Documentation technique

Document technique relatif au CMS mis en place lors du projet **macrise.**

Équipe

Thomas Van Horde : Architecte, développeur de la base Mongo, des classes métiers et contrôleurs

Fabien Nouaillat : Chef de projet, infographiste, intégrateur, ergonome, développeur front office

Sommaire

1. Architecture
    - 1.1. Le système MVC
    - 1.2. Le framework
    - 1.3. La réécriture d'URL
    - 1.4. Le modèle
    - 1.5. La vue
    - 1.6. L'observateur
2. Classes de l'engine
    - 2.1. Base
    - 2.2. Controller
    - 2.3. Vue
    - 2.4. Base de données
    - 2.5. Coremessage
    - 2.6. Email
    - 2.7. Encode
    - 2.8. Pdf
3. Content Manager
    - 3.1. Content type
    - 3.2. Structure
    - 3.3. Structure – Édition
    - 3.4. Classe SimpleContentManager
4. Choix technologiques

1.1–Le 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.

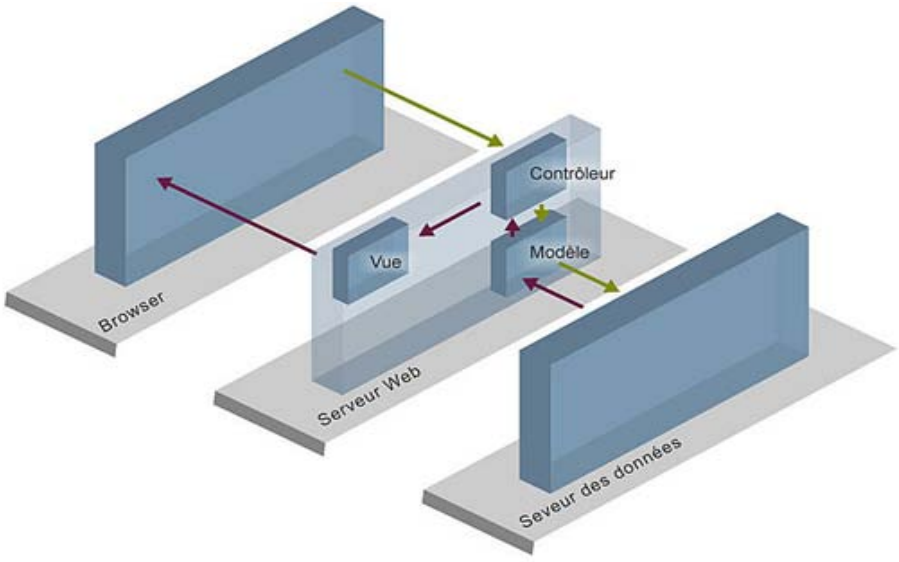
Pour générer une page, un contrôleur réalise presque systématiquement des opérations basiques telles que lire des données, et les afficher. Avec un peu de capacité d'abstraction, on peut voir deux autres couches qui apparaissent : une pour gérer les données (notre **modèle**) et une autre pour gérer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



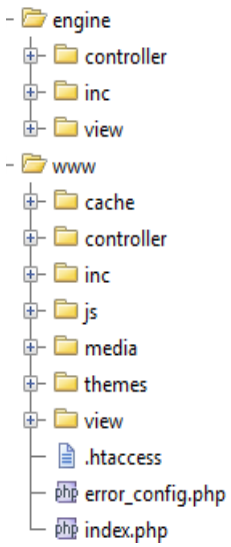
1.2 – Le framework

Le framework est une architecture de base commune à tout projet. Il 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 système (engine/).

Chaque partie dispose de la même architecture de base où on retrouve le principe MVC. On y trouve les dossiers suivants :

- controller/ pour le contrôleur
- inc/ pour l'accès à la base de données
- view/ pour la vue



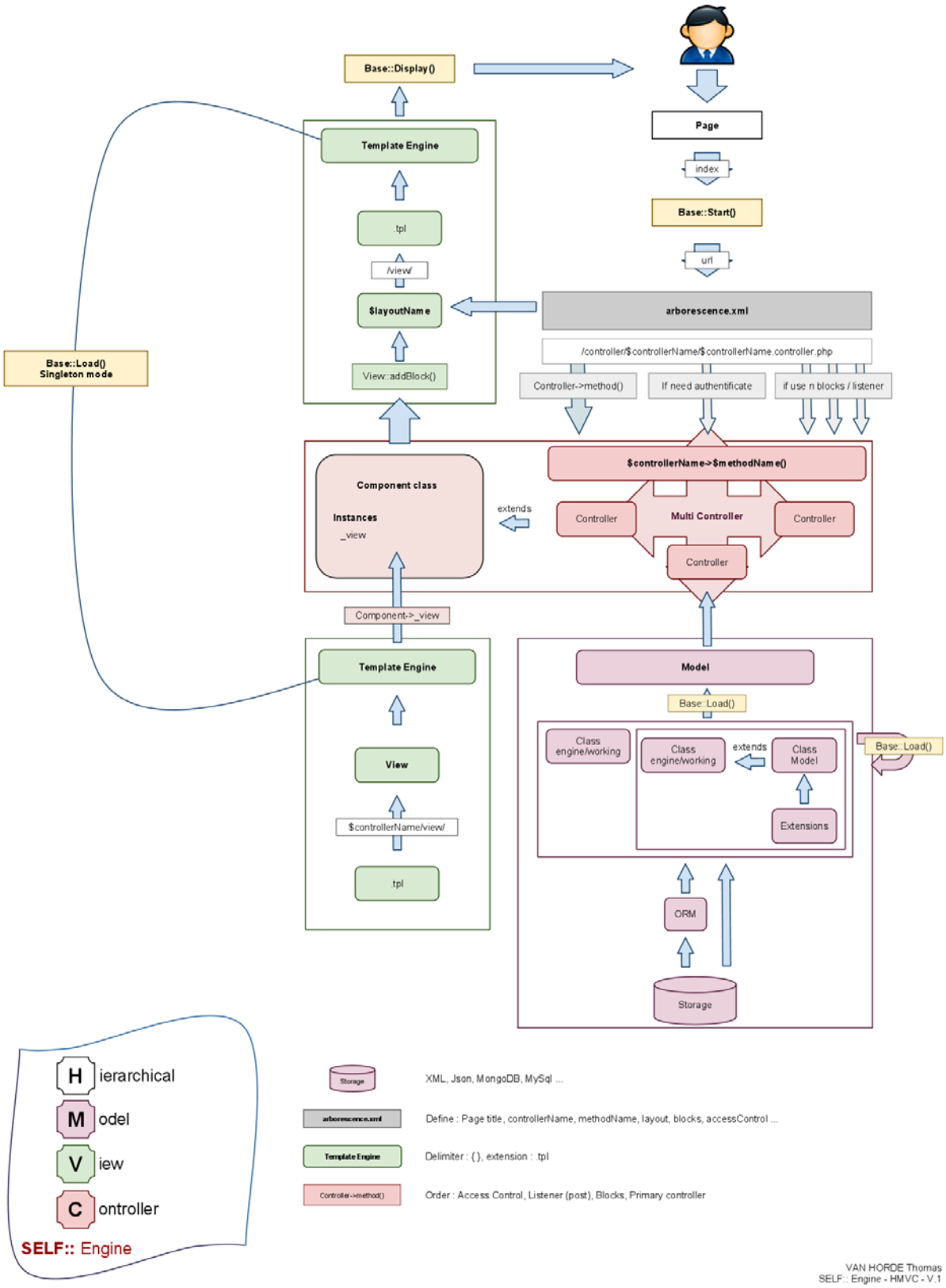
Principe de fonctionnement :

La surcharge
Chaque classe, gabarit et fichier d'informations estsurchargeable.
Ainsi si on appelle une classe, vue ou contrôleur, le système va vérifier sa présence en premier lieu dans le dossier www/ puis - s'il n'est pas trouvé – il ira le récupérer dans le dossier engine/.
Ce principe permet d'élaborer des comportements par défaut pour le(s) projet(s) utilisant le moteur tout en permettant de le modifier (souplesse).

Séparation des classes métiers et des bibliothèques
La principale caractéristique du MVC est le gain de clarté. Nous le mettons encore plus en évidence avec la séparation du moteur et de l'application.

Toutes les bibliothèques (gestion des contrôleurs, auto-chargement des classes, etc.) sont définies dans l'engine et inversement, les classes métiers dans le dossier www/.

Schéma de conception du MVC



1.3–La réécriture d'URL

L'URL Rewriting (ou réécriture d'URL en français) est une technique utilisée pour optimiser le référencement des sites utilisant des pages dynamiques.
Les pages dynamiques sont caractérisées par des adresses 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 la réécriture d'URL, 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'URLRewriting fonctionne selon deux principes de base :

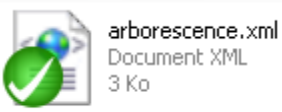
1- Tout ce qui n'existe pas est redirigé sur l'index
2- L'arborescence est gérée via le fichier XML se trouvant dans inc/arborescence.xml

Redirection

L'accès direct aux fichiers est conservé (image, css, js, etc.).
Tous les fichiers virtuels qui n'existeraient pas au sein de l'architecture physique du projet sont redirigés sur l'index (via le .htaccess).

Gestion de l'arborescence

Premier exemple : arrivé sur l'index



Lors de l'arrivée sur l'index, notre URL correspond 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']);

Exemple de contenu du fichier :

```
<arborescence>
	<index>
		<title>Page Index</title>
		<controller>home</controller>
		<layout>defaut.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 blocs récurrents (connexion, etc.)
Method : nom de la méthode à utiliser dans le contrôleur maître.
accessControl : permet de verrouiller un controller par mot de passe

Par exemple :

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

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

```
Class home_controller {

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

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

}
```

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

Second exemple : sous arborescence

URL demandée : monsite.com/articles/lpcm/titre-article-888.html

Le système va rechercher le fichierarborescence.xml.

```
            <lpcm>
                <title>Page article 3 - meme controller</title>
                <controller>articles</controller>
                <method>lpcm</method>
                <layout>defaut.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 des articles (controller/**articles**/articles.controller.php) et appeler sa méthode lpcm().

1.4 – Le modèle

La gestion du modèle est desservie par l'utilisation du moteur de gabarit Smarty.

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

Source Developpez

Chaque contrôleur hérite de la class Component qui dispose de
$this->_view, instance de Smarty.

Les vues relatives aux blocs se trouvent dans controller/<nom_du_controleur>/view/
Les rendus de page (ceux définis dans infos.php) se trouvent dans le dossier view/ à la racine.

assign()

Assignation d'unevaleur
$this->_view->assign('foo','bar');

addBlock()

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

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

Récupérer une valeur dans un gabarit
{ $varName % }

Faire une inclusion de fichier
{include file='view/test/content-top.tpl.php'}

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

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

URL de la documentation : http://www.smarty.net/docsv2/fr/

1.5 – La vue

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

La fonctionnalité vue récupère l'ensemble des assignations de gabarits et variables qui lui ont été faites et les insère dans une mémoire tampon.

La mémoire tampon

L'utilisation d'un buffer (ou mémoire tampon) 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();

1.6 – Les observateurs

L'observateur (ou écouteur) est un terme informatique pour qualifier un élément logiciel qui est à l'écoute d'évènements afin d'effectuer des traitements.

Un observateur permet d'affecter automatiquement des actions prédéfinies 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]

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

La méthode demandée étant nommée« register » et faisant partie des écouteurs de type Post, la méthode s'appellera donc POST_**register**.
Le paramètre $data sera automatiquement renseigné par les données du formulaire ($_POST).

2.1 – Classe Base

Cette classe est le cœur du MVC. Elle permet de lier toutes les classes entre elles et effectue tous les appels. C'est l'index des classes.

Méthode Start()

Comme son nom l'indique, elle initialise le site.
Elle est appelée 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'auto-chargement (__autoload de php5).
Ses paramètres sont : le nom de la classe, un tableau (array) d'arguments (facultatifs) et un paramètre booléen (true/false) ; ce paramètre est optionnel et a pour valeur true par défaut.

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 :

- $SiteObj = Base::Load(CLASS_BASE);
- Base::Load(CLASS_CONTROLLER,array($ControllerName,$control[INFOS_METHOD]));

1.1 Méthode Display()

Cette méthode affiche le rendu d'une page (la vue).

2.2 – Classe Controller

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

2.3 –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 moteur de gabarits Smarty.
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{
        functionarticles_controller(){
                $this->_view = Base::Load(CLASS_COMPONENT)->_view;
        }
}
```

2.4 – Base de données

Utilisation de MongoDB.

2.5 –Coremessage

Cette classe permet la gestion des messages de type « critic », « warning » et « generic ».
L'intégralité de ces messages est définie au sein des fichiers XML nommés**coremessage.xml**.

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

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

Par exemple :
(engine/)
<message id="ERR_LOAD_CLASS">Impossible de charger la classe</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 classe</message>

Il suffit d'utiliser la même clé.

Critic – Warning – Generic

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

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'*);*

2.6 –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 méthode appelle SimpleMail() en l'encodant au format HTML.

Autres :

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

2.7 –Encode

Encode n'est pas une classe à proprement parler. Elle est intégrée à la classe Base.

Encode permet par l'appel d'un contrôleur se trouvant sur site.com/encode/ d'encoder une chaîne de caractères à la volée 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 configuration).

Cet encodage est notamment utilisé pour l'accessControl.

Fonctionnement

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

2.8 –Pdf

HTML2PDF est un convertisseur de code HTML vers PDF écrit en PHP.
Il permet la conversion d'HTML 4.01 valide au format PDF, et est distribué sous licence LGPL.

Cette bibliothèque a été conçue pour gérer principalement les tables imbriquées afin de générer des factures,bons de livraison et autres documents officiels. Elle ne gère pas encore toutes les balises.

Exemple d'utilisation :
Base::Load(**CLASS_PDF**)->simplePDF('members.tpl', 'exemple.pdf');

Se référer au wiki officiel pour connaître toutes les possibilités :
http://wiki.spipu.net/doku.php?id=html2pdf:fr:v4:accueil

3 – Content Manager

Le content manager permet de gérer une multitude de contenus sans avoir à gérer la création de formulaires et la sauvegarde des données associées.

3.1 –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 types : input, checkbox, textarea, wysiwyg, etc.

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

3.2 –Structures

Une structure définit un ensemble de contenu type en vue de créer un formulaire et une structure de base à un objet.
Les structures sont définies dans des fichiers XML :

- engine/inc/contentManager/content_struct.xml :
  Ce fichier définit des structures de base qui sont verrouillées, elle ne servent que d'exemple.
- www/inc/content_struct.xml :
  Ce fichier est généré par 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 types ».

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>
```

3.3 – 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'espacesou de caractères spéciaux,
- une valeur par défaut (qui peut être vide),
- une limite de caractères,
- une « indexation » utile pour le listing.

3.4 – Classe SimpleContentManager

La classe « **SimpleContentManager** » est une classe abstraite facilitant l'interrogation du content manager.
Elle est à utiliser en héritage d'une classe métier. Par exemple :

```php
class Jalon extends SimpleContentManager {

    const COLLECTION = 7;

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

Liste des méthodes accessibles :

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

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

- **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'objets en fonction d'un paramètre et de sa valeur associée.

- **addForm($blockName, $action = false, $template = false)**
  Permet d'insérer un formulaire complet automatiquement
  Exemple : this->_userClass->addForm('content', 'MemberData');
  Gabarit par défaut : engine/inc/contentManager/contentManagerForm.tpl

- **editForm($blockName, $objectId, $action = false, $template = false)**
  Permet d'éditer un objet en recréant un formulaire dynamiquement
  Exemple : $this->_userClass->editForm('content', $_SESSION['user']['uid'], 'MemberData');
  Gabarit par défaut : engine/inc/contentManager/contentManagerForm.tpl

4 - Choix technologiques :

Base de données

MongoDB

Arborescence

Arborescence.xml
Mise en pratique du cours sur le XML pour stocker les données liées à l'arborescence du projet.

PHP 5.3

Dernière version en date, meilleure gestion des objets.

Modèle MVC

Séparation du modèle, de la vue et des contrôleurs.

Smarty

Système de gabarits simple, performant et standardisé. Il permet une maintenance aisée de l'affichage.
Plugin « Image Thumb » :
Génération à la volée d'images compressées et redimensionnement pour améliorer les temps de chargements.

XHTML 1.0 STRICT

Meilleur respect des normes préconisées par le W3C.

Framework LESS CSS

Utilisation de CSS 2.1 et quelques nouveautés du CSS 3.
LESS permet une arborescence et diverses astuces pratiques pour générer du CSS.

Javascript&jQuery

Bibliothèque réputée et documentée pour faciliter le code JS.
Possède des greffons facilement implantables.

Principes de fonctionnement

Séparation du système et du projet

Cette séparation a pour but de clarifier la gestion des comportements par défaut et celle liée au projet. À terme, elle permet également la réutilisation et la maintenance de plusieurs projets (correction de bogues, mises à jour, etc.).

Surcharge

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

Fusion

Certain éléments sont à la fois surchargeable et fusionnables. C'est le cas par exemple du fichier XML d'arborescence. Ces éléments peuvent être réécrits ou au contraire complémentaires (fichier engine et fichier projet).

Surcouche

Les classes importées de l'extérieur (Doctrine, Smarty …) ne sont jamais utilisés directement et ne doivent jamais être modifiés. 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 grandes capacités de fonctionnement (ex : utilisation de la même instance de vue pour la compléter au fur et à mesure).