Introduction au système MVC

Document relatif au système MVC mis en place lors du projet **glovz.**

Equipe

Jean-Baptiste	=>	Développeur, testeur
Thomas	=>	Architecte, développeur
Fabien	=>	Chef de projet, infographiste, intégrateur, ergonome

Sommaire

1.1 - Qu'est-ce qu'un système MVC ?

Avant d'aller plus loin, une petite présentation du MVC est indispensable. **MVC signifie Modèle / Vue / Contrôleur.** C'est un découpage couramment utilisé pour développer des applications web.

Chaque action d'un module appartient en fait à un **contrôleur**. Ce contrôleur sera chargé de générer la page suivant la requête (HTTP) demandée par l'utilisateur. Cette requête inclut des informations comme l'URL, avec ses paramètres `GET`, des données `POST`, des `COOKIES`, etc. Un module peut être divisé en plusieurs contrôleurs, qui contiennent chacun plusieurs actions.

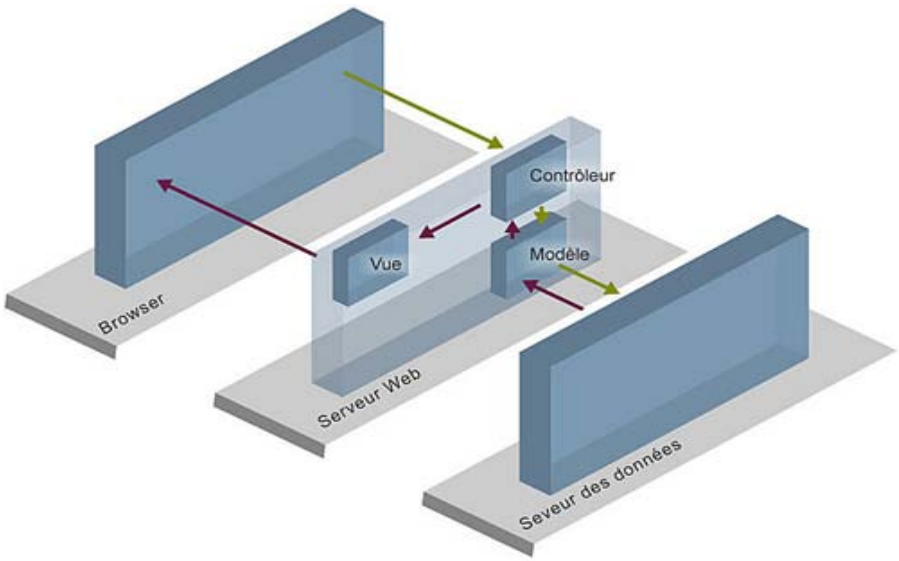
Pour générer une page, un contrôleur réalise presque systématiquement des opérations basiques telles que lire des données, et les afficher. Avec un peu de capacité d'abstraction, on peut voir deux autres couches qui apparaissent : une pour gérer les données (notre **modèle**) et une autre pour gérer l'affichage des pages (notre **vue**).

Le **modèle** : une couche pour gérer les données, ça signifie quoi ? Ça signifie qu'à chaque fois que nous voulons créer, modifier, supprimer ou lire une donnée (exemple, lire les informations d'un utilisateur depuis la base de données MySQL), nous ferons appel à une fonction spécifique qui nous retournera le résultat (sous forme d'un tableau généralement). Ainsi, nous n'aurons AUCUNE requête dans notre contrôleur, juste des appels de fonctions s'occupant de gérer ces requêtes.

La **vue** : une couche pour afficher des pages ? Cela signifie tout simplement que notre contrôleur n'affichera JAMAIS de données directement (via echo ou autre). Il fera appel à une page qui s'occupera d'afficher ce que l'on veut. Cela permet de séparer complètement l'affichage HTML dans le code. Certains utilisent un moteur de templates pour le faire, nous n'en aurons pas besoin : l'organisation des fichiers se suffit à elle-même.
Ceci permet de séparer le travail des *designers* et graphistes (s'occupant de créer et de modifier des vues) et celui des programmeurs (s'occupant du reste).

En externalisant les codes du modèle et de la vue de la partie contrôleur, vous verrez que le contenu d'un contrôleur devient vraiment simple et que sa compréhension est vraiment aisée. Il ne contient plus que la logique de l'application, sans savoir comment ça marche derrière : par exemple, on voit une fonction `verifier_unicite_pseudo()`, on sait qu'il vérifie que le pseudo est unique, mais on ne s'encombre pas des 10 lignes de codes SQL nécessaires en temps normal, celles-ci faisant partie du modèle. De même pour le code HTML qui est externalisé.

Source SDZ.



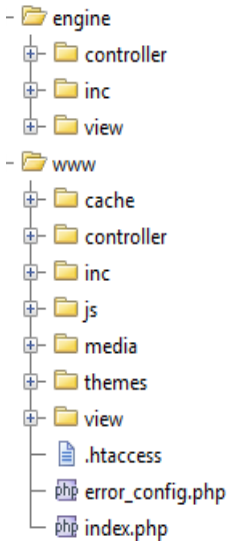
1.2 – Le multicouche

Le système MVC est séparé en deux parties distinctes.

D'un côté le dossier contenant notre projet (www/) et de l'autre, le dossier relatif au moteur de notre MVC (engine/).

Chaque partie dispose de la même architecture de base, dossier :

- controller/,
- inc/,
- view/



Principe de fonctionnement :

La surcharge
Chaque class, template et fichier d'informations est surchargeable.
Ainsi, si un appel vers une classe/vue/controller xxx est effectuée, le système va vérifier sa présence en premier lieu dans le dossier www/ puis, si il n'est pas trouvé, ira le récupéré dans le dossier engine/.
Ce principe permet d'élaboré des comportements par défaut pour le(s) projet(s) utilisant le moteur tout en permettant de le modifier (souplesse).

Séparation des classes métier et des classes librairies
La principale caractéristique d'un MVC, est le gain de clarté.
Ainsi, par le multicouche nous poussons ce principe plus loin.

Toutes les classe librairies (gestion des contrôleurs, autoload, etc …) sont définit dans l'engine. Et inversement, les classes métiers dans le dossier www/

2.1 - L'url rewriting (contrôleur)

L'URL Rewriting (réécriture d'URL en bon français) est une technique utilisée pour optimiser le référencement des sites dynamiques (utilisant des pages dynamiques).
Les pages dynamiques sont caractérisées par des URL complexes, comportant en général un point d'interrogation, éventuellement le caractère & ainsi que des noms de variables et des valeurs.

Exemple : article.php?id=12&page=2&rubrique=5

Dans cet exemple, le fichier article.php est utilisé pour afficher un article dont le texte vient d'une base de données.
C'est un fichier générique, qui peut afficher n'importe quel article, de n'importe quelle rubrique, page par page. Ici on cherche à afficher la page 2 de l'article numéro 12 qui fait partie de la rubrique 5.

« ... »

Le principe est très simple : sur un site qui utilise l'URL Rewriting, on ne peut plus se rendre compte qu'il est basé sur des pages dynamiques. En effet, les URL sont "propres" : elles ne contiennent plus tous les caractères spéciaux comme ? ou &. Personne ne peut savoir qu'il s'agit de pages dynamiques, que ce soit un robot d'indexation ou un internaute.

« ... »

Le webmaster doit changer la façon dont il écrit les liens, selon des règles qu'il va se fixer lui-même. En reprenant l'exemple ci-dessus, on peut remarquer que les URL pour les pages d'articles ont toutes la même forme.

Source webrankinfo

Dans le cas présent

Dans notre système actuel, l'url rewriting fonctionne selon deux principes de base :

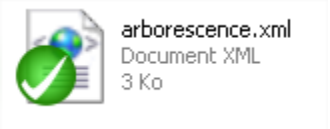
1- Tout ce qui n'existe pas est redirigé sur l'index
2- L'arborescence est gérer via le fichier xml se trouvant dans www/inc (ou engine/)/ arborescence.xml

Redirection

L'accès direct aux fichiers est conservé (image, css, js, etc ...).
Tout les virtuals folder qui n'existerait pas au sein de l'architecture physique du projet sont redirigé sur l'index (via le .htaccess)

Gestion de l'arborescence

Premier exemple : arrivé sur l'index



Lors de l'arrivé sur l'index,

Notre url correspond donc à monsite.com/

Le MVC va ouvrir le fichier arborescence.xml se trouvant dans le dossier inc/

```
$SiteObj = Base::Load(CLASS_BASE);
$SiteObj->Start($_SERVER['REQUEST_URI']);
```

Contenu du fichier :

```
<arborescence>
        <index>
                <title>Page Index</title>
                <controller>home</controller>
                <layout>drupal.tpl</layout>
                <blocks>
                        <block>
                                <controller>articles</controller>
                                <method>formulaireInscription</method>
                        </block>
                        <block>
                                <controller>articles</controller>
                                <method>mafunction</method>
                        </block>
                </blocks>
        </index>
```

Notre url ne disposant pas d'argument, c'est le paramètre « index » qui sera utilisé par défaut

Obligatoire
Title : définit le titre de la page par défaut,
Controller : définit le contrôleur maître à utiliser,
Layout : Choix du gabarit global de la page,

Facultatif
Blocks : Appel des contrôleurs esclaves. Utile pour la gestion des block récurent (connexion, etc ..)
Method : nom de la méthode à utiliser dans le contrôleur maître.
accessControl : permet de verrouiller un controller par mot de passe ex :

```
<accessControl>
   <login>admin</login>
   <password>a7cadb54ca75f2595c6a2a2139aa3e94</password>
</accessControl>
```

Suite à cette lecture du fichier, le contrôleur « **home** » va être appelé automatiquement.
Son adresse se situe controller/**home**/**home**.controller.php

```
Class home_controller {

   function home_controller(){
      $this->_view = Base::Load(CLASS_COMPONENT)->_view;
   }

   function defaut(){
      $this->_view->addBlock('main', 'index.tpl');
   }

}
```

La nomination de la classe est définit par **home**_controller.
Aucune méthode étant définit dans le fichier d'info, c'est la méthode defaut() qui sera automatiquement appelé.

Second exemple : sous arbo

url demandé : monsite.com/articles/lpcm/titre-article-888.html

Le MVC va rechercher le fichier arborescence.xml

```
        <lpcm>
            <title>Page article 3 - meme controller</title>
            <controller>articles</controller>
            <method>lpcm</method>
            <layout>drupal.tpl</layout>
        </lpcm>
    </articles>
```

Dans ce fichier, il trouve l'argument « lpcm » correspondant à notre url, parallèlement il détermine que « titre-article-888.html » correspond à une demande et non pas à un élément d'arborescence, il l'introduit donc dans le $_GET.

Le MVC va initialiser le contrôleur articles (controller/**articles**/articles.controller.php) et appeler sa méthode lpcm().

2.2 – Le model

La gestion du model est desservie par l'utilisation du moteur de template Smarty.

Le concept d'un système de template en général et de Smarty en particulier est de réserver les tâches de production des données à PHP et de mettre le code de présentation (HTML en l'occurence) dans des 'templates' ou modèles, un fichier qu'on suffixera dans nos exemples par '.tpl'

Source Developpez

Chaque constructeur de contrôleur fait appel à Smarty avec
$this->_view = Base::Load(CLASS_COMPONENT)->_view;

Les vues relatif aux blocks se trouvent dans controller/<<controllerName>>/view/
Les layouts de page (ceux défini dans infos.php) se trouvent dans le dossier view/ à la racine.

assign()

Assignation d'une valeur
$this->_view->assign('toto','laaaaa');

addBlock()

Assignation à une variable d'un template au sein d'un même contrôleur
$this->_view->addBlock('blk', 'article.tpl');

Assignation à une variable d'un template externe
$this->_view->addBlock('formulaire', 'formulaire.tpl','controller/articles/view/');

Récupérer une valeur dans un template
{ $varName % }

Faire un include
{include file='view/test/content-top.tpl.php'}

Variable pré-rempli :
Le titre de la page défini dans infos.php est accéssible via { $Title }

Toutes les constantes du site son accéssible automatique
<link rel="stylesheet" type="text/css" href="{ **$SYS_FOLDER** }/themes/pQp.css">

url de la doc :

http://www.smarty.net/docsv2/fr/

2.3 – La vue

La vue est fortement liée au model et son utilisation de Smarty.

La fonctionnalité vue récupère l'ensemble des assignations de templates et variables qui lui ont été faite et les insère dans un buffer.

Le buffer

L'utilisation d'un buffer consiste à stocker temporairement le rendu d'une page afin de séparer projection et traitement d'une page.
Ainsi, une page peut traiter une information sans projeter de code html.
Autre conséquence, si un print_r() ou echo() est utilisé, il apparaitra avant la projection du buffer.

Projection

```
$SiteObj = Base::Load(CLASS_BASE);
$SiteObj->Start($_SERVER['REQUEST_URI']);
$SiteObj->Display();
```

3.1 – Les listeners

Le **listener**, en français **écouteur**, est un terme anglais utilisé de façon générale en informatique pour qualifier un élément logiciel qui est à l'écoute d'évènements afin d'effectuer des traitements.

Source Wikipedia

Un listener permet d'affecter automatiquement des actions prédéfini au code.
Par exemple, la validation d'un formulaire spécifique, déclenche l'appel à un contrôleur type.

Todo

Implémention au sein d'un formulaire
<input type="hidden" value="**articles**[**register**]" name="**todo**">
ControllerName[methodName]

```
function POST_register($data){
    echo 'articles::POST_register() auto !';
    echo '<pre>';
    print_r($data);
    echo '</pre>';
}
```

(extrait du controlleur articles)

La méthode demandé étant nommé register et faisant partie des listener de type Post, la méthode s'appellera donc POST_**register**.
Le paramètre $data, sera automatiquement renseigner par les données du formulaire ($_POST).

4.1.1 – Classe – Engine - Base

Cette classe est le cœur du MVC. Elle permet de liée toutes les classes entre elle et effectue tout les appels. C'est l'index des classes.

Méthode Start()

Comme son nom l'indique, elle initialise le site.
Elle est appelé par l'index avec comme argument $_SERVER['REQUEST_URI']).
C'est cette fonction qui va inclure le bon fichier infos.php et lancer les contrôleurs.

Méthode Load()

Autre méthode très importante dans le MVC, elle permet l'instanciation des classes.
Cette méthode est couplée à une classe externe servant d'autoload (__autoload de php5).
Ces paramètres sont : le nom de la classe, un tableau (array) d'arguments (facultatifs) et le paramètre **singleton** (true/false).

Le singleton

En génie logiciel, le **singleton** est un patron de conception *(design pattern)* dont l'objet est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.
On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà.

Source Wikipedia

Exemple d'utilisation :

```
$SiteObj = Base::Load(CLASS_BASE);
Ou plus complexe,
Base::Load(CLASS_CONTROLLER,array($ControllerName,$control[INFOS_METHOD]));
```

Méthode Display()

Cette méthode affiche le rendu d'une page, c'est notre vue (voir 2.3).

4.1.2 – Classe – Engine - Contrôleur

Cette classe permet l'instanciation via le base->load() des contrôleurs.
Elle n'est jamais appelée directement.

4.1.3 – Classe – Engine - Vue

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc). Ces différents événements sont envoyés au contrôleur. La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

Source wikipedia

La vue utilise le gestionnaire de vue Smarty (voir 2.2 et 2.3).
Son appel s'effectue au niveau de la classe abstraite « component », lui-même appelé par les constructeurs :

```
Class Component {
        var $_view;
        function Component(){
                $this->_bdd = Base::Load('Bdd');
        }
}

Class articles_controller{
        function articles_controller(){
                $this->_view = Base::Load(CLASS_COMPONENT)->_view;
        }
}
```

Voir 2.2 pour plus d'informations.

4.1.4 – Classe – Engine - Bdd

A refaire avec MongoDB.

4.1.5 – Classe – Engine - Coremessage

Cette classe permet la gestion des messages de type « critic », « warning » et « generic ».
L'intégralité de ces messages sont défini au sein de fichiers xml nommé **coremessage.xml**

Une partie se trouve dans engine/inc/**coremessage.xml**
Et la seconde partie dans www/inc/**coremessage.xml**

Le premier fichier défini les messages liée à l'engine en lui-même (erreur de chargement de classes, initialisation etc …).
Le second quant à lui permet de définir des messages liée au projet et autorise également la surcharge des messages de l'engine.

Par exemple :
(engine/)
<message id="ERR_LOAD_CLASS">Impossible de charger la class</message>
Peut tout à fait être remplacé sans toucher au fichier principal dans le fichier du projet (www/)
<message id="ERR_LOAD_CLASS">Impossible de charger la class du projet SEL</message>

Il suffit d'utiliser la même clé.

Critic – Warning – Généric.

Bien qu'utilisant le même fichier, ces trois méthodes d'appel on un comportement différent.
Warning et générique vont simplement retourner un message alors que Critic va arrêter l'exécution de la page (function exit()).

Appel :

Pour appeler l'une de ces méthodes, nous utilisons :
Base::Load(CLASS_CORE_MESSAGE)->**Warning**($MessageCle);

Ex : *Base::Load(CLASS_CORE_MESSAGE)->**Warning**('ERR_LOAD_CLASS');*

4.1.6 – Classe – Engine - Email

Cette classe permet une utilisation simplifiée de PHPMailer.

Elle instance les méthodes SimpleMail() et SimpleMailHTML().

SimpleMail

Base ::Load(CLASS_EMAIL)->SimpleMail($FromMail, $FromName, $ToMail, $Subject, $content)

SimpleMailHTML

Base ::Load(CLASS_EMAIL)->SimpleMailHTML($FromMail, $FromName, $ToMail, $Subject, $content)

Cette methode appel SimpleMail() en lui incluant un encodage en Html

Autres :

Voir doc :
http://www.zenphoto.org/documentation/plugins/PHPMailer/PHPMailer.html

4.1.7 – Classe – Engine - Encode

Encode n'est pas une classe à proprement parlé. Elle est intégrée à la classe Base().

Encode permet par l'appel d'un controller se trouvant sur site.com/encode/ d'encoder une chaine de caractères à la volé en la passant dans l'url par exemple :
Site.com/encode/admin va encoder le mot « admin » et donc ressortir
« a7cadb54ca75f2595c6a2a2139aa3e94 » (suivant l'encodage défini dans le fichier de config).

Cet encodage est notamment utilisé pour l'accessControl (2.1).

Fonctionnement

Son utilisation est défini par l'utilisation des fonctions selEncode() et selDecode()

5 – Content Manager :

Le content manager permet de gérer une multitude de contenu sans avoir à gérer la création de formulaire et la sauvegarde des données associé.

5.1 – Content Manager - Content Type

Un fichier de configuration XML disponible dans *engine/inc/contentManager/content_type.xml* définit les différentes briques de base qui constitue un formulaire.

Exemple de type : input, checkbox, textarea, wysiwyg, etc …

Ces éléments ne sont ni surchargeable, ni complétable par un fichier du dossier projet (www/).

5.2 – Content Manager - Structures

Un structure défini un ensemble de content type en vue de crée un formulaire et une structure de base à un objet.
Les structures sont définit dans des fichiers xml :

- engine/inc/contentManager/content_struct.xml :
  Ce fichier défini des structures de bases qui sont verrouillé, elle ne serve que d'exemple.
- www/inc/content_struct.xml :
  Ce fichier est généré via le back-office. Il n'est pas à éditer manuellement via un IDE.

Chaque définition de structure possède au minimum un id unique, un nom, une description et un état « lock » (true/false).

Suite à l'édition d'une structure, la définition incorpore ensuite les différents « content-type ».

Exemple de structure :

```
<element id="article" locked="true">
    <name>Defaut - Article</name>
    <description>Structure de base d'un article</description>
    <types>
        <type refType="10">
            <name>Titre</name>
            <id>title</id>
            <valeur/>
            <limit>70</limit>
            <index>true</index>
        </type>
        <type refType="40">
            <name>Image</name>
            <id>picture</id>
            <valeur></valeur>
        </type>
        <type refType="20">
            <name>Chapeau</name>
            <id>strapline</id>
            <valeur>800</valeur>
            <index>true</index>
        </type>
        <type refType="21">
            <name>Corps du texte</name>
            <id>content</id>
            <valeur></valeur>
        </type>
    </types>
</element>
```

5.3 - Content Manager – Structures – Edition

L'édition d'une structure s'effectue au sein de l'interface d'administration dans : admin/content-manager/structures/

Chaque « Content-type » est défini par :

- un label, affiché dans les formulaires,
- un Id, qui ne doit pas avoir d'espace et de caractères spéciaux,
- une valeur par défaut (peut être vide),
- une limite de caractère,
- une « indexation », utile pour le listing.

5.4 – Content Manager – classe « SimpleContentManager »

La classe « **simpleContentManager** » est une classe abstraite facilitant l'interrogation du content manager.
Elle est à utilisé en extends d'une classe métier ex :

```php
class Jalon extends SimpleContentManager {

    const COLLECTION = 7;

    public function __construct(){
        $this->_collection = self::COLLECTION;
        parent::__construct();
    }
}
```

Liste des méthodes accessible :

- **getStruct()**
  Retourne la structure de la collection en cour.

- **getAll()**
  Retourne la liste des objets de la collection en cour.

- **get($object_id, $withRelation = false)**
  Retourne un objet avec ses relations ou juste les id références
  Un objet compilé avec ses relations ne peut être enregistré.

- **remove($object_id)**
  Supprime un objet en fonction de son id.

- **update($data, $object_id)**
  Met à jour les données d'un objet.

- **save($data)**
  Enregistre un nouvel objet.

- **findBy($param, $value)**
  Retourne une liste d'objet en fonction d'un paramètre et de sa valeur associé.

6 - Choix Technologique :

Base de données

MongoDB

Arborescence

Arborescence.xml
Mise en pratique du cour sur le Xml pour stocker les données liée à l'arborescence du projet.

Php (version 5.3)

Dernière version en date, meilleure gestion des objets.

MVC

Séparation du model, de la vue et des contrôleurs.

Smarty

Système de templating simple, performant et standardisé. Il permet une maintenance aisée de l'affichage.
Plugin "Image Thumb" :
Génération à la vole d'image compressé et resizer pour amélioré les temps de chargements.

Xhtml & Doctype STRICT

Meilleur respect des normes préconisé par le W3C.
CSS2 Valide et cross-browser.

Javascript

Jquery
Code facilement maintenable et géré par toute l'équipe.

Principes de fonctionnement

Séparation Engine et projet

Cette séparation à pour but de clarifier la gestion des comportements par défaut et celle liée au projet. A terme, elle permet également la réutilisation et la maintenance de plusieurs projets (correction de bug, update …).

Surcharge

Chaque classe et fichier d'informations peut être surchargé en vue de modifier un comportement ou action normalement gérer par l'engine.

Fusion

Certain éléments sont à la fois sur chargeable et fusionnable. C'est le cas par exemple du fichier xml d'arborescence. Ces éléments peuvent être réécrit ou contraire complémentaire (fichier engine et fichier projet).

Surcouche

Les classes importées de l'extérieur (doctrine, smarty …) ne sont jamais utilisé directement et ne doivent jamais être modifié. Certaines classes (bdd, view …) héritent de celles-ci.
Cette pratique permet la mise à jour future de ces classes avec plus de souplesse.

Singleton (Base ::Load())

Toutes les classes instanciées le sont par ce composant. Cette pratique évite une trop grande consommation serveur et autorise de plus grande capacité de fonctionnement (ex : utilisation de la même instance de vue pour la compléter au fur et à mesure).