où on vous demande de faire en sorte que désormais l'application créée ne récupère pas les données concernant les utilisateurs dans la base de données mais dans le webservice. Si le code d'accès aux données sur les utilisateurs se trouve dans plusieurs scripts, vous avez beaucoup de travail qui consiste à copier et à tester la solution. Si vous vous servez de DAO, cela sera plus facile.

DAO est la seule partie de l'application qui contient le code lié à un emplacement concret de données. Grâce à ceci, toutes les modifications liées au médium physique se trouvent dans une seule classe et il est particulièrement facile d'en modifier les noms de tables ou de colonnes. De plus, il est possible dans DAO de masquer les différences entre les bases de données en se servant de la couche d'abstraction d'accès à la base, p. ex. AdoDB.

Configuration et gestion d'erreurs

Avant de présenter un exemple concret de code supportant Smarty et AdoDB, organisons la structure de notre application. Tout d'abord, vous déterminez la structure de répertoires où tous les fichiers se trouveront. La Figure 2 présente une structure de répertoires proposée ; les catalogues contiennent :

- *[catalogue principal]* – fichier *index.php* – Front Controller,
- *actions* – classes d'action (contrôleurs),
- *model* – classes de modèle (doté d'un répertoire séparé pour les classes DAO),
- *templates* – templates Smarty,
- *conf* – fichiers de configuration,
- *lib* – bibliothèques de cadre et bibliothèques d'aide,
- *var* – répertoire pour enregistrer les fichiers temporaires du niveau du serveur Web.

Vous placez le fichier de configuration *index.conf.php* dans le répertoire *conf*. Il est possible d'y déterminer les paramètres qui contrôlent le fonctionnement de tous les modules.

Quand vous ajoutez le support de Smarty et AdoDB, vous modifiez aussi le modèle de réaction aux erreurs. PHP5 est enfin doté du mécanisme de gestion d'événements, utilisons le.

Le Listing 8 présente Front Controller après toutes les modifications décrites.

Puisque AdoDB a été ajouté, la classe DAO doit également être modifiée. C'est elle qui contient les informations concernant l'accès physique à la source externe de données. Heureusement, la modification est simple (Listing 9).

Listing 5. Classe `ModelAndView`

```
<?php
class ModelAndView{
    private $_model;
    private $_view;
    public function getModel(){
        return $this->_model;
    }
    public function setModel(
        $model_arr){
        $this->_model = $model_arr;
    }
    public function addToModel(
        $arr_key, $model_value){
        $this->_model[$arr_key]=
            $model_value;
    }
    public function setView($view){
        $this->_view = $view;
    }
    public function getView(){
        return $this->_view;
    }
}
?>
```

Listing 6. Front Controller activant le contrôleur d'action

```
<?php
require_once('ModelAndView.class.php');
$link = mysql_connect("localhost", "root", "");
    or die("Impossible de se connecter à la DB : " . mysql_error());
mysql_select_db('newsdb') or die(" Impossible de sélectionner la DB!");
$action_to_run = $_GET['action'];
if ($action_to_run != ''){
    $action_file_name = $action_to_run.'.action.php';
    if (file_exists($action_file_name)){
        require_once($action_file_name);
        $actionclassname = $action_to_run.'action';
        $actioncontroller = new $actionclassname();
        $smv = $actioncontroller->processRequest();
    }
    else{
        die("Aucune action à lancer !");
    }
}
else{
    die("Aucun paramètre donné pour trouver l'action !");
}
mysql_close($link);
$result_arr = $smv->getModel();
include($smv->getView());
?>
```

Le contrôleur pour chaque action aura aussi une autre structure (Listing 10).

Couche des services système

Une application type comprend des services système qu'il est impossible d'attribuer seulement à une des couches MVC. Un bon exemple d'un tel service est une création d'enregistrements dans un journal d'événements (en anglais *Logging*). Nous voudrions le plus souvent suivre ce qui se passe aussi bien dans la couche du contrôleur (p. ex. quelles actions sont appelées) que du modèle (p. ex. quelles requêtes sont envoyées dans la base de données). Un autre exemple peut être le support cache, l'envoi des courriels ou la vérification des droits. Tous les services peuvent apparaître plusieurs fois dans le modèle ou dans le contrôleur pratiquement dans chaque application Web. Ils sont des candidats parfaits pour une intégration dans le cadre.

Regardez le Listing 10. L'exemple du service DBManager s'y trouve ; il est chargé de se connecter à la base de données. Remarquez que la classe DBManager est statique. Vous voulez avoir l'accès à la base à partir de différents endroits du cadre et une seule connexion. Nous avons déjà présenté l'utilisation de la classe DBManager dans le constructeur DAO (Listing 10).

Les services système, se trouvant sur plusieurs couches, produisent un certain désordre dans l'architecture du systè-

me. Vous voulez y avoir un accès facile aux plusieurs extraits du code et en même temps, vous avez besoin le plus souvent d'une seule instance de l'objet donné. Dans de tels cas, une solution souvent utilisée, mais à éviter, consisterait à utiliser les variables globales.

Retournez à l'exemple de connexions à la base de données : vous pourriez ouvrir les connexions dans *Front Controller*, enregistrer dans la variable globale où vous iriez en cas de besoin. Hélas, l'utilisation des variables globales dans ce cas présente deux inconvénients importants. Premièrement, cela signifie la nécessité d'activer la bibliothèque AdoDB et la connexion permanente à la base, même pour les actions qui n'ont pas besoin d'avoir accès à la DB (p. ex. formulaire d'ajout d'une nouvelle actualité). Deuxièmement, votre code commence à dépendre beaucoup de la variable globale ; si vous vouliez utiliser votre objet DAO dans un autre projet, vous ne devriez pas oublier alors de créer toutes les variables globales nécessaires.

Réfléchissons sur ce qui se passerait si vous vouliez utiliser DAO dans un autre projet qui gère lui-même les connexions à la DB ? Bien évidemment, vous devriez doubler cette connexion dans la variable globale.

Même si l'utilisation d'un service système statique du type DBManager ne vous protège pas contre tous les inconvénients liés aux variables globales, elle les rend moins gênants. Il est facile de modifier un simple code de DBManager en intégrant ainsi votre DAO à n'importe quelle stratégie de connexion à la base. À cette occasion, vous avez obtenu l'effet appelé *lazy include*, grâce auquel la bibliothèque nécessaire est activée et initialisée uniquement quand elle est vraiment nécessaire.

Tâches ennuyeuses encore plus faciles

Vous avez construit une architecture solide. Votre cadre peut constituer déjà un squelette assez stable pour les futures

Listing 7. Contrôleur d'action

```
<?php
require_once
    ('newsmodeldao.class.php');
require_once
    ('newsmodel.class.php');
class newslistaction{
    private $_newsmodeldao;
    public function __construct(){
        $this->_newsmodeldao =
            new NewsModelDao();
    }
    public function processRequest(){
        $result_arr =
            $this->_newsmodeldao->
                findAllNews();
        $mv = new ModelAndView();
        $mv->setModel($result_arr);
        $mv->setView
            ('news_list.view.php');
        return $mv;
    }
}
?>
```

Listing 8. Front Controller après l'ajout de gestion d'événements

```
<?php
require_once('conf/index.conf.php');
require_once('lib/core/ModelAndView.class.php');
try{
    $action_to_run = $_GET['action'];
    if ($action_to_run != ''){
        $action_file_name = DIR_ACTIONS.'/'.$action_to_run.'.action.php';
        if (file_exists($action_file_name))
        {
            require_once($action_file_name);
            $actionclassname = $action_to_run.'action';
            $actioncontroller = new $actionclassname();
            $smv = $actioncontroller->processRequest();
            if($smv!=null){
                require_once(SMARTY_DIR_LIB.'/Smarty.class.php');
                $smarty = new Smarty();
                $smarty->tempalte_dir = SMARTY_DIR_TEMPALTES;
                $smarty->compile_dir = SMARTY_DIR_TEMPALTES_C;
                $smarty->debugging = SMARTY_DEBUG;
                $smarty->assign($smv->getModel());
                $smarty->display($smv->getView());
            }
        }
        else {
            throw new Exception("Aucun fichier d'action'$action_file_name'
                à lancer !");
        }
    }
    else{
        throw new Exception("Aucune action à lancer !");
    }
}
catch (Exception $e){
    print_r($e);
}
?>
```

applications. Il y a une répartition des répertoires, une configuration centralisée, les couches du programme séparées (contrôleurs, modèle doté de DAO, vue dotée du système de templates) et les services système (gestion de connexions à la DB). Tous ces éléments vous permettent de réfléchir tranquillement sur la structure des services Web les plus développés. Vous savez où exactement ajouter les nouvelles fonctionnalités. Grâce à ce point, le projet restera toujours sous votre contrôle.

Les bonnes bases, c'est l'essentiel mais rien ne vous empêche de vous faciliter le travail. Regardez encore une fois chaque couche du point de vue de l'amélioration et de la réduction du processus de la réalisation de l'application.

Modèle

Il est difficile d'améliorer l'écriture du code des objets métier à proprement parler. Ce code est caractéristique pour chaque projet : la fonctionnalité que vous créez. Néanmoins, il est possible d'améliorer certaines choses dans ce point. Retournez au Listing 4 et regardez plus précisément le constructeur de la classe `NewsModel`. Vous voyez que tous les champs de l'objet sont donnés en tant que tableau associatif. Un tel constructeur est très pratique pour récupérer les objets du modèle doté de DAO. Créez donc une classe de la base pour tous les objets d'affaires

Listing 9. DAO doté de gestion de ADOdb

```
<?php
class NewsModelDao{
    private $_connection;
    public function __construct
        ($connection){
        $this->_connection =
            $connection;
    }
    public function findAllNews(){
        $query = "SELECT * FROM
            my_news";
        $result = $this->_connection->
            Execute($query);
        $result_arr = array();
        while ($line = $result->
            FetchRow()) {
            $result_arr[] =
                new NewsModel($line);
        }
        return $result_arr;
    }
}
```

contenant un tel constructeur par défaut. Le Listing 11 présente les classes : `BaseMode` et une classe modifiée `NewsModel`.

Lors du travail sur une application développée, vous passez sûrement beaucoup de temps à créer les classes DAO. Cette activité devient très déprimante à partir du moment où vous remarquez que les classes DAO se ressemblent beaucoup. Rien d'étonnant : DAO est chargé d'englober les opérations appelées CRUD (en anglais *Create, Read, Update, Delete*), donc de créer, de récupérer, d'actualiser et de supprimer les objets. Dans le cas des classes métier plus complexes, la lecture peut être compliquée mais l'actualisation et la suppression se ressemblent beaucoup. C'est la réalisation d'une requête SQL. Seuls les noms de tables et les champs d'objet changent.

Personne n'aime écrire toujours le même code ; pour cette raison, depuis un certain temps apparaissent les bibliothèques grâce auxquelles il est possible d'oublier tous ces détails désagréables liés à SQL. Ces bibliothèques et outils mappent les objets dans les tableaux dans une base de données relationnelle. Ils sont définis comme les outils ORM (en anglais *Object Relational Mapping*). Ils sont trop complexes pour les présenter dans cet article. Nous vous encourageons toutefois à vous intéresser aux

Listing 10. Contrôleur d'action

```
<?php
require_once(DIR_MODELDDAO.
    '/newsmodeldao.class.php');
require_once(DIR_MODEL.
    '/newsmodel.class.php');
class newslistaction{
    private $_newsmodeldao;
    public function __construct(){
        $dbconn = DBManager::
            getConnection();
        $this->_newsmodeldao =
            new NewsModelDao($dbconn);
    }
    public function
        processRequest(){
        $result_arr['news_list'] =
            $this->_newsmodeldao->
                findAllNews();
        $mv = new ModelAndView();
        $mv->setModel($result_arr);
        $mv->setView
            ('news_list.tpl.html');
        return $mv;
    }
}
```

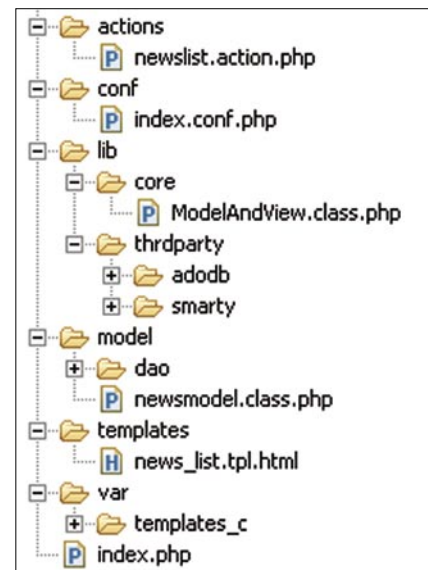


Figure 2. Répertoire dans une application basée sur le cadre

projets tels que Propel ou Pear::DB_DataObject. Si vous vous en servez dans un framework, le temps d'écriture des objets DAO peut se réduire considérablement car il n'est plus nécessaire de coller chaque requête SQL.

Vue

Jusqu'à présent, un exemple d'action ne contenait que la présentation de la liste d'actualités mais une application réelle consiste également à gérer de nombreux formulaires. Une action qui gère un tel formulaire comprend le plus souvent les étapes suivantes :

- validation,
- transformation de données de `$_GET` et `$_POST` en objets d'affaires,
- enregistrement/actualisation de l'objet d'affaires.

Le contrôleur est chargé de l'étape de validation (mis à part l'utilisation de JavaScript qui se trouve bien dans la couche

Listing 11. Classe de la base pour les objets d'affaires

```
class BaseModel{
    public function __construct
        ($data_array){
        foreach ($data_array
            as $k => $v){
            $this->$k = $v;
        }
    }
}
```

de la vue) ; quant à la couche de modèle (plus précisément DAO), elle est chargée d'enregistrer les objets ; il est donc recommandé de s'occuper de créer la vue de formulaires pour que la transformation des données envoyées en objets soit la plus facile possible. Le nom de chaque champ de formulaire a la forme suivante :

name='[nom_de_la_classe][nom_du_champ]' où :

- nom_de_la_classe – nom de la classe métier à laquelle appartient le champ donné,
- nom_du_champ – champ de la classe.

Quand le formulaire a une telle structure, la tâche du contrôleur ne consiste qu'à transmettre la variable `$_POST['nom_de_la_classe']` au constructeur de l'objet d'affaires, p. ex. :

```
$bizobj = new NewsModel
($_POST['NewsModel']);
```

La validation est particulièrement importante car les données du formulaire (arrivées directement de l'utilisateur) arrivent dans la couche du modèle. Il faut être particulièrement attentif lors de l'utilisation de ces solutions dans la partie

Listing 12. MultiActionController

```
...
class newsMultiActionController
{
    private $_newsmodeldao;
    public function __construct(){
        $dbconn = DBManager::
            getConnection();
        $this->_newsmodeldao =
            new NewsModelDao($dbconn);
    }
    public function list_action(){
        $result_arr['news_list'] =
            $this->_newsmodeldao->
                findAllNews();
        $mv = new ModelAndView();
        $mv->setModel($result_arr);
        $mv->setView
            ('news_list.tpl.html');
        return $mv;
    }
    public function addform_action(){
        $mv = new ModelAndView();
        $mv->setView
            ('news_addform.tpl.html');
        return $mv;
    }
}
```

du serveur accessible partout. Quand au module d'administration, c'est une approche parfaite.

De même qu'écrire les classes DAO est ennuyant, la réalisation des formulaires n'appartient pas non plus aux activités les plus fascinantes. Mis à part tous les aspects liés à HTML, les formulaires se ressemblent : ils contiennent souvent les champs présents dans la base et dans les objets d'affaires. Pour PHP, il existe des packages (p. ex. PEAR::DB_DataObject_FormBuilder) qui automatisent le processus de génération des formulaires prêts. En pratique, il pourrait toute-

fois être difficile d'utiliser ces formulaires génériques. Il s'avère souvent que le formulaire entier peut être créé automatiquement ... à part un seul champ.

Contrôleur

L'ajout de *Front Controller* nous a donné d'énormes possibilités pour contrôler les activités qui sont lancées pour chaque action. Ce bloc d'action est très pratique car il facilite considérablement l'ajout de p. ex. de la gestion des sessions ou d'authentification. Après avoir effectué les tâches communes, le contrôle est transmis au contrôleur d'action adéquat.

Listing 13. Front Controller et gestion de MultiActionController

```
...
try{
    $multiaction_controller = $_GET['module'];
    if ($multiaction_controller != ''){
        $multiaction_controller_file_name =
            DIR_ACTIONS.'/'.$multiaction_controller.'.module.php';
        if (file_exists($multiaction_controller_file_name)){
            require_once($multiaction_controller_file_name);
            $multiactionclassname =
                $multiaction_controller.'MultiActionController';
            $actioncontroller = new $multiactionclassname();
            $action_to_run = $_GET['action'];
            if ($action_to_run!=''){
                $method_name = $action_to_run.'_action';
                if (method_exists($actioncontroller, $method_name)){
                    $mv = $actioncontroller->$method_name();
                    if ($mv!=null){
                        require_once(SMARTY_DIR_LIB.'/Smarty.class.php');
                        $smarty = new Smarty();
                        $smarty->tempalte_dir = SMARTY_DIR_TEMPALTES;
                        $smarty->compile_dir = SMARTY_DIR_TEMPALTES_C;
                        $smarty->debugging = SMARTY_DEBUG;
                        $smarty->assign($mv->getModel());
                        $smarty->display($mv->getView());
                    }
                }
                else{
                    throw new Exception("Aucune action '$method_name'
                        dans le module '$multiaction_controller!'");
                }
            }
            else{
                throw new Exception("Le nom de l'action à lancer n'a pas été
                    donné !");
            }
        }
        else{
            throw new Exception("Aucun fichier '$multiaction_controller_file_name'
                contenant le contrôleur d'action !");
        }
    }
    else{
        throw new Exception("Le nom du module n'a pas été donné !");
    }
}
catch (Exception $e){
    print_r($e);
}
```

Rien n'empêche le *Front Controller* de choisir une façon d'agir différente.

Une autre approche, souvent utilisée, consiste à placer toutes les actions du modèle sélectionné dans une seule classe. Regardez le Listing 12 qui présente le code d'un exemple de contrôleur qui regroupe plusieurs actions (en anglais *Multi Action Controller*). Il est évident que le *Front Controller* (Listing 13) ainsi que le format URL doivent également être modifiés, p. ex. : `http://[URL de Front Controller]?module=[nom_du_module]&action=[nom_de_l'action]`.

La solution présentée permet de limiter le nombre des fichiers créés. Quand vous ouvrez l'un d'eux pour l'éditer, vous pouvez voir la fonctionn-

alité complète du module, ce qui facilite le développement et la surveillance du code. Après avoir créé de nombreux contrôleurs d'action, vous remarquerez qu'une partie du code se ressemble et se répète dans l'ensemble des actions du module. Manipuler les objets est similaire, indépendamment du fait que ce sont des actualités ou une liste d'utilisateurs. Ces ressemblances apparaissent particulièrement bien pendant la réalisation de différents types des modules d'administration. Pouvons-nous nous faciliter la vie ? Bien évidemment, il suffit de préparer un contrôleur général du type Multi Action. Il comprendra sûrement les actions comme :

- *list* – affichage de la liste des objets,
- *addform*, *add* – affichage du formulaire d'un nouvel objet et son ajout,
- *editform*, *edit* – affichage du format de l'édition de l'objet et son actualisation,
- *delete* – suppression de l'objet.

Il est assez facile de paramétrer ce contrôleur : il suffit de donner le nom de l'objet DAO. Regardez le Listing 14. Écrire une application, n'est-il pas une activité simple et agréable ?

Architecture générale

Vous êtes partis d'un simple exemple du Listing 1 et vous êtes arrivés à un cadre assez développé ; vous pouvez vous baser sur ce cadre pour créer des applications solides. Regardez encore une fois l'architecture complète (Figure 3) de ce que vous venez de réaliser.

Cette figure nous démontre qu'un cadre se compose de plusieurs modules qui travaillent ensemble mais qui ne sont pas connectés en permanence. Cette séparation de chaque module est très importante. Il est ainsi plus facile d'échanger le moteur de rendu des templates : il suffit de copier quelques lignes de code dans un seul endroit précisément déterminé. Le même principe concerne la stratégie de sélection du contrôleur d'action.

Le cadre n'impose pas non plus de transfert de contrôle lors de gestion des formulaires. Vous pouvez également préparer les contrôleurs d'action où un tel transfert serait codé. Mais vous n'êtes pas obligés de le faire ! C'est cela l'essentiel d'un framework utile : c'est un ensemble d'outils dont vous avez juste-ment besoin pour résoudre le problème.

Conclusion

Si vous utilisez un cadre en tant que base, il permettra de bien organiser votre projet et de réduire considérablement le temps du travail sur l'application. Une seule condition s'impose : Savoir choisir un bon cadre ou savoir en créer un vous-même.

Si vous décidez de choisir une solution prête à l'emploi, il est recommandé de prendre votre temps pour trouver et bien connaître un cadre concret. Rien n'est pire que de constater au milieu de son projet que le squelette choisi vous gêne plus qu'il ne vous aide. Soyez attentifs aux produits où il n'est pas possible d'échanger les éléments du cadre. Évitez à tout prix les logiciels dont la

Listing 14. Contrôleur général de plusieurs actions

```
abstract class AbstractMultiActionController {
    protected $_modelclassname;
    protected $_daoclass;
    protected $_dbconn;
    protected $_tpl_prefix;

    public function __construct(){
        $this->_dbconn = DBManager::getConnection();
    }

    public function list_action(){
        $result_arr['items_list'] = $this->_daoclass->findAllItems();
        $mv = new ModelAndView();
        $mv->setModel($result_arr);
        $mv->setView($this->_tpl_prefix.'_list.tpl.html');

        return $mv;
    }

    public function addform_action(){
        $mv = new ModelAndView();
        $mv->setModel($_POST);
        $mv->setView($this->_tpl_prefix.'_addform.tpl.html');

        return $mv;
    }

    public function add_action(){
        $biz_obj = new $this->_modelclassname($_POST[$this->_modelclassname]);
        $this->_newsmodeldao->create($biz_obj);

        return $this->list_action();
    }

    ....
}

class newsMultiActionController extends AbstractMultiActionController{
    private $_newsmodeldao;

    public function __construct(){
        parent::__construct();
        $this->_modelclassname = 'NewsModel';
        $this->_daoclass = new NewsModelDao($this->_dbconn);
        $this->_tpl_prefix = 'news';
    }
}
```

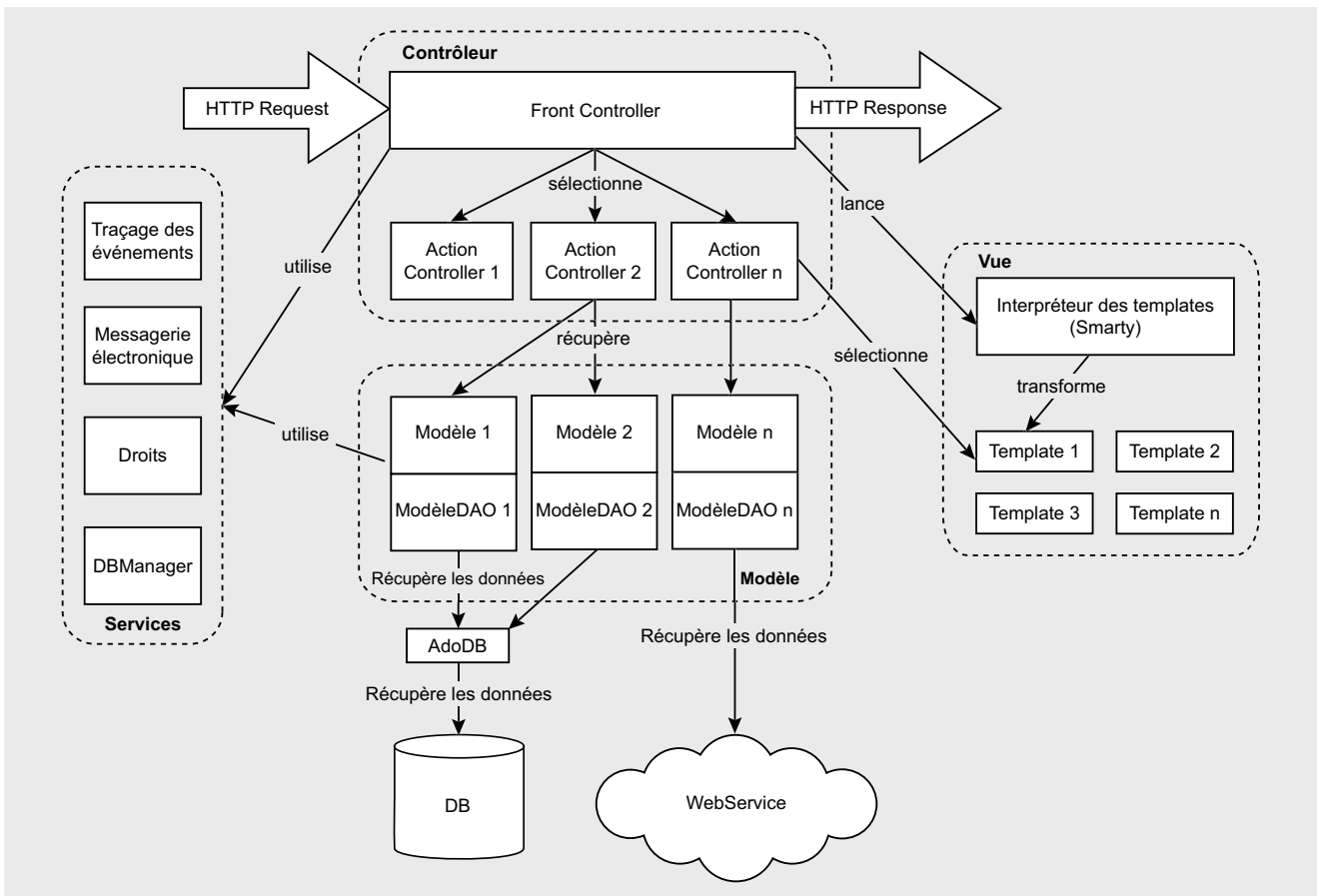


Figure 3. Architecture d'un cadre créé

logique de gestion des formulaires est fixe (p. ex. enregistrement des données doit être précédée par la validation) ou d'autres actions. Si le cadre impose trop de contraintes ou d'exigences, vous passerez la plupart du temps à adapter la logique de l'application à la philosophie du cadre et non à réaliser vos tâches.

Commencer à travailler sur son propre cadre est une décision sérieuse. Comme vous pouvez le constater sur les listings présentés, le code n'est pas particulièrement compliqué mais sa réalisation sous la forme présentée a nécessité plusieurs mois de tests et d'erreurs liés aux autres cadres. Si vous voulez écrire un

bon squelette, rien ne vaut mieux qu'une bonne expérience. Grâce à elle, vous serez capables d'identifier et de résoudre les problèmes répétitifs rencontrés dans de précédents projets. Parce que le cadre n'est rien d'autre que l'expérience de nombreux programmeurs placée dans un code concret. ■

Tableau 1. Comparaison des frameworks pour PHP les plus populaires ; les quatre premières positions sont les solutions les plus mûres

Nom	URL	Version la plus récente	Date de la version la plus récente	Développé depuis	Documentation de l'utilisateur	Documentation API
WACT	wact.sourceforge.net	0.2 alpha	2004-12	2003	+	+
Mojavi	www.mojavi.org	3.0.0-DEV	Actualisé tous les jours	2003	+	+
Seagull	seagull.phpkitchen.com	0.4.0 dev1	2004-12	2004	+	+
PHP Reusable Web Framework	www.rwfphp.org	0.1.2	2004-10	2003	+	+
InterJinn	www.interjinn.com	0.9.2	2004	2001	+	+
php.MVC	www.phpmvc.net	0.3.4 beta	2004-04	2002	+	+
Phrame	phrame.sourceforge.net	2.1 RC1	2004-01	2002	+	+
Ismo	ismo.morrdusk.net	0.1.4	2004-06	2002	+	+
binarycloud	www.binarycloud.com	3.0.0 RC2	2004-07	2004	+	+
Medusa	medusa.tigris.org	0.1.1	2004-07	2004	+	+
Studs	www.mojavelinux.com/projects/studs/	0.9.4	2004-09	2003	+	+