# Projet III : Framework MVC2

Jean-Luc Chassaing, Damien Gaie, Adrien Salais

20 avril 2004

# Table des matières

# Chapitre 1

# Cahier des charges

## 1.1 Problématique

Le PHP est un langage créé spécialement pour afficher du contenu dynamique dans des pages web. Il est très simple à apprendre et à manipuler. Ce langage s'intègre dans le code HTML des pages web ce qui est très pratique pour créer rapidement de petites applications internet. Cette méthode semble efficace au premier abord, mais cela fini par poser un certain nombre de problèmes [CRAN00] :

- Il est difficile de réutiliser le code déjà écrit, puisqu'il a tendance à dépendre de la présentation.
- Le mélange présentation et code applicatif rend l'application difficile à maintenir et à étendre.
- Cette technique n'est pas adapté au développement d'applications professionnelles où le travail de graphisme est fait séparément du travail de développement.

Des qu'une application augmente de taille il devient donc nécessaire de séparer autant que possible la présentation du code. La présentation (HTML ou autres) ne doit contenir que du code de formatage de données et le code applicatif proprement dit (accès aux bases de données, calculs divers sur les données, etc...) doit être exécuté ailleurs. Il est aussi nécessaire de créer une architecture cohérente pour l'application, afin de faciliter la maintenance et de favoriser le travail en équipe.

Un framework est un ensemble de de scripts et d'outils fournissant un cadre commun de développement aux programmeurs, rendant les applications plus simple à comprendre et à maintenir.

Le framework devra proposer une architecture permettant de séparer le contenu de la présentation, et devra aussi apporter un certain nombre de so-

lutions permettant de régler les problèmes communément rencontrés dans le développement d'applications internet.

## 1.2 Séparation du contenu et de la présentation

### 1.2.1 Description succincte du modèle MVC2

Le modèle MVC : Modèle Vue Contrôleur [POSA96], propose une solution élégante au problème de séparation du contenu et de la présentation, en divisant l'application en trois parties. Le Modèle contient la logique l'application et son état (les données persistantes). La Vue représente l'interface utilisateur. Le Contrôleur reçoit les demandes des utilisateurs, les valide, exécute les méthodes du domaine et choisit la vue à afficher.

La logique métier (le modèle) est séparé de la présentation et peut donc être développée séparément et être réutilisable. La vue ne contient plus que du code de présentation et peut être presque entièrement écrite par une équipe de graphistes. Le contrôleur quand à lui, permet de valider ce qui est saisi par l'utilisateur, permettant ainsi de renforcer la sécurité.

La première version de ce modèle nécessite une multitude de contrôleurs (le contrôle des entrées est décentralisé) représentant autant de points d'entrée différents dans l'application, ce qui peut fragiliser l'architecture (code de contrôle redondant, difficulté de savoir ce que fait l'utilisateur, etc...).

La deuxième version du modèle n'autorise plus qu'un point d'entrée à l'application : un contrôleur frontal qui choisi l'action à effectuer en fonction des requêtes du client. C'est cette version, plus adaptée aux applications internet, qui devra être implémentée par le framework.

### 1.2.2 Détail de l'implémentation

Le framework Struts [GIRA01] [HUST02] fournira une piste de reflexion pour l'implémentation.

Le framework devra comporter un contrôleur principal qui sera appelé à chaque requête. Il sera constitué d'une classe et d'un fichier de configuration XML permettant de paramétrer finement l'application.

Après avoir validé les paramètres envoyés par le client, le contrôleur principal devra choisir un contrôleur spécifique correspondant à la demande du client. Ce contrôleur spécifique (qui sera nommé par la suite "action") sera chargé d'exécuter les méthodes du domaine et renverra une série de données qui devront êtres affichées par la vue. Une action sera constituée d'une classe contenant le code à exécuter et d'un descripteur XML permettant de définir la vue à exécuter ainsi qu'un certain nombre de variables de configuration.

Une fois l'action effectuée une vue se chargera de formater les données pour l'affichage.

## 1.3 Autres fonctionnalités

### 1.3.1 Architecture orientée cas d'utilisations

Une mauvaise architecture interne peu ruiner les bénéfices apportés par le modèle MVC2. Une application mal structurée est très difficile à maintenir, même si elle est basée sur le modèle MVC2.

L'architecture retenue devra être orientée cas d'utilisations. Les actions devront être rassemblées dans des dossiers en fonction du cas d'utilisation auquel elles appartiennent. Un dossier correspondra donc à un cas d'utilisation et ne contiendra qu'un petit nombre d'actions. Si l'on veut par exemple modifier une action de récupération des informations boursières, il suffira de chercher le dossier correspondant au cas d'utilisation "InformationBousieres" et chercher l'action correspondante.

Le framework devra s'efforcer d'encourager ce type d'architecture interne et faire en sorte qu'une mauvaise structuration soit difficile à mettre en place.

### 1.3.2 Sécurité

Dans l'édition 2004 du *The ten most critical web application security vulnerabilities* [OWAS04], au moins trois des failles recensées viennent d'erreurs de programmations au niveau applicatif :

1. Cross Site Scritpting
2. Problèmes d'injection
3. Mauvaise gestion des erreurs

Le framework devra prendre en compte ces problèmes et fournira des solutions pour éviter que des erreurs de programation mènent à une faille. Les considérations de sécurité devront être la préocupation principale du framework et, si cela est nécessaire, la sureté de fonctionnement sera privilégiée face aux performances.

### 1.3.3 Validation des données entrées par les utilisateurs

Dans une application internet, les principales sources de bugs et de failles de sécurité proviennent des données envoyées par les clients. En aucun cas, il ne faut faire confiance à ces données et elles doivent absolument passer par un processus de validation avant d'être utilisées.

Le framework devra proposer une gestion automatisée de cette validation. Les données issue des utilisateurs devant être traitées par une action devront être décrites dans le descripteur XML de cette action. Elles seront décrites de manière précise (leur type,leur provenance, leur valeur minimale et maximale etc..) et seront validées par une fonction ou une expression régulière avant l'exécution de l'action. Les données non décrites dans le descripteur de l'action ne devront pas être accessible par l'action, afin de limiter les erreurs de programmation. De plus, les caractères spéciaux de ces données devront êtres encodés en charactères HTML afin de limiter les risques de faille de sécurité.

Comme il n'est pas possible de définir de façon exaustive les fonctions et les expressions régulières permettant la validation. Elles devront donc pouvoir être paramétrées dans un fichier de configuration XML.

### 1.3.4 Connection aux sources de données

Les classes du modèle qui seront exécutées par les actions vont probablement accéder à des données persistantes en provenance de bases de données. Si la gestion de la connection n'est pas gérée de facon centralisée, c'est à dire si chaque classe contient les informations de connection, la migration vers une autre source de données est très complexe. Il faut stocker dans un endroit unique les données nécessaires à la connection (comme l'URI du serveur, l'utilisateur etc...).

Le framework devra donc permettre de configurer des accès à différentes sources de données. La configuration devra se faire simplement en renseignant un paramètre du fichier de configuration du framework.

# Chapitre 2

# Spécifications fonctionnelles

## 2.1 Avant-propos

Le framework propose une implémentation du modèle MVC2 et reprend certains concepts du framework Struts. Struts écrit en java fourni une très bonne implémentation du modèle MVC2, il pourrait donc être tentant de vouloir copier trait pour trait le fonctionnement de Struts. Mais le langage PHP est très différent de java et vouloir copier Struts peut se révéler peu adapté à PHP poser de nombreux problèmes. Le framework n'est donc absolument pas une copie conforme de Struts.

## 2.2 Le modèle MVC2

Le diagramme de collaboration de la figure 2.1 présente le fonctionnement général du framework.

Une requête est traitée de la manière suivante :

1. L'utilisateur envoie un événement au contrôleur (en appuyant sur un bouton par exemple).
2. Le contrôleur identifie la requête, la valide et dirige celle-ci vers le contrôleur spécialisé.
3. Le contrôleur spécialisé demande l'exécution de méthodes au modèle.
4. Le modèle exécute les méthodes demandées et renvoie des données au contrôleur spécialisé.
5. Le contrôleur spécialisé nommé action, insère et modifie des données dans le *dataSet* et retourne une réponse au contrôleur principal.
6. Le contrôleur principal sélectionne la vue à appeler en fonction de la réponse retournée par l'action .

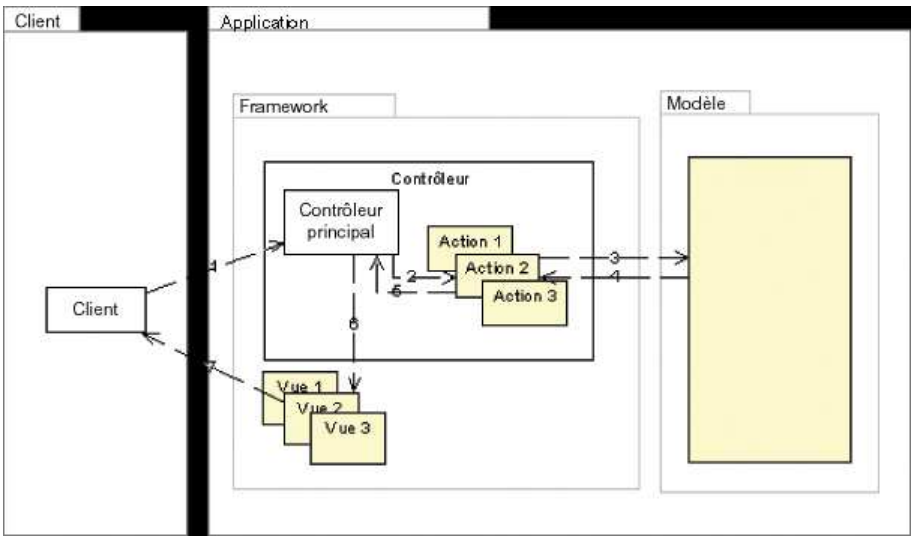


FIG. 2.1 – Modèle MVC 2

7. La vue se charge de l'affichage.

## 2.3 Le contrôleur principal

Le contrôleur principal est géré par une classe nommée *FrontControler*. C'est le pivot du modèle MVC2 car il gère tout le cycle de vie d'une requête en vérifiant les entrées, appelant les contrôleurs spécialisés ainsi que les vues.

Après avoir choisi l'action à effectuer en analysant les paramètres envoyé par l'utilisateur, le contrôleur vérifie que l'action possède un descripteur puis charge celui-ci. Si l'action est de type forward (Tableau 2.4.2.2), le forward défini dans la vue est directement appelée, sinon le contrôleur vérifie que la classe d'action définie dans le descripteur existe, charge le fichier de classe et crée une instance de celle-ci. Ensuite l'action est effectuée et en fonction de ce qui est retourné, le contrôleur appelle une vue ou exécute une nouvelle action. Finalement, le contrôleur retourne le contenu renvoyé par la vue, ou un message d'erreur si quelque chose c'est mal passé.

## 2.4 Le contrôleur spécialisé

### 2.4.1 Description

Les contrôleurs spécialisés sont constitué d'une classe étendant la classe *Action* et d'un descripteur XML. L'action reçoit des données en entrée (voir chapitre 2.6) et appelle des objets du modèle. Il faut bien se souvenir que les actions font partie du contrôleur et donc ne doivent pas effectuer directement des calculs relevant du modèle. Les objets du modèle retournent ensuite des données, ce qui permet à l'action de modifier des données du *dataSet*. Enfin l'action indique au contrôleur principal son état de sortie en retournant un objet *Forward*. Cet objet peut contenir soit le nom d'une vue, soit le nom d'une autre action à exécuter. La figure 2.2 présente un exemple de classe action.

```
<?php
require_once (FRAMEWORK_DIR."/core/Action.class.php");
require_once ("model/HelloModel.class.php");

class HelloAction extends Action{

    function execute(&$dataSet,$mapping){
        $model = new HelloModel();
        //execution de methode du modèle
        $hello = $model->doStuff($dataSet->request->get("hello_id"));

        //insertion d'une donnée dns le dataset
        $dataSet->request->set("hello",$hello);

        //renvoi d'un forward
        return $mapping->findForward("SUCCESS");
        }
    }
}
```

FIG. 2.2 – Fichier de classe d'une action

### 2.4.2 Configuration

Les actions sont regroupées dans des répertoires correspondant au cas d'utilisation auquel elles appartiennent. Les descripteurs des actions sont regroupés dans un fichier commun, situé à la racine du répertoire du cas d'utilisation et nommé configuration.xml.

La figure 2.3 montre un exemple d'un fichier de configuration d'un cas d'utilisation. Les chapitres suivants décriront les différents éléments XML de ce fichier.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<conf>
    <global-forwards>
         <forward name="ERROR"  path="views/error.php"/>
    </global-forwards>
    <action_map default="index" >
        <action name="index" forward="FORM">
            <forward name="FORM" path="views/helloFormulaire.php"/>
        </action>

        <action name="helloWorld" class="HelloWorldAction">
           <authentification right="user"/>
           <forward name="SUCCESS" path="views/hello.php"/>
           <forward name="FORM" path="views/helloFormulaire.php"/>
            <input use-token="true" input-forward="FORM">
                <field name="foo" scope="REQUEST" null="false" content="ALNUM"
                                        size_min="4" size_max="33"/>
                <field name="bar" scope="SESSION" null="false" content="DIGIT"
                                        val_min="4" val_max="33"/>
           </input>
        </action>

    </action_map>
</conf>
```

FIG. 2.3 – Fichier de configuration d'un cas d'utilisation

#### 2.4.2.1 L'élément *action-map*

L'élément *action-map* contient la définition de toutes les actions disponibles pour un cas d'utilisation donné. il peut contenir zéro ou plusieurs éléments *action* et eventuellement un élément glogal-forward(voir chapitre 2.5) :

```
<!ELEMENT action-map (glogal-forward?,action*)>
```

Le tableau 2.1 décrit les attributs disponibles pour l'élément *action-map*.

| Nom | Requis | Description |
|---|---|---|
| default | oui | Défini l'action à exécuter par défaut si le cas d'utilisation est appelé sans préciser d'action |

TAB. 2.1 – Attributs de l'élément *action-map*

#### 2.4.2.2 L'élément *action*

L'élément *action* permet de décrire précisément le comportement d'une action. Il peut contenir un certain nombre d'élément fils :

```
<!ELEMENT action (authentification?,input-vars?,forward*)>
```

Le tableau 2.2 décrit les attributs disponibles pour l'élément *action*.

| Nom | Requis | Description |
|---|---|---|
| name | oui | Défini le nom de l'action. C'est ce nom qui sera passé en paramètre dans l'url |
| forward | non | Si cet attribut est renseigné, aucune action ne sera exécutée, et le forward dont le nom est défini dans cet attribut est directement appellé. |
| class | non | Classe chargée d'exécuter l'action demandée. Si l'attribut forward est renseigné, cet attribut ne sera pas pris en compte |

TAB. 2.2 – Attributs de l'élément *action*

#### 2.4.2.3 L'élément *authentification*

L'élément *authentification* permet de définir des droits d'accès pour cette action. L'authentification n'est pas une fonctionnalité décrite dans un framework MVC2, mais elle est si indispensable dans ue application internet qu'elle à été incluse dans le framework.

Le problème est que les méthodes d'authentification sont multiples et il est impossible de proposer une solution universelle. Pour cela, l'authentification est implémentée dans le framework sous la forme d'un cas d'utilisation et d'un classe permettant de vérifier les droits du client. Le fait que ce soit un cas d'utilisation presque comme les autres permet à un développeur de changer simplement de méthode d'authentification. Par défaut la solution proposée s'appuie sur mysql.

Le tableau 2.2 décrit les attributs disponibles pour l'élément *authentification*.

| Nom | Requis | Description |
|---|---|---|
| name | oui | Défini une liste de droits séparés par des virgules. L'action ne sera éxécutée que si l'utilisateur possède l'un ou l'autre de ces droits. |

TAB. 2.3 – Attributs de l'élément *authentification*

#### 2.4.2.4 L'élément *forward*

L'élément *forward* permet d'associer un nom symbolique à une vue ou une action. (voir le chapitre sur les vues pour plus de détails //TODO) Le tableau 2.4 décrit les attributs disponibles pour l'élément *forward*.

| Nom | Requis | Description |
|---|---|---|
| name | oui | Nom qui permettra à une action d'indiquer qu'il faut afficher cette vue. |
| path | oui | Vue associée à ce forward. le chemin d'accès est relatif au répertoire du cas d'utilisation. |
| use-case | non | Si le forward défini le nom d'un action, cet attribut permet de d'indiquer dans quel cas d'utilisation se trouve l'action. Si l'attribut path est défini, cet attribut sera ignoré. |
| action | non | Défini une action. Si l'attribut path est défini, cet attribut sera ignoré. |

TAB. 2.4 – Attributs de l'élément *forward*

#### 2.4.2.5 L'élément *input*

L'élément *input* contient la description de toutes les données qui seront validées puis passées aux actions. il peut contenir zéro ou plusieurs éléments *data* :

```
<!ELEMENT input (data*)>
```

Le tableau 2.5 décrit les attributs disponibles pour l'élément *input*.

| Nom | Requis | Description |
|---|---|---|
| input-forward | non | Indique quel *forward* executer si il y a des erreurs de validation. Si cet attribut n'est pas renseigné, l'action se déroule normalement, même si des erreurs de validation existent. Ce sera à l'action de gérer ces erreurs. |
| use-token | non | Indique si la validation doit prendre en compte un token pour éviter le rejeu et les attaques de type *Cross-Site Request Forgeries* (voir chapitre 2.8.2) . |

TAB. 2.5 – Attributs de l'élément *input*

#### 2.4.2.6 L'élément *data*

L'élément *data* décrit les règles de validation d'une une variable devant être passée à l'action.

Le tableau 2.6 décrit les attributs disponibles pour l'élément *data*.

## 2.5 Les vues

### 2.5.1 Description

Dans le framework, les vues sont uniquement utilisées pour formatter des données en vue de l'affichage, elle ne font donc que lire et afficher correctement des données issues du *dataSet*. Les vue sont donc de simples fichiers PHP qui vont contenir du HTML et un peu de code PHP . Le code PHP requis est minimal et doit se résumer à des lectures de variables et des boucles simple.

### 2.5.2 Utilisation d'un langage de template

Un langage de template défini une série de balises qui permettent d'interagir avec le framework sans avoir à insérer du code dans les vues.

Struts fournit un certains nombre de librairies de *tags* qui permettent d'éviter d'insérer du code java dans les pages JSP. Cette approche est très efficace en java et ne pose pas de problèmes de performances car les pages *jsp* sont pré-compilées en *servlets*. Il n'en va pas de même avec PHP.

Tout d'abord, les pages php ne sont pas précompilées et le fait d'insérer des balises spéciales dans la vue nécessite un traitement lourd à chaque requêtes : il faut en effet effectuer une traduction de ces balises en code PHP avant de pouvoir exécuter effectivement ces vues. La définition d'un langage de template peut donc poser un problème de performance [BIFA02] .

| Nom | Requis | Description |
|---|---|---|
| name | oui | Nom de la variable. |
| scope | oui | Portée de la variable. La valeur de la variable sera recherchée soit dans les cookies soit dans les variables de session, soit dans les variables envoyées par le navigateur (les variables get et post sont regroupées en une seule "portée"). Donc trois valeurs sont disponibles : "COOKIES", "SESSION", "REQUEST" |
| null | non | Indique si la variable peut être vide. La valeur par défaut est "false" (la variable ne peut pas être vide) |
| content | non | Nom de la méthode de validation à appliquer. Ces méthodes de validation sont définies dans le fichier validation.xml |
| size-min | non | Nombre de caractères minimum possible pour la variable |
| size-max | non | Nombre de caractères maximum possible pour la variable |
| val-min | non | Valeur minimum de la variable si celle-ci est de type numérique |
| val-max | non | Valeur maximum de la variable si celle-ci est de type numérique |

TAB. 2.6 – Attributs de l'élément *data*

```
Name: {$name}<br>

{section name=curuser loop=$user}
    <tr>
        <td bgcolor="#FFFFFF" align="center">{$user[curuser].id}
</td>
        <td bgcolor="#FFFFFF">{$user[curuser].name}</td>
        <td bgcolor="#FFFFFF">
<a href="mailto:{$user[curuser].email}">
{$user[curuser].email}
</a>
</td>
        <td bgcolor="#FFFFFF" align="center">
{if $user[curuser].banned}X{else}&{/if}
</td>
    </tr>
{/section}
```

FIG. 2.4 – Exemple de vue utilisant Smarty

La complexité de java le rend peut adapté au pages jsp, rendant le code difficile à maintenir, et cela qui justifie l'utilisation de librairies de tags. Mais le langage PHP a été spécialement conçu pour être intégré dans des pages html et il est très simple à utiliser. Un langage de template n'apporte donc pas grand choses à PHP. La figure 2.4 présente un exemple de template Smarty[1] (un des systèmes de templates les plus avancé de PHP), tandis que la figure 2.5 montre son équivalent en php. On voit bien qu'un tel langage ne simplifie pas vraiment la création de vues et nécessite l'apprentissage d'une syntaxe spéciale.

Pour ces raisons, aucun langage de template n'a été défini dans le framework. Les vues utilise du PHP "basique" et un certains nombre de fonctions simplifiant l'accès aux données leurs sont accessibles.

### 2.5.3 Configuration

Les vues sont liées aux actions par les paramètres *forward* contenus dans les descripteurs de chaque action. Il est aussi possible de définir des *forwards* globaux qui forward seront disponibles pour toutes les actions. Les chapitres suivants décriront le paramétrage de tels *forward*.

#### 2.5.3.1 L'élément *global-forward*

L'élément *global-forward* permet de lier des noms symboliques à des vues ou à des actions et ce, au niveau du framework. Si une action redéfinit un *forward* qui existe déjà dans un global-forward, c'est le *forward* local qui sera utilisé. Cet élément peut contenir zéro ou plusieurs éléments *forward* :

```
<!ELEMENT global-forward (forward*)>
```

#### 2.5.3.2 L'élément *forward*

L'élément *forward* permet d'associer un nom symbolique à une vue ou à une action

Le tableau 2.4 décrit les attributs disponibles pour l'élément *forward*.

## 2.6 Le système de validation

### 2.6.1 Description

Lors de l'exécution d'une requête, un certain nombre de variables est disponible. Ces données proviennent soit de la requête utilisateur (se sont les variables

[1] `http://smarty.php.net`

| Nom | Requis | Description |
|---|---|---|
| name | oui | Nom qui permettra à une action d'indiquer qu'il faut afficher cette vue. |
| path | non | Vue associée à ce forward. le chemin d'accès est relatif au répertoire racine du framework (la ou se situe le fichier index.php). |
| use-case | non | Si le forward défini le nom d'un action, cet attribut permet de d'indiquer dans quel cas d'utilisation se trouve l'action. Si l'attribut path est défini, cet attribut sera ignoré. |
| action | non | Défini une action. Si l'attribut path est défini, cet attribut sera ignoré. |

TAB. 2.7 – Attributs de l'élément *forward* (dans un élément *global-forward*)

```
Name:<?=$name;?><br>

<?php foreach($this->get('users') as $user) { ?>
    <tr>
        <td bgcolor="#FFFFFF" align="center"><?=$user['id'];?>
</td>
        <td bgcolor="#FFFFFF"><?=$user['name'];?></td>
        <td bgcolor="#FFFFFF"><a href="mailto:
<?=$user['email'];?>"><?=$user['email'];?></a>
</td>
        <td bgcolor="#FFFFFF" align="center">
<?=($user['banned'] ? 'X' : '&');?>
</td>
    </tr>
<?php } ?>
```

FIG. 2.5 – Exemple de vue utilisant php

issues des GET, des POST et des des cookies ), soit de la session en cours. Les actions et les vues de l'application peuvent normalement accéder à ces données via des variables *superglobales* nommées $_GET, $_POST, $_SESSION, $_COOKIES. Mais les données de ces variables n'ont absolument pas été vérifiées et validées et les utiliser tel quel peut se révéler dangereux.

La notion de *dataSet* utilisée dans le framework propose un mecanisme pour manipuler et valider ces données. Les actions et les vues ne manipulent plus des données issues des variables *superglobales* mais les variables issues du dataSet. Les données du dataSet sont validées et considérées fiables, ainsi les actions et les vue n'ont pas à se préocuper de la validité des données qu'elle manipulent.

Les données dataSet sont de trois types :

1. les données *request* qui n'existent que durant l'exécution de la requête ;
2. les données persistantes *cookie* stockées du coté client ;
3. les données *session* stockées du coté serveur et détruites à la fin de la session.

Ces trois types de données sont stockés dans trois objets distincts (les objets request,session,cookie) du dataset. Par exemple, pour accéder a une donnée de session le dode suivant est éxécuté :

```
$dataSet->session->get('maVariable');
```

Pour ecrire dans une variable de session le code suivant est utilisé :

```
$dataSet->session->set('maVariable','une valeur');
```

En plus de ces données, le dataSet fournit un objet d'erreur qui peut être utilisé pour envoyer des erreurs affichables par les vues.

Ce système de dataSet permet de gérer de façon simple toutes les données disponibles durant le cycle de vie d'une requête. Ce système sert juste à renvoyer des erreurs à l'utilisateur et n'est pas utilisé en interne par le framework qui utilise un système de gestion d'erreurs plus puissant.

Dans ce système, les variables issues d'un GET ou d'un POST (au sens HTTP) ne sont pas différentiées car elle sont considérées comme des varaibles de portée *request*. Cela signifie qu'il ne faut pas qu'une variable d'un GET ai le même nom qu'une variable de POST, pour une requête donnée. En cas de conflit de nom, c'est la contenu de la variable POST qui sera conservé. Cela rend le site plus vulnérables aux attaques *Cross-Site Request Forgeries*, mais la séparation des GET et des POST n'est pas une solution efficace contre ce type d'attaque et le framework fourni une autre solution (voir chapitre 2.8.2.3).

Les données issues des requêtes utilisateur doivent absolument être validées avant d'être disponibles aux actions. Le framework fournit une classe *Validator* permettant de valider de façon aisée les données situées dans le dataSet.

Pendant l'initialisation du dataSet, celui-ci recoit les données décrites dans le descripteur de l'action à effectuer. En même temps, il passe à un objet *Validator* les règles de validation de ces données. une fois le données copiée, celle ci sont validées par le *Validator*. Si des données ne correpondent pas aux règles de validation elles sont suprimées et une erreure de validation est générée

Si des erreurs de validation sur des données de type *request* sont trouvées et que l'action à un attribut input-forward renseigné (chapitre 2.4.2.2), l'action n'est pas exécuté et le *forward* nommé dans input-forward est appelée, ce qui permet de revenir automatiquement à un formulaire si celui-ci n'a pas été bien rempli.

### 2.6.2 Configuration

La configuration du système de validation se fait à deux endroits. La définition des données à valider, ainsi que les règles de validations, sont définies dans le descripteur de chaque action(chapitre 2.4.2.6 ). La définition des méthodes permettant de valider précisément le contenu des données se fait dans le fichier de configuration du framework. La figure 2.6 montre un fragment du fichier de validation fourni par défaut avec le framework.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<validation>
     <method name="DIGIT"  type="ereg" value="^([[:digit:]])+$"
               error="ne doit contenir que des chiffres"/>

     <method name="NUM"  type="function" value="is_numeric"
               error="doit être un nombre"/>

     <method name="ALNUM"  type="function" value="ctype_alnum"
               error="ne doit contenir que des lettres et des nombres"/>

     <method name="ALPHA"  type="function" value="ctype_alpha"
               error="ne doit contenir que des lettres"/>
</validation>
```

FIG. 2.6 – Configuration des méthodes de validation

En dehors de l'élément nommé *validation* ne comprend que des éléments *method*. Ils permettent de définir des fonctions ou des expressions régulières qui permettrons de vérifier la validité des données du *dataSet*.

Le tableau 2.8 décrit les attributs disponibles pour l'élément *method*.

| Nom | Requis | Description |
|---|---|---|
| name | oui | Défini le nom de la méthode. C'est ce nom qui sera utilisé dans l'attribut *content* des éléments *data* dans les descripteurs de l'action |
| type | oui | Indique si la méthode est une fonction ou une expression régulière |
| value | oui | Nom de la fonction à exécuter ou valeur de l'expression régulière à appliquer. |
| error | oui | Message d'erreur à retourner en cas d'erreur de validation |

TAB. 2.8 – Attributs de l'élément *method*

## 2.7 Accès aux données

### 2.7.1 Description

Pour éviter au modèle d'être trop dépendant des sources de données, il ne faut pas que le modèle connaisse et manipule les informations de connections. Ces informations sont donc définies dans le fichier de configuration principal (figure 2.7). Le framework propose une classe *DatasourceFactory* qui permet aux objets du modèle de récupérer un objet de source de données en indiquant simplement son nom. Le chapitre suivant décrira les différents éléments de confi-

```
<datasource name="MYSQL" class="MysqlDatabase" >
    <URI value="localhost"/>
    <user-name value="myUser"/>
    <password value="myPassword"/>
    <base value="CNAM"/>
</datasource>
```

FIG. 2.7 – Exemple de définition d'une source de donnée

guration d'une source de données.

### 2.7.2 Configuration

#### 2.7.2.1 L'élément *datasource*

L'élément *datatasource* défini de manière globale la source de donnée. Il possède un certain nombre d'éléments fils :

```
<!ELEMENT datasource (URI,user-name,password,base)>
```

Le tableau 2.9 décrit les attributs disponibles pour l'élément *datasource*.

| Nom | Requis | Description |
|---|---|---|
| name | oui | Nom de la source de données, c'est avec ce nom que les objets du modèle récupéreront un objet de source de données |
| class | oui | Nom de la classe qui sera chargée de gérer la connection et l'interrogation des sources de données |

TAB. 2.9 – Attributs de l'élément *datasource*

#### 2.7.2.2 L'élément *URI*

L'élément *URI* permet d'indiquer l'adresse du serveur gérant la source de données. Il n'a qu'un attribut nommé *value* qui reçoit l'adresse.

#### 2.7.2.3 L'élément *user-name*

L'élément *user-name* permet d'indiquer le nom d'un utilisateur ayant le droit de se connecter à la source de données. Il n'a qu'un attribut nommé *value* qui reçoit le nom.

#### 2.7.2.4 L'élément *password*

L'élément *password* permet d'indiquer le mot de passe qui, associé au nom d'utilisateur, permettra de s'authentifier auprès de la source de données. Il n'a qu'un attribut nommé *value* qui reçoit le mot de passe.

#### 2.7.2.5 L'élément *base*

L'élément *base* permet d'indiquer le nom de la base qui contient les données de l'application. Il n'a qu'un attribut nommé *value* qui reçoit le nom de la base.

## 2.8 Fonctionnalités liées à la sécurité

Les problèmes de sécurité des applications Web seront traitées en détails dans un autre document. Ce chapitre se contente de décrire un certains nombre fonctionnalités du framework qui ont été créées pour contrer certaines failles.

### 2.8.1 Problèmes des sessions

Le protocole HTTP est sans état. Les sessions permettent d'émuler un mode connecté en utilisant un identifiant de sessions transmis à chaques requêtes. Il existe un grand nombre d'attaques se basant sur les sessions, l'indentifiant peut

être volé de nombreuses façons différentes et ne suffit donc pas à garantir une bonne sécurité.

Pour limiter les attaques basées sur les sessions et augmenter par la même la sécurité, le framework attache l'adresse IP du client à l'identifiant de session. Quand une session est créée pour la première fois, l'adresse IP du client est stockée dans une variable de session. Ensuite , à chaque requête, l'adresse IP du client est vérifiée avec celle stockée dans la session, et si elles diffèrent, la session est détruite. Ainsi , le vol d'identifiant de session est rendu plus difficile.

Pour sécuriser efficacement une session il faudrait, en plus de vérifier l'adresse IP, généré un nouvel identifiant de session après chaque requête mais en PHP, il serait trop coûteux de faire une telle chose...

### 2.8.2 Le principe de *token*

#### 2.8.2.1 Fonctionnement

Les *tokens* sont utilisés pour régler certains problèmes qui peuvent survenir au niveau des formulaires.

Pour gérer les *tokens*, le framework utilise un *nonce* : quand le client demande un formulaire, une valeur aléatoire (*nonce*) est générée et stockée dans une variable de session. Cette valeur est aussi incluse dans un champs caché du formulaire ou dans l'url de l'attribut *target* de la balise HTML *post* . Quand le formulaire est posté, le framework vérifie que la valeur contenue dans le champ du formulaire est la même que celle stockée dans la session.

Les chapitres 2.8.2.2 et 2.8.2.3 décriront les problèmes pouvant être réglés par les tokens.

#### 2.8.2.2 Prévention du *double post*

Quand le réseau est chargé, et que l'on clique sur le bouton POST d'un formulaire, le formulaire peu rester affiché un certains temps avant que la page ne change. Pendant ce laps de temps, l'utilisateur, pensant que son action n'a pas été prise en compte, peut être tenté de cliquer une seconde fois sur le bouton. Il en résulte un second envoi du même formulaire qui risque d'être traité une seconde fois par le système et cela peut éventuellement compromettre l'intégrité des données de l'application. Ce problème est généralement appelé *double post*.

Un token permet de régler ce problème : lors du premier traitement de la requête, les nonces vérifiés sont égaux, le nonce est donc détruit et l'action est effectuée. Si un *double post* survient, le nonce ayant été détruit lors du premier post, une erreur de validation sera générée et l'action ne sera pas exécutée. Le client obtiendra une erreur probablement un peu déroutante, mais l'intégrité du système sera préservée.

Le token rend donc impossible le "rejeux" d'un formulaire.

#### 2.8.2.3 Préventions des attaques de type *CSRF*

Les attaques de type *Cross-Site Request Forgeries* sont peu connues et difficiles à éviter. Elles consistent à faire exécuter des requêtes par un client habilité. Ces requêtes sont valides et donc passent le processus de validation. L'exploitation de la faille peut se faire par l'envoi d'un mail à l'utilisateur contenant un lien vers une requête de l'application. Si le client habilité est authentifié et clique sur le lien, alors la requête sera effectuée avec ses propres privilèges, ce qui peut être désastreux.

Comme on ne peut pas forcer le client à ne pas cliquer sur un lien, il faut fournir une solution efficace au niveau du framework. Le framework propose de protéger les requêtes nécessitant des données en provenance d'un formulaire. Pendant l'étape de validation, le framework peut vérifier que les données proviennent bien d'un formulaire de l'application, comblant ainsi la faille CSRF.

Un token se révèle très adapté à ce type de vérification. Lors de la vérification des nonces, si les deux valeurs sont égales, alors la provenance des données est assurée car le formulaire à forcément été généré par le framework (qui est le seul a pouvoir générer des nonces valides). La valeur du nonce change à chaque appel du formulaire, il est donc très difficile de forger un faux formulaire et de le faire exécuter par le client. Le nonce doit avoir une courte période de validité, rendant encore plus difficile la création d'un faux formulaire.

Il ne faut pas tomber dans la paranoïa et ne protéger que les formulaires vraiment sensibles (par exemple, il ne sert absolument à rien de protéger un formulaire de recherche).

#### 2.8.2.4 Configuration

La configuration se fait en deux étapes. Il faut d'abord prévenir le framework qu'un test est nécessaire en renseignant le paramètre *use-token* et le paramètre *input-forward* (voir tableau 2.5) dans l'élément input du descripteur de l'action concernée. Ensuite, dans le formulaire d'où doivent provenir les données il faut appeler une fonction nommée token() qui se chargera d'insérer un champs caché contenant le nonce. Si le token doit être inséré dans une url, il faut utiliser la fonction url_token().

.

## 2.9 Problèmes rencontrés

### 2.9.1 Problèmes du langage PHP

Pour utiliser le framework, le développeur implémentera des classes étendant la classe *Action* du framework. Ces classes devront être chargées dynamiquement par le framework durant l'execution d'une requête. Comme il n'existe pas de mécanismes d'introspection en PHP, une autre solution a été mise en place. Une classe étendant la classe *Action* devra être située dans un fichier portant le même nom que la classe suivi de l'extension *.class.php* (par exemple une classe nommée TestAction sera située dans un fichier TestAction.class.php). Quand cette action devra être éxécutée, le framework incluera le fichier correspondant, créera un objet et exécutera la méthode *execute()* de cet objet.

Dans les servlets en java, il est possible de rediriger des URL vers un servlet spécial. Par exemple il est possible de faire executer toutes les URL se terminant par *.do* par un servlet. Cela est très pratique pour faire exécuter un certain type d'URL par un contrôleur principal. Comme il n'existe pas de mécanisme aussi évolué en PHP, l'appel du contôleur principal doit se faire d'une autre façon : on appelle qu'une page nommée index.php et le nom de l'action à effectuer et passé en paramètre dans l'URL. Une URL permettant de déclencher une action est donc de la forme :

```
http://www.monsite.org/index.php?action="foo"
```

PHP n'est pas un vrai langage objet, il lui manque un certain nombre de concept qui peuvent de révéler assez gênant.

Tout d'abord, il n'y a pas d'encapsulation et cela pose des problèmes de protection des données et certains objets du framework nécessitent des données privées uniquement accessibles via des accesseurs (des methode get et set ).

Il n'y a pas non plus de notion d'interface, ni de classe abstraites. Cela n'est pas vraiment un problème car les variables n'étant pas typées, il est très simple d'obtenir le même effets que le polymorphisme. Mais l'implémentation n'est pas forcement très "propre".

Il n'y a pas d'exceptions. Il est nécessaire de coder une classe de gestion d'erreur permettant d'émuler plus ou moins les exceptions et permettre de faire remonter les erreurs proprement. Mais la gestion des erreurs reste peu pratique et ne remplace pas les exceptions.

Un autre problème existe pour les fichiers de configuration. Ceux-ci étant en XML il serait intéressant de définir une DTD afin de pouvoir valider ces fichiers lors de leur lecture. Mais la vérification d'un fichier XML par une DTD dans PHP 4 n'est pas aisée, il faut utiliser le module DOMXML qui est considéré comme expérimental (et donc impropre à l'utilisation en production) et mal documenté.

La plupart des problèmes liés au modèle objet de PHP seront réglés dans la version 5 de PHP. En effet cette version fournira un vrai langage objet avec l'encapsulation, les interfaces, et les exceptions. De plus une nouvelle API nommée simpleXML sera disponible et permettra de manipuler facilement les fichier XML. Il sera donc possible de faire valider par une DTD les fichiers de configuration XML.

### 2.9.2 Problèmes de performances

Php est un langage interprété : à chaque requêtes, le fichier PHP appellé est lu, compilé, puis exécuté. Ce mode de fonctionnement pose un problème de performance qui peut être en partie réglé en utilisant des modules comme Turck MMCache[2] ou Zend accelerator [3] qui permettent de mettre en cache le code compilé par php.

Mais ces modules de cache ne règlent pas tous les problèmes. Il n'est pas possible d'avoir un contrôleur unique, chargé au démarrage de l'application, comme il est possible de faire avec les servlets en java. En PHP le contrôleur principal doit être rechargé à chaque appel ce qui pose des problèmes au niveau de la configuration. En effet la configuration du framework et des actions se font dans des fichier XML et comme il faut les recharger à chaque appel (et donc les *parser*), cela se révèle très coûteux en temps de traitement. Pour régler ce problème, les fichiers XML sont préalablement traduits en fichiers de variables PHP qui sont eux inclus dans le framework. Ce qui signifie que pendant la phase de développement, il est possible de *parser* les fichiers de configuration à chaque appel alors qu'en production, le *parsing* n'est plus effectué et l'on se contente d'inclure les fichiers de variables précédemment générés, ce qui permet d'obtenir de bonnes performances.

Mais la traduction préalable ne règle pas tous les problèmes de performance. En effet si on utilise un fichier de configuration unique pour décrire toutes les actions du framework (comme dans le cas de Struts), plus le nombre d'action sera grand et plus le fichier de variable à inclure sera important, ce qui diminuera les performances. Deux solutions sont envisageables.

La première solution consiste à mettre chaque descripteur d'action dans un fichier séparé et de n'inclure que le fichier de configuration de l'action devant être éxécutée. Le problème est qu'il en résulte un grand nombre de fichiers de configuration.

La deuxième solution consiste à regrouper les actions en fonctions des cas d'utilisation auquelles elles appartiennent. Les actions d'un même cas d'utilisation seront situées dans un répertoire portant le nom du cas d'utilisation et un fichier de configuration situé dans ce répertoire décrira toutes les actions. Ainsi, seul le fichier de configuration du cas d'utilisation de l'action devant être

[2] `http://turck-mmcache.sourceforge.net/`
[3] `http://www.zend.com/store/products/zend-accelerator.php`

éxécuté sera chargé. Le problème de cette méthode est que pour appeller une action il faudra deux paramètres : le nom de l'action et le nom du cas d'utilisation auquelle elle appartient. Un appel d'action deviendra donc :

```
http://www.monsite.org/index.php?usecase="foo"&action="bar"
```

C'est la deuxième solution qui a été retenue car bien qu'elle rajoute un paramêtre à la requête, elle permet d'obtenir une bonne structuration interne de l'application.

# Bibliographie

[CRAN00] CRANE (Aaron), "Experiences of using PHP in large websites", document au format pdf, 2000, http ://www.gbDirect.co.uk, 11 pages.

[BIFA02] BIOT (Nicolas), FAUVEAU (Armel), "Mise en oeuvre de FastTemplate, PHPlib, Vtemplate, Smarty et Modelixe", document au format pdf, février 2002, http ://www.phpindex.com/download/Templates_V1.2.pdf, 66 pages.

[GIRA01] GIRARD (Didier), RIBETTE (Jean-Noël), "OpenSource-Java : MVC2 avec Struts", document au format pdf, mai 2001, http ://www.application-servers.com/pagePublications.jsp, 7 pages.

[HUST02] HUSTED (Ted), DUMOULIN (Cedric), FRANCISCUS (George), WINTERFELD (David), "Struts in Action" ,manning, Novembre 2002, 630 pages.

[POSA96] BUSCHMANN (Frank), MEUNIER (Regine), ROHNERT (Hans), SOMMERLAD (Peter), STAL (Michael), "Pattern-oriented software architecture, volume 1" ,manning, 1996, 467 pages.

[OWAS04] "Open Web Application Security Project : The ten most critical web application security vulnerabilities" OWASPTopTen2004.pdf, 2004, http ://www.owasp.org/documentation/topten, 45 pages.

# Table des figures

# Liste des tableaux