Documentation technique du Framework

Diapazen

ISEN | école d'ingénieurs

v1– 11/06/2013

Sommaire

Introduction

Selon Wikipédia, un framework est un ensemble d'outils et de composants logiciels organisés conformément à un plan d'architecture, l'ensemble formant ou promouvant un squelette de programme.

L'écriture d'un framework peut être longue et compliqué mais l'avantage est la réutilisation du code par la suite, il permet de formaliser une architecture adaptée au besoin de l'entreprise. Il tire parti de l'expérience des développements antérieurs.

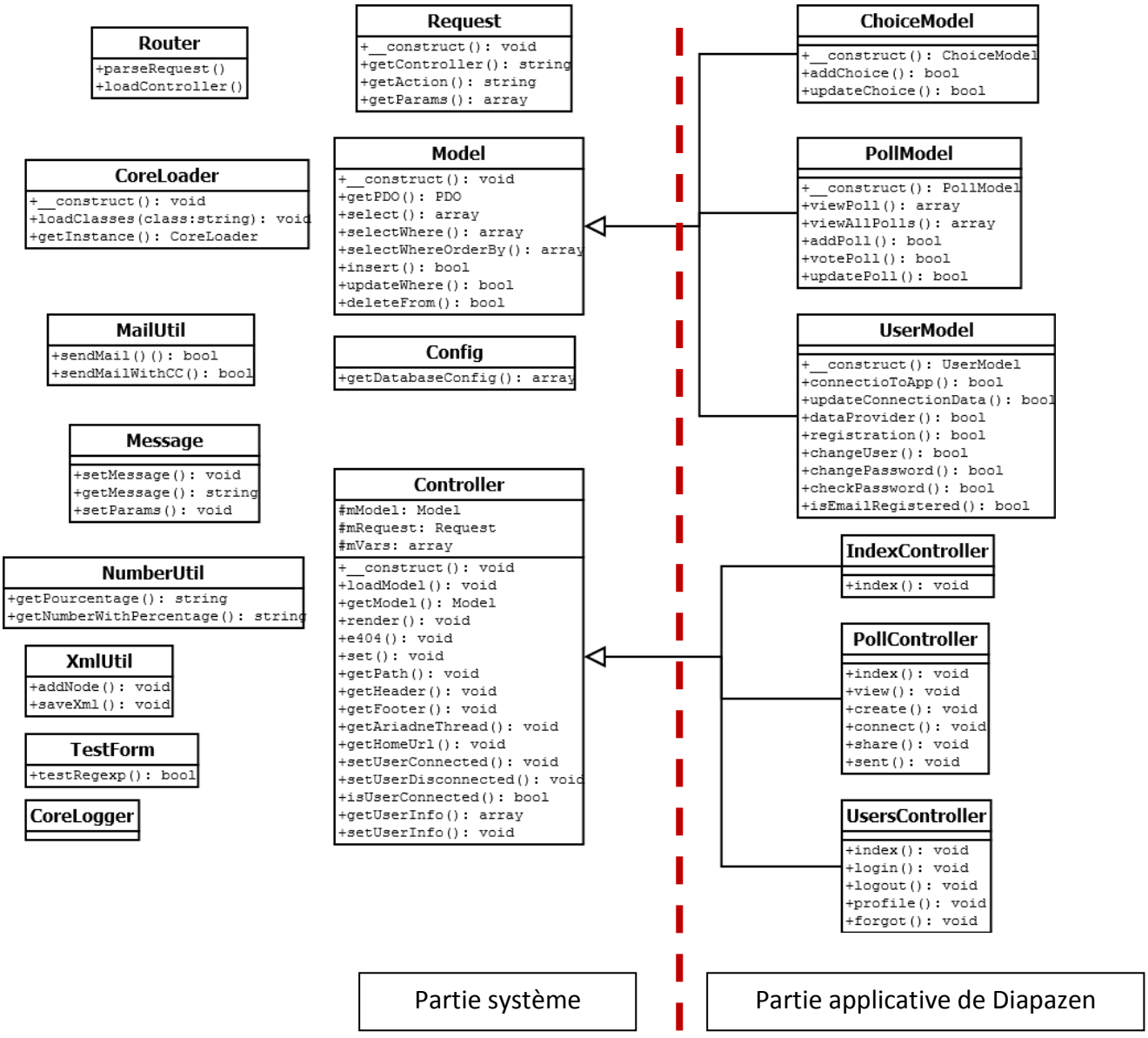
Au lieu d'utiliser un framework existant, nous avons préféré écrire le nôtre afin de mieux comprendre leurs fonctionnements. Pour cela nous nous sommes inspirés du Zend Framework dans une version fortement simplifiée. Le modèle de développement retenu est le modèle Open Source. Notre framework et l'application qui l'accompagne sont libres et gratuits, ils sont sous licence GNU GPL v3. Le langage choisi pour le framework est PHP5 Objet. L'interface utilisateur est écrite en HTML5 et CSS3.

Le code source complet ainsi que la documentation phpDoc sont disponibles sur le site github.com/diapazen.

I. Diagramme de classe UML

Nous avons choisi de développer un framework objet. C'est-à-dire qu'il suffit d'utiliser le mécanisme d'héritage des classes mères du système afin de réaliser l'application souhaitée. De plus le framework utilise le patron de conception Modèle-Vue-Contrôleur, ce qui permet une très bonne séparation des différentes couches de l'application. L'affichage est alors complètement séparé de la gestion des données.

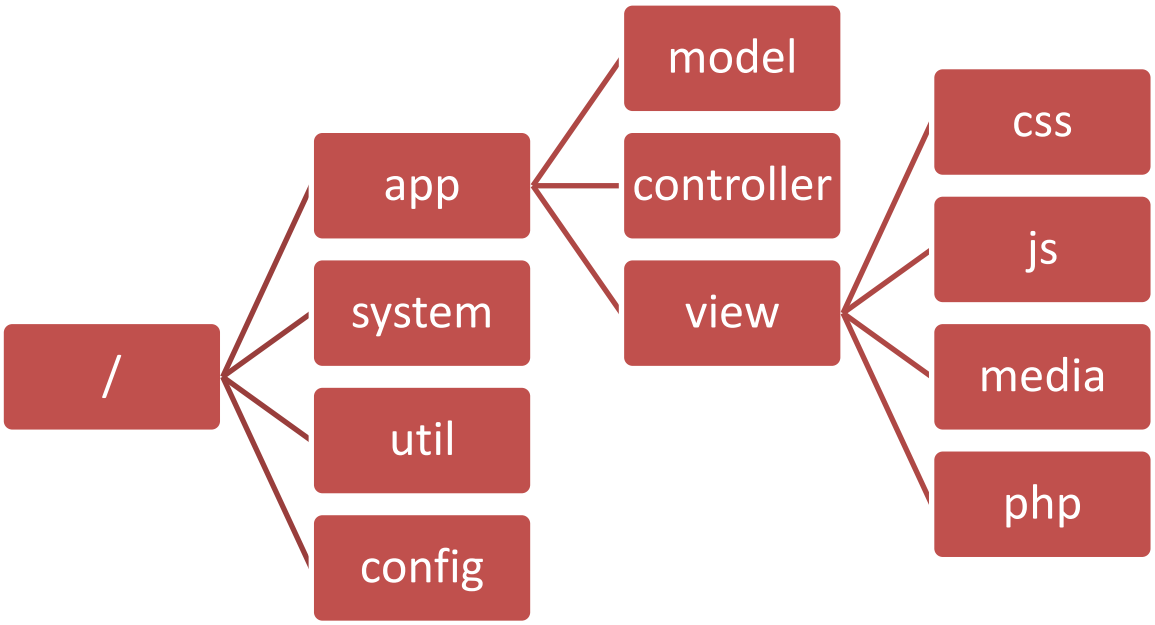
Ci-dessous le diagramme de classe UML volontairement simplifié. Nous allons détailler le fonctionnement des différentes classes par la suite.



II. Organisation interne du framework

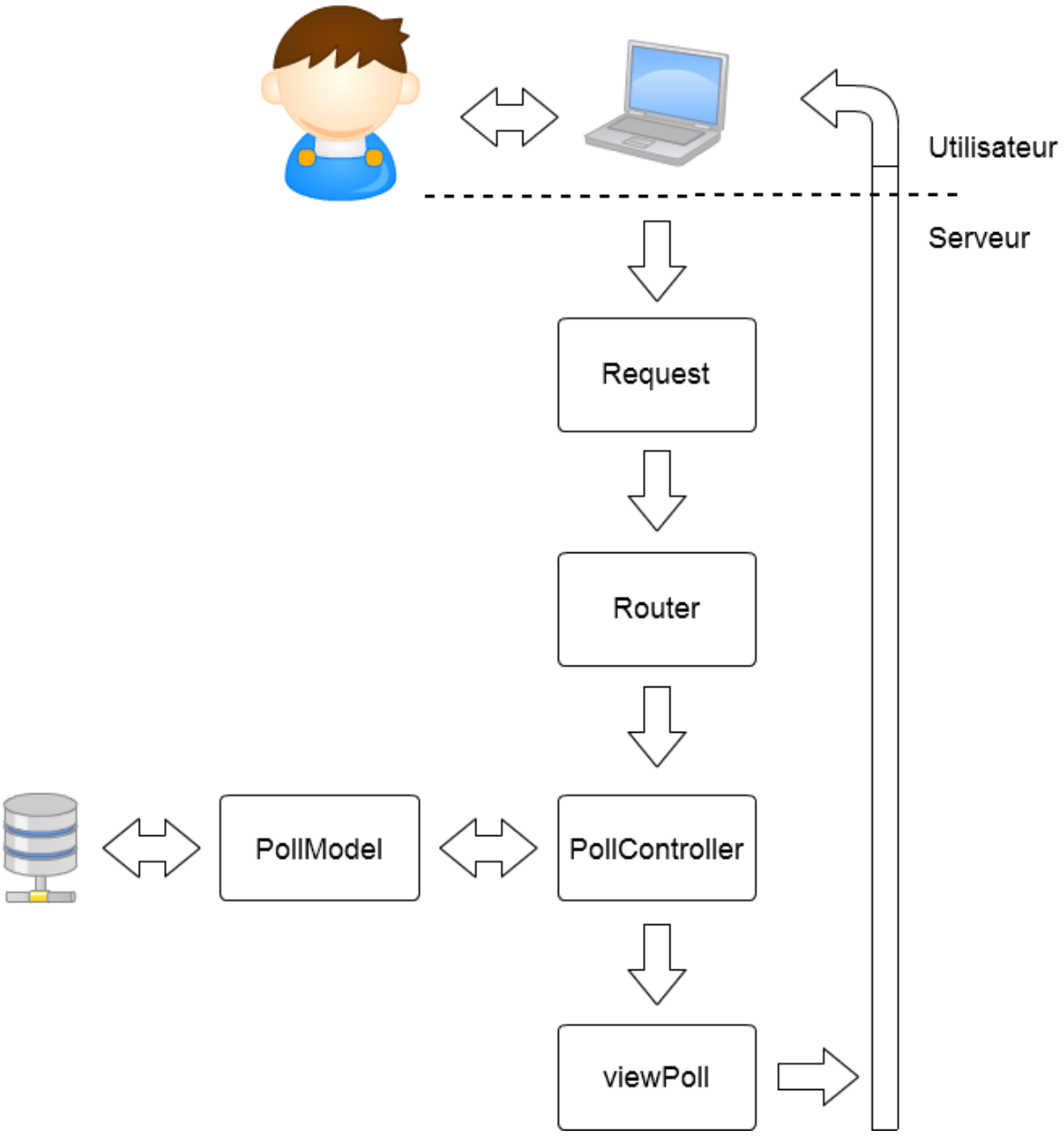
Les fichiers sources du framework sont dans le dossier « system ». Une application utilisant le framework possède ses fichiers dans le dossier « app ». La configuration du framework est dans le dossier « config ». Les classes utilitaires pouvant être reprises dans d'autres projets sont dans le dossier « util ».

Tout ce qui concerne l'application se trouve dans le dossier « app ». Le dossier « model » contient toutes les classes métier gérant les données de l'application. Le dossier « controller » contient les classes métier gérant les différentes pages du site. Enfin le dossier « view » qui contient l'ensemble des fichiers pour l'interface utilisateur.



III. Diagramme de flux

Afin de mieux comprendre le fonctionnement du framework, voici un exemple avec l'affichage d'un sondage sur Diapazen.



IV. Fonctionnement général

Imaginons que l'utilisateur clique sur l'url suivante :
http://diapazen.com/poll/view/85Fr45p2Ba

Sachant que le framework utilise la réécriture d'url sous la forme :
domaine.com/controller/action/param1/param2/…

Le framework va alors analyser l'url fourni par l'utilisateur, et grâce au routeur, charger le contrôleur correspondant à l'url. Ici il s'agit du « PollController » qui gère les sondages. Dans l'url, on peut remarquer le mot « view ». Cela signifie que l'action du « PollController » est l'affichage d'un sondage. On remarquera aussi que l'identifiant du sondage est le premier et unique paramètre de l'url.

Ce contrôleur peut faire appel à un modèle. Le modèle est une couche d'abstraction des données. Il s'occupe de la base de données en y faisant les requêtes nécessaire pour obtenir ou modifier des données. Dans notre exemple le « PollModel » fait toutes les requêtes nécessaires à la base de données pour la gestion des sondages.

Une fois les données récupérées de la base, le contrôleur va les envoyer à la vue. La vue reçoit alors les données et génère le code HTML/CSS qui sera affiché dans le navigateur de l'utilisateur. Ainsi dans notre exemple, le « PollController » après avoir récupéré les données du sondage à afficher provenant du « PollModel », va appeler la vue « pollView » qui se charge de générer la page web affichant le sondage.

V. Description détaillée

a. Request

La classe « Request » analyse l'url et stocke le nom du contrôleur à charger, son action à effectuer et la liste des paramètres optionnels possibles.

Ci-dessous la forme de l'url choisie :

http://domaine.com/controller/action/param1/param2/…

Pour illustrer le fonctionnement, voici un exemple :

Ainsi le contrôleur est « poll », l'action est « view » et il y a un paramètre « bdef6513f8 ».

b. Router

La classe Router est la première instanciée. Elle crée un objet Request qui lui permet de connaître le contrôleur à charger.

Par convention: si le contrôleur est par exemple « poll », alors la classe Router doit charger une classe nommé « PollController ». Si elle est inexistante, le routeur demande au contrôleur principal du framework d'afficher une erreur 404. En revanche, si la classe existe alors le routeur l'instancie et tente d'appeler la méthode correspondant à l'action de la classe Request, ici la méthode « view ». Si la méthode n'existe pas, alors une erreur 404 est renvoyée par le contrôleur principal du framework.

c. Controller

La classe « Controller » du framework est une classe majeure du framework. Elle permet de faire le lien entre les données et l'affichage du site.

Le principe est que chaque page du site possède son propre contrôleur. Ainsi la page À propos aura comme contrôleur « AboutController », la page d'accueil aura pour contrôleur « IndexController » et ainsi de suite. Tous les contrôleurs héritent du contrôleur principal du framework situé dans le dossier « system ».

Chaque page du site pouvant afficher différentes données selon une action passée dans l'url, ainsi tous les contrôleurs enfant doivent implémenter la méthode associé à l'action correspondante. Par exemple si l'action est « view » du contrôleur « PollController », alors ce dernier doit implémenter la méthode « view » comme suit :

```php
public function view($params = null)
{

}
```

Où *$params* est un tableau contenant la liste des paramètres optionnels de l'url.

Ci-dessous le code typique d'une méthode d'un contrôleur :

```php
public function view($params = null)
{
    // On charge le modèle qui gère les sondages
    $this->loadModel('poll');

    // On fait une requête vers la bdd
    $res = $this->getModel()->viewPoll();

    // On définit la variable maVar pour la vue
    $this->set('maVar', 'orange');

    // On appelle le rendu de la vue
    $this->render('pollCreation');
}
```

d. Model

Le modèle effectue toute la gestion des données. C'est-à-dire qu'il récupère, ajoute, modifie et supprime les données. Ici les données sont stockées dans une base de données MySQL.

La classe « Model » située dans le dossier « system » du framework, contient les méthodes principales de gestion de la base de données tel que :

- La sélection
- La modification
- L'ajout
- La suppression

Ces méthodes sont une couche d'abstraction aux requêtes SQL. Elles sont protégés contre la majorité des attaques SQL et XSS.

Pour utiliser ces méthodes, il est recommandé de créer une nouvelle classe avec le suffixe « Model ». Par exemple la classe qui gère les sondages s'appelle « PollModel ». Cette nouvelle classe doit hériter de la classe « Model » du framework.

Le code ci-dessous ajoute un vote à un sondage :

```
public function votePoll($choiceId, $value)
{
    $data = array('id' => 'NULL', 'choice_id' => $choiceId, 'value' => $value);
    return $this->insert($data, 'dpz_results');
}
```

Pour plus d'informations sur la signification des paramètres, reportez-vous à la documentation du code source.

e. View

Les vues sont de simples fichiers php avec essentiellement du code HTML.

Aucun traitement ou logique doit être fait dans la vue. Elle sert uniquement à présenter les données à l'utilisateur. En revanche des instructions simples comme « echo », des boucles et les méthodes des contrôleurs sont autorisés.

Ci-dessous un exemple d'une vue :

```
<?php $this->getHeader(); ?>

        <div id="content" class="about">
            <p class="title">À propos de Diapazen</p>
            <br>
            <p class="text">Diapazen permet de planifier rapidement
            <br>
            <p class="text">C'est un service libre et gratuit réalis
        </div>

<?php $this->getFooter(); ?>
```

On peut remarques les méthodes getHeader() et getFooter() qui sont définies dans le contrôleur principal. La méthode getHeader() charge le fichier « header.php », de même pour le footer.

Dans un contrôleur pour passer des informations à la vue il faut utiliser la méthode set(nomDeLaVariable, ValeurDeLaVariable) avant d'appeler le rendu. Ainsi dans notre vue une variable sera automatiquement créé selon le nom passé par la méthode set.

Exemple dans un contrôleur :

```
// On définit la variable maVar pour la vue
$this->set('maVar', 'orange');

// On appelle le rendu de la vue
$this->render('pollCreation');
```

Ainsi dans la vue pollCreation.php

```
<!-- du code HTML -->

<p>La couleur est: <?php echo $maVar; ?></p>

<!-- encore du code HTML -->
```

Ce qui affiche : `La couleur est: orange`

f. CoreException

La classe « CoreException » permet de lancer des exceptions personnalisées.

Cette classe hérite de la classe Exception et son fonctionnement reste le même. Elle permet aussi de journaliser automatiquement les exceptions provenant de la base de données.

g. CoreLoader

Le framework permet d'instancier très facilement les classes grâce à l'auto-chargement. Ainsi, il suffit juste d'instancier la classe, sans passer par des require ou include.

La classe CoreLoader, n'ayant besoin que d'une seule instance, elle respecte le patron de conception Singleton. De plus son instanciation se fait à l'initialisation du framework.

CoreLoader utilise la fonction spl_autoload de PHP.

Les divers dossiers sont renseignés.

```
set_include_path(get_include_path() . PATH_SEPARATOR . CONTROLLER_ROOT);
set_include_path(get_include_path() . PATH_SEPARATOR . MODEL_ROOT);
set_include_path(get_include_path() . PATH_SEPARATOR . UTIL_ROOT);
set_include_path(get_include_path() . PATH_SEPARATOR . WRITER_ROOT);
```

De même pour les extensions de fichiers

```
spl_autoload_extensions('.class.php, .php, .inc.php');
```

Enfin on spécifie quelle méthode doit être appelée pour le chargement d'une classe.

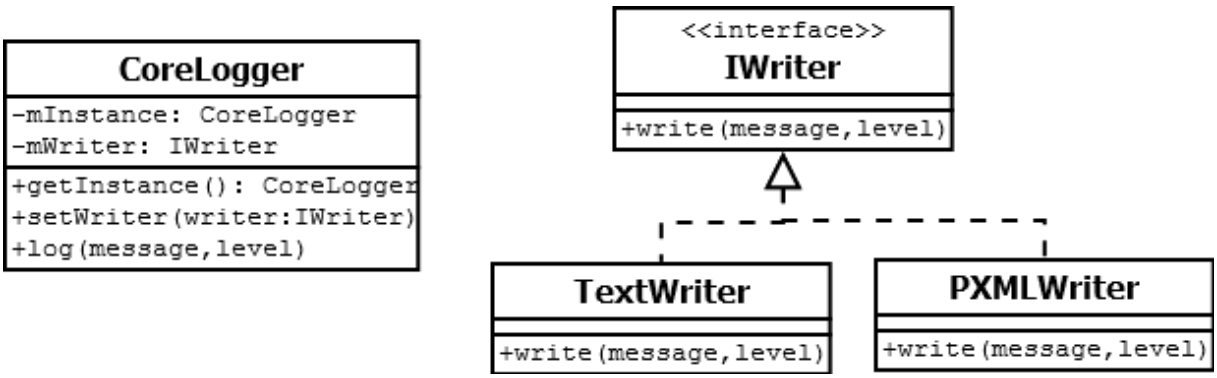
```
spl_autoload_register(array($this, 'loadClasses'), false);
```

La méthode loadClasse vérifie si le fichier existe dans l'un des dossiers renseigné. Si le fichier contenant la classe est introuvable, une exception 404 est lancée.

h. CoreLogger

Afin de pouvoir effectuer la maintenance du framework, un système de journalisation a été mis en place.

Diagramme UML du système de journalisation :



La classe CoreLogger respecte le patron de conception Singleton car elle doit être instanciée qu'une seule fois.

Ces méthodes publiques sont :

- getInstance() : récupère l'instance du système de journalisation.
- setWriter() : affecte une classe implémentant l'interface IWriter comme système de sauvegarde des logs.
- log() : Log le message.

Chaque classe implémentant l'interface IWriter peut avoir son propre fonctionnement. Une classe peut journaliser des informations sous forme texte et une autre sous forme XML. Pour ajouter un nouveau format de stockage des logs, il suffit d'implémenter l'interface IWriter.